Runtime Code Generation

by

David Keppel

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1996

Approved by____

Chairperson of Supervisory committee.

Program Authorized
to Offer Degree____

Date____

Doctoral Dissertation

Signature ______________________

Date ______________________

University of Washington

Abstract

Runtime Code Generation

by David Keppel
Chairperson of the Supervisory Committee: Professor Susan J. Eggers
Department of Computer
Science and Engineering

This dissertation is about *Runtime Code Generation*, or just *RTCG*, which is the general idea of introducing new code into a running program. Although the idea of RTCG predates the first programmable-store computers, the tradeoffs between statically-generated and dynamically-generated code have been at best under-reported and at worst poorly-understood. The goal of this dissertation is to demystify RTCG and turn it in to "just another tool" in the programmers toolbox, with a well-understood set of tradeoffs.

Historically, RTCG has been used to improve various aspects of code such as performance, flexibility, modularity, code reuse and portability. However, RTCG doesn't add any fundamentally new powers to a computer. Thus, the ultimate utility of RTCG hinges on performance — the ability to improve performance or the ability to maintain performance while improving some other metrics such as flexibility or portability. Part I of this dissertation studies how RTCG can produce code that is faster than statically-typed, statically-compiled and statically-optimized programs. Part II of this dissertation addresses more general concerns: it uses the results of Part I to explain how RTCG can improve other metrics, such as flexibility and portability, without hurting performance; it investigates instruction cache coherency; and it provides an extensive history of runtime code generation, starting with early programmable computers.

Contents

List of Figures

List of Tables

Many people deserve thanks for their help with this dissertation: my parents, Barbara and C. Robert Keppel; lifelong mentors, including Milt Raymond, Jerry Pearson and Bing Chen; my academic advisors, especially Susan J. Eggers and Robert R. Henry; people who paved the way for my work, especially Henry "Qua" Massalin, whose research first kindled my interest in an the area that has turned into my degree, Peter Deutsch, whose work rekindled my interest, and Bob Cmelik, whose work gave me access to the guts of a large system. Finally, several hundred people have provided direct technical contributions and other kinds of support, while my friends have contributed peace, interest and inspiration. These people have acknowledged the small help I have given them, but so many people have helped me that I cannot acknowledge each of them here — I hope they will forgive me now, and that later I can better acknowledge them individually. Some of my challengers also deserve credit for posing hard questions and never being satisfied with the answers. Finally, I take advantage every day of the work of people who have struggled to make the world a better place — these people, most of all, are the giants upon whose shoulders I stand.

Chapter 1

Introduction

All constants are variable and all variables are constant.

–Finagle's Law

Runtime code generation (*RTCG*) is the general area of introducing new code into a running program. Runtime code generation has been used in a wide variety of systems to improve metrics such as performance, portability, interoperability, ease of construction and debugging. This dissertation studies RTCG at two levels. First, Part I studies *runtime-compiled value-specific optimization* (*VSO*), in which code is dynamically optimized around the current values of program variables. Part I studies the transformations and implementations needed for VSO, demonstrates that it produces speedups over statically-optimized, statically-typed code, and examines the effects of processor architecture and runtime optimization level on VSO's profitability. Part II then examines RTCG in the more general context of dynamic optimization and code modification. It provides basic definitions for discussing RTCG and then examines a wide variety of code modification issues, RTCG design and implementation techniques, some traps and pitfalls that have plagued RTCG over the years, and provides references to a broad base of related work.

1.1 RTCG Opportunities and Costs

RTCG provides dynamic change of representation. The opportunity that it provides is a mechanism for converting from generic and abstract forms to faster-executing, invocation-specific concrete forms. The intuition is that RTCG can dynamically optimize around information that is unavailable during static compilation, such as the platform's architecture and implementation and the types and values of runtime data. The cost of RTCG is that dynamic code generation takes a part of the program's execution time, so the optimized code must run for long enough or run enough faster that it more than pays for the code generation costs.

For example, Figure 1.1 shows a simple model of code performance, with execution time a function of the problem size. The *asymptotic speedup* is the speedup for large problem sizes. For long-running problems, the compilation cost is small compared to the savings in execution time from using dynamically-optimized code. The *breakeven point* is the smallest problem size for which

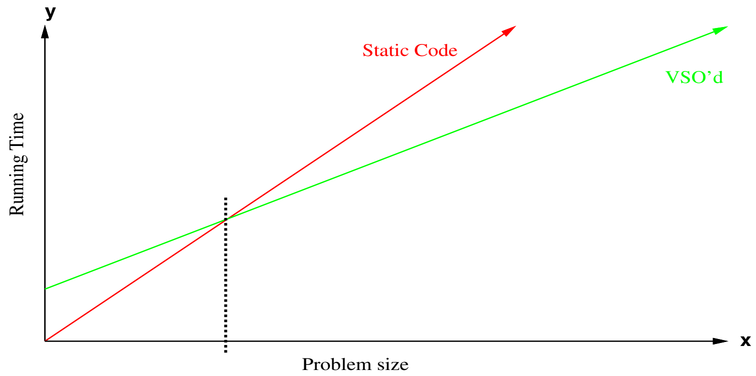


Figure 1.1: A simple model of code performance.

The steeper line is slower per invocation, but the shallower line is more expensive to create. Thus, the steeper line is used to the left of the vertical *breakeven point* and the shallower line is used to the right.

dynamically-generated code is faster. The observation here is that the faster RTCG'd code must pay back the runtime compilation cost before the application as a whole shows speedups.

It might seem, intuitively, that purely static optimization should produce the fastest code, because even long-running static optimizations can be performed without consuming any of a program's execution time. However, many static optimizations make the code larger, as well as faster, because the static compiler produces code fragments for each of many value combinations that *might* arise at runtime. Over-ambitious static optimization thus leads to a code explosion that causes programs to grow unacceptably. In comparison, dynamic optimization can wait until the actual values are known and then generate code as good as huge static code but without the space cost.

For example, Figure 1.2 shows a loop in which the variable `x` is set outside the loop, and then used in each iteration of the loop to perform control flow. Each time the loop is invoked, there are `M` iterations of the loop, but only the first iteration's control flow does "real" work; each of the following `M-1` iterations each go to the same case as on the first iteration.

A static compiler can apply optimizations such as *loop inversion*, which interchanges a loop with control flow inside the loop. It can be applied here since the control flow depends only on the variable `x`, set outside the loop; the result is shown in Figure 1.3. Loop inversion is an example of *static special-casing*, where (approximately) the entire loop is replicated for each "arm" of the control flow construct, but each loop contains only the code for the arm selected by the outer control flow. In this example, static special casing eliminates the `switch` statement from the middle of each loop, but at the expense of space: there are now `N` copies of the `for` loop and of statement `S0`. Worse yet, specializing statically over several variables gives code growth that is exponential in the number of variables (§B.1), so static code size rapidly becomes unmanageable.

Although there are exponentially many possible cases for a fragment, only one of them is in

```
for i=1..M {
    S0
    switch (x) {
    case 1: S1
    case 2: S2
      ...
    case N: SN
    }
}
```

Figure 1.2: A loop with embedded control flow.

The variable x is changed only outside the loop, so for each M iterations of the loop, the switch always goes to the same case.

```
switch (x) {
case 1: for i=1..M { S0 S1 }
case 2: for i=1..M { S0 S2 }
  ...
case N: for i=1..M { S0 SM }
}
```

Figure 1.3: An example of loop inversion.

Loop-invariant control flow inside of a loop is moved outside the loop by dispatching once to a specialized copy of the original inner loop.

use at any given time. Dynamic optimization can wait until runtime, examine the values of the key variables, then compile just the case that is actually needed, discarding it when it is no loner needed. For example, Figure 1.4 shows the dynamically-generated code when x has the value 2. This code is identical to case 2 in Figure 1.3, but only a single specialized case, and thus only a single for loop and S0, exist at any given time.

One key observation behind dynamic optimization is that the runtime compilers perform traditional optimizations, such as constant folding and loop unrolling, but, since the optimizer has more information, the optimizations are more effective. For example, a static compiler might know the type and location of a variable; as shown above, a runtime optimizer can also know the value and

```
for i=1..M {
    SO S2
}
```

Figure 1.4: An inner loop implementation, generated on demand.

The inner loop code that is generated depends on the current value of x. The inner loop is as fast as with loop inversion, but this implementation needs only one copy of the for loop and of S0.

can use the actual value to produce better code. A second key observation is that where static specialization is limited by space, dynamic optimization is limited by time. Thus, in order to be profitable, the code improvements must more than pay for the compilation cost.

1.2 Dissertation Overview

Part I of this dissertation focuses on VSO, which is dynamic optimization using the current values of variables. The dissertation investigates the opportunities for performance improvements in statically-compiled, statically-typed uniprocessor programs.[1] The experiments presented here show general characteristics of programs that are amenable to VSO; they also study tradeoffs between various VSO implementations.

The fundamental question of VSO is: "Is it worth it?" That is, the runtime cost of optimization must produce code that is enough faster that it improves the *overall* program execution time, despite the added cost of compilation. One contribution of this dissertation is to perform a feasibility study of VSO. These studies use several hardware platforms based on contemporary RISC processors, and show that VSO can lead to overall speedups, measured as high as 2.5 over highly-tuned static code. The results also show cases where VSO leads to slowdowns, because compile times do not get amortized or because of instruction cache self-interference caused by the runtime-generated code.[2]

Dynamic compile time is crucial to overall performance: reducing VSO's dynamic compile costs reduces the breakeven point, which in turn improves the range of inputs where VSO is profitable, or, equivalently, improves VSO's profitability for given problem size. A second contribution of this dissertation is to experimentally validate the model in Figure 1.1 and to experimentally study the significance of code generator speed. The results show that some applications can take advantage of today's general-purpose retargetable code generators, but that other applications need faster code generators to be profitable across a wide range of inputs.

Intuitively, code that is generated quickly will be of low quality. Increasing the level of runtime optimization should improve the code quality and thus the asymptotic performance benefit. There is a risk, however, that performing more optimizations will also inflate the dynamic compile time so much that the faster code can't pay back the compile costs. Alternatively, performing too few optimizations may hurt the code quality so much that it is slower than statically-optimized code. A third contribution of this dissertation is to experimentally study the effects of optimization level on overall performance. A perhaps surprising result is that increasing optimization levels rarely produce better code – that is, fast and simple code generation can produce fast-running code. The insight here is that VSO tends to replace variables with constants, and thus replace indefinite information with definite information. As a result, simple runtime constant-propagation optimizations are very effective.

[1] "Statically-typed" is used in the sense of C or FORTRAN; that is, programming languages lacking constructs that result in dynamic type dispatch. The original programs are thus already free of type dispatch overhead and improvements are due soley to eliminating value dispatch overhead. See Appendix A for a more precise definition.

[2] The slowdowns due to self-interference are arguably an artifact of the implementation, but nonetheless interesting, because they arise in a way that wouldn't occur in systems using purely static code.

Although VSO often leads to substantial speedups, the improvement sometimes falls short of what might be predicted from comparing the number of instructions executed by the static and VSO'd versions. For example, in one case a VSO'd application executed 40% fewer instructions but showed only a 20% speedup. The difference might be due to a variety of factors – changes in the mix of instructions executed by the VSO'd version, interaction of hardware features such as branch prediction, caches, and so on. A fourth contribution of this dissertation is to analyze the low-level behavior and show that such subtle interactions were rarely relevant. In most cases, the bottlenecks were fixed overheads that were present in both the static and VSO'd versions, such as limited memory bandwidth and I/O costs. With VSO, the absolute costs were unchanged, but since the costs were amortized over shorter runs, they were relatively more significant.

A fifth contribution of this dissertation is pedagogical. Part II examines RTCG in the more general context of dynamic optimization and code modification. It provides more precise definitions for discussing RTCG, including terms such as "runtime", "code" and "generation". It then examines a wide variety of code modification issues, RTCG design and implementation techniques, reports on some traps and pitfalls that have plagued RTCG over the years, and provides references to a broad base of related work. The study thus serves as a framework for identifying, understanding and categorizing uses of RTCG.

1.3 Outline

The remainder of the dissertation is organized as follows. In Part I, Chapter 2 explains the basic operation of VSO and shows, in more detail, why VSO provides speedups and how static code can be transformed for VSO. Chapter 3 describes the implementation used for this dissertation, including details of the transformation, memory allocation, code generation strategies, and instruction cache management. Chapter 4 describes the applications used as benchmarks and the datasets used to drive the program runs; the measurement techniques, based elapsed time measurements, tracing, and simulation; and the platforms used for the studies. Chapters 5 and 6 present performance results; the first chapter reports the measured execution times and observed optimizations, while the second reports low-level measurements, such as changes in the instruction mix and component costs. Chapter 7 describes future work: how VSO could be recognized and transformed automatically, and how static estimation errors would affect the profitability of the resulting system. In Part II, Chapter 9 provides specific definitions relating to RTCG and Chapter 10 provides an extensive survey of RTCG system design and implementation issues and techniques. Chapter 11 investigates the implementation and cost of instruction cache flushing on a variety of machines and describes a portable interface that provides portability while at the same time maintaining performance close to that of hand-written machine-specific coherency implementations. Finally, Chapter 12 concludes.

Part I

VSO Experiments

Chapter 2

Transformation

Representation <u>*is*</u> *the essence of programming.*

–Fred Brooks, *The Mythical Man-Month* [Brooks 75]

This chapter describes how a source code fragment is transformed so that it can be optimized at runtime around the values of variables. Section 2.1 first discusses limitations of static optimization and some of the opportunities exposed by these optimizations. Section 2.2 describes *application-specific compilers*, which execute application operations and use the results to optimize and generate code that executes the remaining operations more efficiently. Section 2.3 describes various implementations of application-specific compilers and general tradeoffs in ease of construction, code generator performance, and quality of the runtime-generated code. These transformations also potentially introduce instruction cache coherency problems; Section 2.4 describes how coherency is ensured.

2.1 Static and Dynamic Optimization

Static optimization is fundamentally constrained by the limited amount of information that is available statically. To illustrate the limitations of static optimization, this section introduces a running example and shows how it can be optimized, first by using static application-specific information (§2.1.3), and then by using *invocation-specific* runtime information to perform VSO (§2.1.4). The final VSO implementation is as general as the slower and more general static code, yet it executes roughly half the number of instructions of the faster and less general static code.

2.1.1 An Example: Sorting Records

Consider a hypothetical application that manages a database of houses for sale and allows users to interactively sort and display lists of houses in various orders. Each house is described by a record, such as is shown in Figure 2.1, and the user describes a sorting preference by naming fields in some order. For example, the *sort vector* {NBEDRM, PRICE} represents a user command to sort first by the number of bedrooms and then subsort by price.

The sorting example makes use of a standard sorting routine, qsort, that in turn calls a *comparison predicate* that compares pairs of records. The comparison predicate takes three arguments, the two

```
record {
  int nbedrm        /* Number of bedrooms */
  int nbathrm       /* Number of bathrooms */
  char *nbrhood     /* Name of the neighborhood */
  int price         /* Listed price */
  int sqm           /* Floor area, meter^2 */
  float taxrate     /* Annual tax rate */
  int yardsqm       /* Size of lot, meter^2 */
   ...              /* ... and so on */
} house_t
```

Figure 2.1: An example record that describes a house.

records and the sort vector. The predicate returns an integer that is negative, zero or positive, depending on whether the first record is less than, equal to, or greater than the second record, respectively.

Profiling the application shows that a substantial fraction of the total time is spent in the comparison predicate. Although the sorting algorithm, data layout and so on might also be improved, such optimizations also increase the percentage of time spent in the predicate, and thus the relative importance of the predicate to the overall execution time. Thus, the focus here is on optimizing the comparison predicate.

2.1.2 Generic Record Comparator

Records can be compared using a generic library routine; the routine is parameterized by a list of <*type*, *offset*> tuples. That is, the sort vector {NBEDRM, PRICE} is translated to a list such as:

```
{integer, offsetof (nbedrm, house_t)},
{integer, offsetof (price, house_t)},
{done, 0}
```

The above list says that the sort predicate should first compare two integer fields that appear in each record at an offset of offsetof (nbedrm, house_t). For a pair of records, if the nbedrm fields are different, the predicate can return immediately. If the fields are identical, then the predicate compares integer fields at offsetof (price, house_t). Again, if they are different, the predicate returns; otherwise the predicate has reached the end of the vector and so it returns 0 to indicate the records are equal for the purpose of this comparison.

Figure 2.2 shows a prototypical static code function that compares two records by comparing fields in an order described by a list of <*type*, *offset*> tuples stored in the global variable vec. For efficiency, *type* is encoded as a pointer to a corresponding code fragment [Bell 73, Eckhouse Jr. & Morris 79, Kogge 82, Pittman 87], INT, CHAR, etc., with DONE indicating no more fields to compare.

The execution cost varies with the record values, but if the nbedrm fields are equal and the

```
cmp:                        ! call cmp(p0,p1)       Iteration
        mov vec → r1        ! tuple list            1
L0:     ld (r1) → r2        ! get <type>            1    2
        ld 4(r1) → r3       ! get <offset>          1    2
        add #8, r1          ! ptr++                 1    2
        j *r2               ! dispatch              1    2

INT:                        ! int compare
        add p0,r3 → r9      ! r9 := field address   1    2
        ld (r9) → r4        ! load from record 0    1    2
        add p1,r3 → r9      ! r9 := field address   1    2
        ld (r9) → r5        ! load from record 1    1    2
        sub r5,r4 → r0      ! compare fields        1    2
        bz r0,L0            ! continue if =         1    2
        ret                 ! else return <, >           2
CHAR:
        add p0,r3 → r9      ! char compare
        ldb (r9) → r4       ! ...
        add p1,r3 → r9      ! ...
        ldb (r9) → r5       ! ...
        sub r5,r4 → r0      ! ...
        bz r0,L0            ! continue if =
        ret                 ! return <, >

... ! other compare types

DONE:
        mov #0 → r0         ! records are =
        ret
```

Figure 2.2: A prototypical static code function that compares two records.

Registers p0 and p1 hold pointers to the records being compared; register r0 holds the return value, indicating which record is less. The numbers on the right indicate which instructions are executed, and in which iteration, when the predicate is invoked with the tuple list {INT,0}, {INT,24} and when comparing two records where the first pair of fields have identical values and the second pair of fields are different; twenty-two instructions are executed.

price fields are different, execution will take the 22 instructions, as shown in Figure 2.2. Note that there is some overhead for fetching the *offset* field on each iteration; machines with register+register addressing will execute the above in fewer instructions, but still execute more instructions than if the comparator is specialized to the application, as described below.

2.1.3 Static Specialization

The generic record comparator is useful for a wide variety of applications, but needlessly general for any given application. An application-specific comparator eliminates the overhead of fetching and adding each *offset*; it replaces each generic field type comparison, such as INT, CHAR, and so on, with a structure-specific comparison that is optimized to the type and offset of each structure

```
cmp:                       ! call cmp(p0,p1)       Iteration
        mov vec → r1       ! tuple list            1
L0:     ld (r1) → r2       ! get <type>            1     2
        add #8, r1         ! ptr++                 1     2
        j *r2              ! dispatch              1     2

INT_NBEDRM:                ! int compare, off=0
        ld 0(p0) → r4      ! ...                   1
        ld 0(p1) → r5      ! ...                   1
        sub r5,r4 → r0     ! ...                   1
        bz r0,L0           ! continue if =         1
        ret                ! return <, >
INT_NBATHRM:               ! int compare, off=4
        ld 4(p0) → r4      ! ...
        ld 4(p1) → r5      ! ...
        sub r5,r4 → r0     ! ...
        bz r0,L0           ! continue if =
        ret                ! return <, >
...
FLOAT_TAXRATE:             ! float compare, off=24
        ld 24(p0) → r4     ! ...                         2
        ld 24(p1) → r5     ! ...                         2
        sub r5,r4 → r0     ! ...                         2
        bz r0,L0           ! continue if =               2
        ret                ! return <, >                 2
...
DONE:
        mov #0 → r0        ! records are =
        ret
```

Figure 2.3: Application-specific static code that compares two records.

The application-specific static code embeds the layout of particular records. The code is similar to that in Figure 2.2, but each field comparison fragment is specialized to both the type and the offset, where the fragments in Figure 2.2 will work for any offset. Numbers on the right correspond to those in Figure 2.2, but only sixteen instructions are executed.

element. The resulting code has separate comparison code for each structure element. For example, even though the NBEDRM and NBATHRM fields are both int types, they are at different offsets in the structure, and so each has code optimized around the particular offset. The resulting comparator executes fewer instructions, because the code for INT is copied and specialized for each field offset. The dispatch operation on *type* is replaced with a dispatch on *name*, which encodes both *type* and *offset*. Thus, the corresponding runtime load and add of *offset* are avoided.

The resulting code is also potentially larger, because each INT comparison fragment in the generic comparator is replaced by a group of offset-specific fragments, INT_NBEDRM, INT_NBATHRM, and so on. Figure 2.3 shows an application-specific version of the code in Figure 2.2. For a particular sort vector and a particular pair of records, the application-specific version executes only 16 instructions instead of the generic version's 22.

In principle, it is possible to further optimize this code by special-casing and on each possible sort order, for example, `cmp_NBEDRM_PRICE` to compare the number of bedrooms and then the price. However, for a record with F fields there are more than $F!$ such comparators (§4.1.4). For example, just 10 fields gives more than one million comparison functions. Thus, extensive special casing is, in practice, impossible because of the space overhead.[1]

Extensive special-casing may be viewed as a speculative transformation that attempts to provide specialized code for each path that *might* arise at runtime. Clearly, however, some possible paths are exercised rarely or never at all. An alternative is *common-casing*, which attempts to special-case the subset of cases that occur most frequently. However, common-casing still suffers from code growth (§B.1); for example, common-casing of just one tenth of one percent of the cases for sorting ten fields still generates over 10,000 comparison functions. Yet, common-casing of too few special cases hurts coverage and means that a large number of invocations must use the slower general code, resulting in reduced performance. Also, reducing the number of special cases makes performance more sensitive to differences between the profile or heuristic used to select the specialized cases. Likewise, changes in workload may invalidate the assumptions used in deriving the common cases. In the worst case, a new workload may use the general code *every* time, reducing performance to that of unspecialized code.

For any given invocation, however, the sorting routine needs just one comparison function. Since "which one" depends on runtime information, static optimization is limited. However, more information is available at runtime, so we turn to runtime construction of the comparison predicate.

2.1.4 Dynamically-Compiled Record Comparator

The sort vector changes at runtime, but it is constant over any given call to `qsort` and thus is constant over many calls to the comparison predicate. Since some operations in the predicate depend only on the sort vector, the comparator may be optimized around the current value of the sort vector and reoptimized (recompiled) when the sort vector changes. Thus, it is possible to generate highly specialized code without the space explosion, because the specialized predicates are generated on demand and discarded again before generating new code.

For example, the vector {NBEDRM, PRICE} means that the first iteration always dispatches to code to compare the number of bedrooms, independent of the record values. The second iteration, if executed, always compares the price fields, and the third iteration, if executed, always returns `0` to indicate record equality. Thus, for a given sort vector it is possible to build a special-purpose comparator that always compares fields in one fixed order.

The underlying theory of evaluating code with respect to some of its arguments is called *partial evaluation* (§10.7). The interesting idea here is to apply partial evaluation at runtime. Creating optimized code at runtime is discussed in the next section (§2.2); for now consider just the final code

[1] In principle, it is even possible to special-case the generic record comparator (§2.1.2), by having one case for integer fields at offset `0`, one for integer fields at offset `4`, one for `8`, and so on. However, specializing the generic comparator fails for some records, because the generic record comparator needs to be able to handle records of arbitrary size. Even for bounded record sizes, specializing the generic comparator results in more cases than specializing the application-specific comparator.

```
codebuf:                   ! cmp for NBEDRM, PRICE    Iteration
M0:   ld 0(p0) → r4        ! NBEDRM from record 0     1
      ld 0(p1) → r5        ! NBEDRM from record 1     1
      sub r5,r4 → r0       ! compare fields           1
      bz r0,M1             ! continue if =            1
      ret                  ! else return <, >
M1:   ld 24(p0) → r4       ! PRICE from record 0          2
      ld 24(p1) → r5       ! PRICE from record 1          2
      sub r5,r4 → r0       ! compare fields               2
      bz r0,M2             ! continue if =                2
      ret                  ! else return <, >             2
M2:   mov #0 →r0
      ret
```

Figure 2.4: A dynamically-compiled value-specific record comparison routine.

The routine first compares two integer fields at offset 0, then two integer fields at offset 24. The VSO implementation is fully unrolled and has no dispatching on the sort vector. The numbers to the right correspond to the iteration numbers shown in Figure 2.2 and 2.3; the dynamically-compiled routine executes only half as many instructions as the faster application-specific static code.

that is produced at runtime. Figure 2.4 shows dynamically-compiled code that implements sorting for the sort vector {NBEDRM, PRICE}.

A variety of optimizations make this code better than the statically-optimized code. Since the sort vector's length is known, the inner loop can be fully unrolled. Since the type and offset for each iteration is known, all dispatching on those values can be removed. The only operations left in the final code are those that depend directly on values in the structures: loading, comparing, and branching on the values of structure elements.

The VSO'd code thus executes nine instructions, where static code versions executed 16 and 22. Of course, these savings do not directly double the performance of the application, since the predicate is only one part of the application's core code, and since the above code does not include the execution time required to create the predicates on demand. Nonetheless, the savings do substantially improve the overall performance of the program.

2.2 Application-Specific Compilers

The preceding section shows by example that VSO'd code can be faster than its static counterpart. However, as discussed in Section 1.1, the profitability of VSO depends on being able to optimize and generate code quickly: VSO'd code is faster than static code only when the VSO code improvements more than pay for the cost of compiling it dynamically. This section shows how VSO can be implemented using an *application-specific dynamic compiler* (or just *application-specific compiler*) that is limited in what it can compile, but which is specialized and thus fast. This section shows

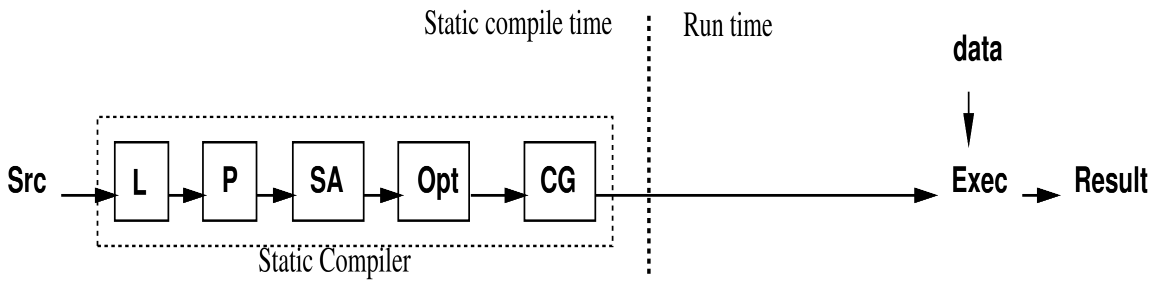


Figure 2.5: Schematic diagram of a conventional compiler.

A "traditional" compiler performs all code analysis at *static compile time*, before any of the program's data becomes available.

how the front end of such a compiler is derived from the source code;[2] the following section (§2.3) describes ways that a dynamic compiler can be optimized for fast code generation.

2.2.1 Static Analysis

Figure 2.5 shows a schematic of compilation with a traditional static compiler. The input program is lexically analyzed, parsed, semantically analyzed and optimized. Eventually executable code is produced. In the example of the previous section, the statically-compiled code would be either the generic record comparator (§2.1.2) or the application-specific comparator (§2.1.3). The important characteristic of fully static compilation is that all code generation is performed before the program is invoked. At runtime, the static code executes and consumes the input data to produce the program results.

Figure 2.6 shows a schematic of compilation with a compiler that performs VSO. During static compilation, most of the program is compiled normally, but the static compiler also produces a dynamic compiler. At program runtime, the dynamic compiler executes and in doing so consumes some of the data values to produce code that is optimized around them. The resulting code is then executed; it consumes the remaining input data and produces the final results.

Within an application code fragment, some variables change frequently, while others may change only rarely. When a variable changes slowly but is used in an inner loop, the value of the variable is constant across many iterations of that loop. Thus the value[3] of the variable may be used to generate special-purpose code for the loop; these variables are thus known as *specialization variables*. Optimized code is generated from the current values of the specialization variables by segregating the loop into operations that depend soley on the specialization variables and operations that also depend

[2]Application-specific compilers may be specified explicitly [Engler et al. 96], may be generated by the static compiler based on programmer hints in the source code [Auslander et al. 96, Leone & Lee 96b, Consel & Noel 96], or may be deduced from the source code by the static compiler. The ideas in this section apply to all of these cases but the discussion is oriented towards automated transformation.

[3]The line between type and value is at best blurry. For example, a subrange might be considered either a generalization of value or a specialization of type.

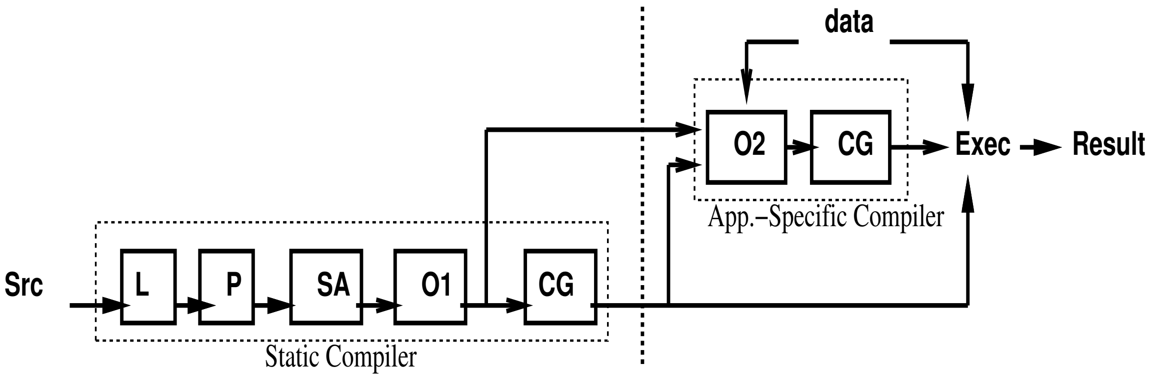


Figure 2.6: Schematic diagram of a compiler that implements VSO.

A VSO compiler performs most analysis and code generation statically, but some analysis and code generation is performed at runtime. To reduce the runtime compiler's execution time, the static compiler produces an *application-specific compiler*. The application-specific compiler is specialized to compile only constructs that appear in the particular application, and because it is less general, is also faster.

on other values. Operations that depend only on specialization variables are executed outside of the loop and are used to produce new code for the inner loop. The code that evaluates the specialization variables and produces new code is an application-specific compiler.

The application-specific compiler can be derived as follows. Given some specialization variables (choosing such variables is discussed in Chapter 7), the static compiler uses, e.g., flow analysis to identify operations that depend only on the specialization variables. Those operations are executed by the application-specific compiler and are used to control code generation. Those operations that also depend on other values are also copied into the application-specific compiler, but instead of being executed by the compiler they appear as directions to emit code; they are thus *deferred* by the application-specific compiler[4] and are instead executed by the code that it produces [Leone & Lee 94, Auslander et al. 96, Leone & Lee 96b, Consel & Noel 96].

2.2.2 An Example

The record comparison example of the previous section is good for showing how static optimizations cause space explosion, but it is too large for analyzing static compiler transformations in detail, because the examples rapidly get cumbersome. Thus, the example for this section is the one shown previously in Figure 1.2.

The example code is a `for` loop containing a `switch` that depends only on a variable `x`, set outside of the loop. Figure 2.7a shows the example code with a box around those operations that depend

[4] In partial evaluation terminology, the application-specific compiler is the *generating extension* of the original code when specialized with respect to the specialization variables, and the code that is generated at runtime is the *residual* code.

```
for i=1..M {
    S0
    switch (x) {
    case 1:   S1
    case 2:   S2
      ...
    case N:   SN
    }
}
```

(a)

```
for i=1..M {
    S0
    switch (x) {
    case 1:   S1
    case 2:   S2
    ...
    case N:   SN
    }
}
```

(b)

Figure 2.7: Data-dependence of operations.

Operations that depend (a) only on x and (b) on values besides x. Operations that depend solely on x may be performed as soon as the value of x is known. Those that also depend on other values must wait until the remaining values are known.

```
emit ("for i=1..M {")
emit ("S0")
switch (x) {
case 1: emit ("S1")
case 2: emit ("S2")
  ...
case N: emit ("SN")
}
emit ("}")
```

(a)

```
for i=1..M {
  S0
  S2
}
```

(b)

Figure 2.8: An application-specific compiler and example output.

This compiler generates code for the source fragment of Figure 2.7. When called with x equal to 2, it produces the code (b).

only on x. The boxed code may be executed as soon as the value of x is known. Figure 2.7b shows the same code with boxes around the operations that also depend on other values. Those parts can only be executed later, when the additional data values are available.

An application-specific compiler derived from the above fragment is shown in Figure 2.8a. The application-specific compiler directly executes the boxed operations in Figure 2.7a, and as it runs it generates code for the boxed operations in Figure 2.7b. Figure 2.8b shows the code that the compiler produces for the case where x is 2.

For clarity, this section has described the derivation of the "front end" of an application-specific compiler, such as is shown in Figure 2.8a. The output is source code text, as shown in Figure 2.8b. The "back end" can be thought of as a conventional compiler that produces relocatable object code which is then dynamically linked with the running application. The next section shows how a static compiler can produce application-specific compilers that emit machine code more directly, thus improving compilation times.

2.3 Tree and Template Compilers

The overall performance of VSO is affected by both the runtime compiler's speed and the quality of the code that it produces. This section describes two application-specific compilers that differ in implementation, code generator technology and code generator performance. The first implementation is a *tree compiler* that produces a tree-like internal representation and then passes it to a retargetable code generator to produce executable code.[5] The second is a *template compiler* which generates code by concatenating and patching fragments of machine code called *templates*. The tree compiler is easier to build using existing tools (in particular, it can use existing retargetable code generators) and makes fine-grained optimization decisions and thus may produce higher-quality code. However, the template compiler can produce code more quickly, as fast as a few instructions per generated instruction.

Another way to view the compilers is that the intermediate representation or *IR*, which is the form manipulated by the runtime compiler, may be either machine-independent or machine-dependent, and either application-independent or specialized to the application [Henry 87]. Static specialization produces faster compilers, but can reduce the runtime compiler's flexibility and thus the quality of the final code; in the trivial limit when all work is performed statically, the code quality is that of a static compiler. The tree compiler is an example of an implementation using IR nodes that have no machine or application specialization. In contrast, the template compiler's nodes are specialized to both the machine and the application.

The starting place for the following discussion is a code fragment, such as the one in Figure 2.7. The boxed code in Figure 2.7a is copied into the application-specific compiler and thus appears as static code. The boxed code in Figure 2.7b is generated at runtime by the application-specific compiler. The goal of this section is to show how runtime code generation can be implemented efficiently; in essence, it shows how to efficiently implement "`emit`" of Figure 2.8a.

2.3.1 Tree Compiler

During static compilation, the source code is parsed into a tree-like internal form. The static compiler identifies both deferred operations and those operations that depend only on specialization variables (as described in the previous section). For example, Figure 2.9 shows the parsed source code with a dashed line around operations depending only on `x`.

The static compiler builds tree fragments for the code that depends on variables besides `x` and stores them as constant data in the application. These correspond to the boxed code fragments in Figure 2.7b. Note that some of the tree fragments may be incomplete; holes appear wherever a node has a child subtree that depends only on specialization variables. At runtime, the application-specific

[5] There are many kinds of tree compilers. The term "tree compiler" is used in this chapter to mean any compiler which performs at least instruction selection and basic optimizations. Later chapters use the term more narrowly to mean fast general-purpose code generators which produce code using using local (basic block-level) optimizations. The same term, "tree compiler", is used both places in an effort to minimize the amount of new terminology. Likewise, there are many kinds of template compilers, and a particular style is described here in order to make the presentation concrete. For both kinds of compilers, the presentation is intended to err towards being specific rather than general.

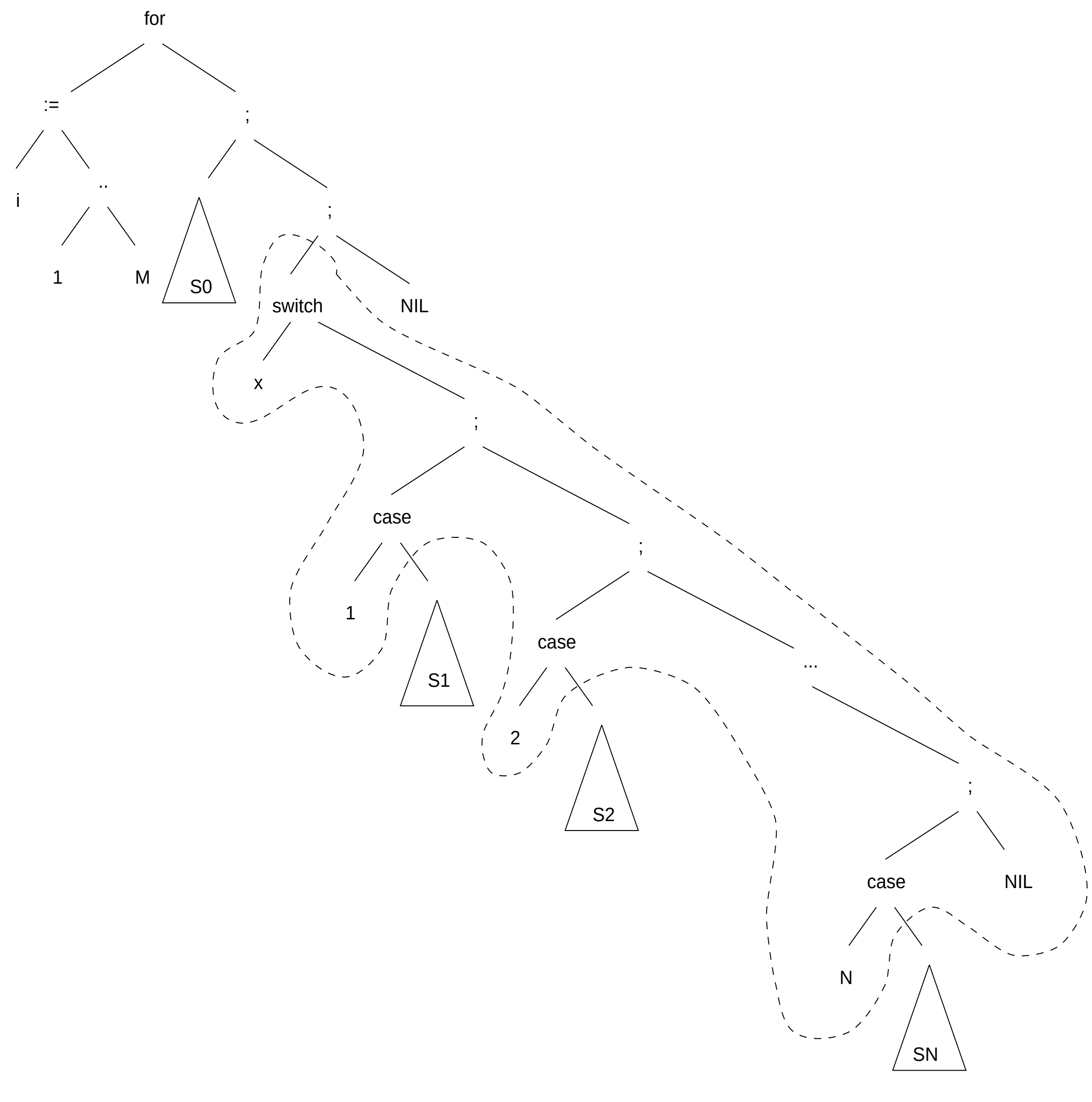


Figure 2.9: A static compiler parse tree.

The parse tree produced by the static compiler for the code of Figure 2.7, with type annotations omitted for clarity. Code inside the dashed line depends only on x.

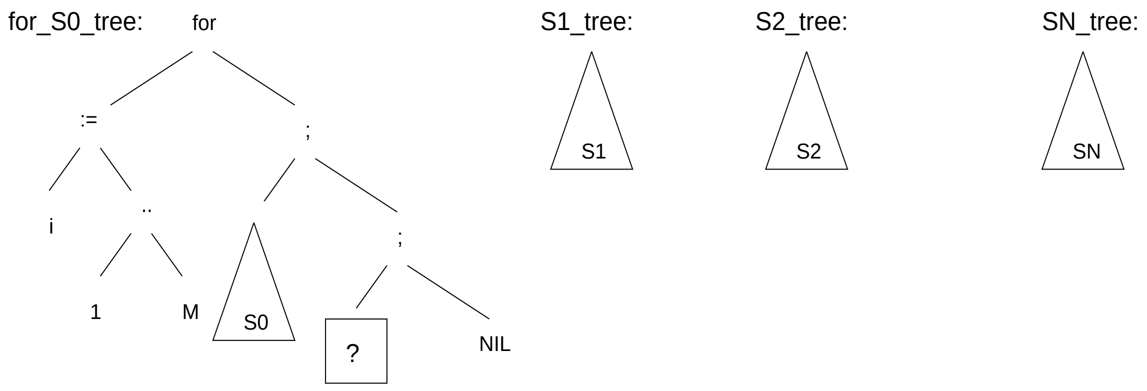


Figure 2.10: Trees for a tree compiler.

Trees built by a static compiler when it produces a tree compiler for the code of Figure 2.7; type annotations are omitted for clarity. The tree for_S0_tree has code for both the for loop and S0 because they are always treated as a single unit. The node "$\boxed{?}$" is a hole that will be filled at runtime with one of S1 through SN.

compiler evaluates the operations that depend only on specialization variables, and uses the results to build special-purpose tree fragments that replace the holes.

For example, the tree fragments extracted from the tree in Figure 2.9 are one for the for loop and S0, and one for each of S1 through SN. These fragments, shown in Figure 2.10, appear as constant data in the final application. Since the loop body may contain any of S0 through SN, the tree data structure that gets stored in the application has a hole (shown as "$\boxed{?}$") to represent the uncertainty over which subtree will finally appear here. The tree also has some symbol table data structure or access method (not shown) for locating all such holes in any given fragment.[6]

The static compiler also generates code that combines the trees in data-dependent ways. For example, the code in Figure 2.11 builds a tree using the tree fragments in Figure 2.10. When x is 2, the application-specific compiler produces a tree that has a for loop whose body is two statements, S0 and S2. Although tree fragments may have holes, the final tree has no holes. The application-specific compiler passes the final tree to a conventional code generator to produce executable code.

Trees are manipulated at runtime with just a few operations. The "tree copy" operation (denoted copy in Figure 2.11) duplicates a tree. The "fill hole" operation (denoted tree.hole[*hnum*] = ...) finds all instances of hole *hnum* and replaces them with the given tree. The "build constant" operation (not used in the example) is used where the current value of a specialization variable is a constant in dynamically-generated code. "Build constant" creates a tree fragment representing a constant; it is needed because specialization variables take on too many values to statically generate one tree fragment for each possible runtime value.

For simplicity, the final tree is built bottom-up. The bottom-up order is discovered by postorder

[6]Holes may actually be represented other ways; see §3.1.4.

```
switch (x) {
case 1: y = copy (S1_tree)
case 2: y = copy (S2_tree)
...
case N: y = copy (SN_tree)
}

z = copy (for_S0_tree)
z.hole[0] = y
codebuf = alloc (...)
codegen (codebuf, z)
```

Figure 2.11: A tree compiler.

A tree compiler for the code fragment of Figure 2.7. The tree is built bottom-up and there are no holes in the final tree passed to the code generator. The tree compiler then allocates codebuf, which is storage for the executable code produced by the code generator.

traversal of the static parse tree. Building the tree bottom-up ensures that each subtree is complete (free of holes) when it is inserted into another tree. In turn, that ensures that each invocation of the "fill hole" operation needs to update only holes within a single copy of a statically-generated tree fragment. Limiting the scope of the "fill hole" operation simplifies and thus speeds hole filling.

2.3.2 Template Compiler

A template compiler is like a tree compiler, but the general-purpose tree code generator is eliminated and the tree fragments are replaced with machine-code *templates*. The template code generator is faster, because more work is performed statically and thus it manipulates a representation much closer to the final machine code. In particular, the template compiler is only capable of generating code for the particular application; the tree compilers' general pattern matching is thus dramatically simplified. Similarly, the template compiler performs few, if any, runtime decisions for instruction selection. Finally, each IR node in a template compiler typically corresponds to several instructions in the final code, whereas each node in a tree compiler typically describes only part of a final machine instruction.

For example, the tree compiler described in the previous section manipulates machine-independent trees and then generates code by passing trees to a general-purpose code generator. As a consequence, the runtime code generator performs all operations that map machine-independent trees to machine code. In addition, the tree compiler's general-purpose instruction selector matches arithmetic operations against both integer and floating-point instructions, even though such work is always wasted for applications that VSO only integer code.

Templates are copied and concatenated, just as trees are in tree compilers. Templates are concatenated by placing them in memory one after another at their final location. Templates also have a small amount of symbolic information. First, since some templates use position-dependent code, templates also have relocation information, so that the "copy" operation can patch the templates when they are moved. Some templates are also incomplete; for example, an instruction immediate may be given a value at runtime. Here, the symbolic information describes how and where to insert values. Template code generation copies the templates directly to their final destination and performs relocation and patching in-place. Since template copies are not themselves copied again, the relocation information is not copied.

Templates also have holes, as with trees. Some holes are filled with code fragments which may be different sizes. For example, the for loop body has a hole that may be filled with any of S1 through SN, and the machine code for each of those cases may be different sizes. Thus each "parent" template with H code hole instances is split into up to $H + 1$ fragments. A split "parent" template is copied by copying the first fragment, copying the first code hole instance, copying the second fragment, the second code hole, the third fragment, and so on [Auslander et al. 96, Consel & Noel 96]. Split templates may contain branches around the variable-size code that fills the holes, so each copied fragment may need to be patched with the target fragment's address. Some backpatching may thus be needed. Data holes are immediate fields in instructions and generally require similar treatment (in the experiments presented here, data holes do not require template splitting).

In principle, template compilers are built by first building a tree compiler and then, during static compilation, running the code generator over each tree fragment in order to generate each template. In practice, more sophistication is needed, e.g. for register allocation [Leone & Lee 94].

Figure 2.12 shows a template compiler for the code fragment of Figure 2.7. Templates are copied linearly to form the final code fragment. Linear order is discovered with inorder traversal of the static parse tree. Since the for loop template contains a hole, it is split into two fragments, a loop prologue and a loop epilogue. When the template compiler copies the prologue, it saves the destination address (top) for later use, when copying the epilogue. The template compiler then patches in the body's template, which is a template for one of S1 through SN. When the template compiler copies the loop epilogue, it patches the epilogue copy with the prologue's address, so the epilogue can branch back to the top of the loop.

Figure 2.12 oversimplifies the patching, since a template with many fragments may, in general, have jumps between all of the fragments. However, the static compiler knows which fragments have branches and thus can patch in final addresses only where needed. For example, a general for loop has indefinite bounds and may thus branch forwards when the loop body executes zero times. However, the for loop in our example always executes the body, so the prologue fragment needs no forward branch, and the static compiler only creates a single call to fragment_patch, for the epilogue.

Note also that laying out code linearly can complicate the handling of control transfers, such as goto statements, which can transfer from within a hole's code into a parent fragment or into another hole.[7] Supporting gotos requires additional runtime data structures.

Figure 2.13 shows example templates for the boxed code fragments of Figure 2.7b (corresponding also to the tree fragments in Figure 2.10). The relocation information for each template is not shown

[7]However, break in a switch is not a problem, since the control transfer is associated with the switch template.

```
codebuf = alloc (...)
dst = &codebuf[0]

copy (dst, for_tplt_fragment0)
top = fragment_addr (dst, for_tplt_fragment0)
dst = dst + sizeof(for_S0_tplt_fragment0)

switch (x) {
case 1:   copy (dst, S1_tplt)
          dst = dst + sizeof (S1_tplt)
case 2:   copy (dst, S2_tplt)
          dst = dst + sizeof (S2_tplt)
  ...
case N:   copy (dst, SN_tplt)
          dst = dst + sizeof (SN_tplt)
}

copy (dst, for_S0_tplt_fragment1)
fragment_patch (dst, for_S0_tplt_fragment1, top)
dst = dst + sizeof (for_S0_tplt_fragment1)
```

Figure 2.12: A template compiler.

A template compiler for the code fragment of Figure 2.7. For any templates that contain position-dependent code, the copy operation relocates the template to its new address, using symbol table information stored with the template. The for loop template has a hole and is therefore split into two parts, a prologue, for_S0_tplt_fragment0, and an epilogue, for_S0_tplt_fragment1. The epilogue may branch to the prologue, so the template compiler remembers the address of the prologue, top, and patches the epilogue where it contains the branch back. Since the prologue's branch entry may be different than the template's starting address, fragment_addr uses the relocation information for the template to compute where the epilogue should branch.

but consists of information to update position-dependent instructions plus branch information for the prologue and epilogue fragments. The prologue uses an offset to discover the location of TOP within each copy of for_S0_tplt_fragment0, and the epilogue uses an offset and instruction type information for the location and format of the branch offset within copies of for_S0_tplt_fragment1. The instruction type information distinguishes between different instruction formats, which are patched different ways. Figure 2.14 shows how the templates are mapped directly onto the code produced by the template compiler.

2.3.3 Comparison

Compilation with a template compiler is substantially cheaper than with a tree compiler: the tree compiler's compilation cost includes general-purpose, pattern-based instruction selection, sophisticated optimization, code scheduling and register allocation, whereas template compilation is as simple as a series of simple block copy operations of aligned constant data, plus a few patching

```
for_S0_tplt_fragment0:             S1_tplt:
          mov #1 → r0                shl ...
TOP:      ld ...                     sub ...
          add ...
          st ...                   S2_tplt:
                                     xor ...
                                     and ...
for_S0_tplt_fragment1:
          add #1, r0
          bne r0, #N, [?]
                                   SN_tplt:
                                     mov ...
                                     st ...
```

Figure 2.13: Templates used for the template compiler of Figure 2.12.

Instruction mnemonics are simply examples, and ellipsis after them ("...") indicate omitted details. The template for_S0_tplt_fragment0 has symbol table information that includes the offset of TOP; likewise, the template for_S0_tplt_fragment1 has symbol table information for the location and format of the hole $\boxed{?}$.

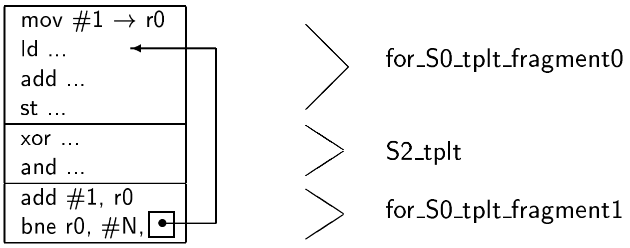


Figure 2.14: Code generated by the template compiler of Figure 2.12 when x is 2.

The boxed branch target is computed when the application-specific compiler runs because the size of the inner loop depends on the value of x and the corresponding inner loop case.

operations.[8] However, a template compiler is more complicated to produce using existing tools and it may generate inferior code.

Put another way, the template compiler is faster than a tree compiler because more machine and application mapping operations are performed during static compilation. Indeed, the template compiler is so specialized that it can produce only different versions of the application's VSO'd inner loop. In contrast, a tree compiler's back end is a general-purpose code generator that could be used for compiling any code, and might even be the same one used by the static compiler.

The same tuning and specialization that makes the template compiler very fast, however, may cause it to produce less-optimized code. Problems arise because, for example, some candidate optimizations span templates. For example, S0 may update a value that is also updated by S3. The

[8]Both the copy and the patching may be inlined; see Section 3.2.1.

template with S0 may load, modify, and store the value, only to have it immediately modified and stored again by S3. However, unless *all* templates use the value, failing to store the value in S0 may lead to erroneous results. In contrast, the tree compiler can more easily optimize across all of the code that is being generated: optimizations are performed after a whole tree is constructed, and can thus span static boundaries.

Note that a variety of techniques can improve the optimization opportunities with a template compiler. Examples include simple peephole optimizations on the runtime-generated machine code [Massalin & Pu 89, Massalin 92a, Fitzhardinge 93, Auslander et al. 96], or generating two versions of S0, one which stores the result and one which doesn't. However, these complexities make the template compiler larger, slower, and, as the number of runtime decisions increase, make the template compiler more and more like a general-purpose code generator that has been tuned to an application's most common operations [Henry 87].

Another practical consideration is that the tree compiler uses a standard, relatively conventional code generator, and so there is less development involved than in producing a template compiler [Engler & Proebsting 94, Engler et al. 95]. For example, the tree compiler may use the same code generator as the one used by the static compiler. In contrast, each template compiler back end is unique. Given current partial evaluators and code generators, generating templates automatically is substantially more difficult, though possible [Leone & Lee 94, Auslander et al. 96, Consel & Noel 96]. The tree compiler thus offers a simplified route to producing a working VSO system.

2.4 Cache Coherency

Since VSO involves changing the instruction space during program execution, a final p art of VSO transformations is ensuring instruction cache coherency. Thus, the final part of the transformation is ensuring that code space is allocated and freed properly, and that the necessary instruction cache coherency is maintained.

For example, Figure 2.15(a) shows a static code fragment and Figure 2.15(b) shows the same routine with the inner loop removed and replaced with code that implements VSO. Here, "asc (x)" is the call to the application-specific compiler, "icoherency (code)" ensures that the instruction caches are consistent with the newly-generate code, "(invoke code)" invokes the dynamically-generated code, and "free (code)" reclaims the storage. From a transformational standpoint, the important feature of p' is that the instruction space updates are performed before the first invocation of code, and that code finishes executing before the space it uses is deallocated.[9] More details of cache coherency appear in Section 3.4;

[9] Phased allocation is typical of VSO, but is not a requirement. For example, the application-specific compiler could generate code that is itself self-modifying.

```
procedure p(x, y)
{
    ...
    for i=1..M {
        ...
    }
    ...
}
```

(a)

```
procedure p'(x, y)
{
    ...
    code = asc (x)
    icoherency (code)
    (invoke code)(y)
    free (code)
    ...
}
```

(b)

Figure 2.15: A source fragment and a transformation to use VSO.

The application-specific compiler manages any instruction-space incoherency that it causes, and the code is then constant until it is no longer needed.

Chapter 3

Implementation

Neither more nor less code than is required.

–Jeff Chase, alluding to Humpty Dumpty [Carroll 60]

This chapter describes details of the VSO implementation used in this dissertation. The goal is to describe the experimental system in detail, to show how particular problems were resolved, and to also give direction for future systems. Topics of this chapter include tree and template compiler implementation details, (§3.1 and §3.2), including dynamic linking; code alignment (§3.3); and instruction cache coherency (§3.4).

One goal of this dissertation is to show where and why VSO is profitable, and to show where a sophisticated but complex implementation is needed, and where a simpler implementation is good enough. To that end, much of the work here is a feasibility study, exploring the benefits of VSO and examining the tradeoffs between implementations, in order to determine where a simple and fast runtime compiler can cheaply provide most of the benefits of a slower sophisticated compiler.

For these reasons, the experiments are run using prototype compilers where the application-specific compilers are implemented by hand or in some way that is overly-general and thus inefficient. Each application was transformed by hand-analyzing the code that would be VSO'd and then hand-building the appropriate trees or templates and the code to concatenate them (discussed below). Hand-transformation is entirely static; no human intervention is required at runtime. Hand-construction of the application-specific compilers makes it more difficult to build each prototype than if an automated system were available. However, the extra labor was still less than would be required to construct an automated system.

3.1 Tree Compiler Implementation

In the tree compiler implementation, each application is transformed to produce tree fragments and an application-specific compiler like those of the previous chapter (see Figures 2.10 and 2.11). The application is also modified to call the application-specific compiler, invoke the dynamically-generated code, and then clean up (see §2.4). At runtime, the program is called normally, and code

```
type_t *type_prim (enum {UCHAR, INT, FLOAT, ...} prim)
type_t *type_pointer (type_t *base_type)
type_t *type_list (type_t *type, type_t *arg_list)
type_t *type_function (type_t *return_type, type_t *arg_list)
type_t *type_copy (type_t *base_type)
```

Figure 3.1: Runtime compiler routines that build type trees.

The routine type_prim creates a tree representing a basic type; pointer creates a pointer to a basic type; list creates lists of types used in argument lists, etc.; function creates a function type; and copy duplicates a type tree.

```
int_type = type_prim (INT)
void_type = type_prim (VOID)
void_ptr_type = type_pointer (void_type)
args = type_list (type_copy (void_ptr_type), type_copy (void_ptr_type))
func = type_function (int_type, args)
ptr_to_func = type_pointer (func)
```

Figure 3.2: An example of creating a type tree.

Code that creates a type signature for a pointer to a routine that is passed two "void *" pointers and returns an integer.

is generated and invoked without further intervention. This section provides more detail about the data structures, transformations, and allocation policies used by the prototype.

3.1.1 Types

The tree compiler makes limited use of types. Since the runtime library is written in C and C types are not first-class objects [Kernighan & Ritchie 88, Atkinson & Morrison 85], the types are implemented manually, and are represented as trees. Because this VSO system uses a statically-typed language,[1] operation types are known statically. Thus, types are used to represent information — for example, the sizes of machine-dependent objects such as characters and integers — but are not used to choose dynamically between, say, typed operations such as integer multiply and floating-point multiply. Figures 3.1 and 3.2 show routines for manipulating types and an example of such a manipulation.

In principle, type signatures can also be used to debug the tree IR. For example, both subtrees of an integer addition must have a recursive type signature that evaluates to an integer. However, no such checking is implemented in the prototype, nor would it be needed in a production system, where all tree IR would be manipulated automatically and in a type-safe manner. In contrast, an IR

[1]This dissertation focuses on programs written using statically-typed languages (§A). VSO can also be used with dynamically-typed source languages. However, since type dispatch is usually more expensive than value dispatch, VSO is typically profitable only when dynamic type dispatch overheads have already been removed.

```
{
    enum {IADD, IMUL, FADD, ...} op_type;
    UNION {
        normal     { tree_t *left_child, *right_child }
        variable   { sym_table_entry_t *sym }
        iconst     { int iconst_val }
        fconst     { float fconst_val }
        hole       { int holenum; tree_t *filled }
        root       { relocation_t *r, tree_t *subtree }
    }
} tree_t;
```

Figure 3.3: A prototypical tree node.

The operation type, op_type describes both the kind of node (normal, variable, constants, hole, root) and, for normal nodes, the type of operation that the node represents. Any given node is of only one type, so node storage for each type is overlaid.

that is designed for direct programmer manipulation [Engler & Proebsting 94] should incorporate type checking.

3.1.2 Trees

The tree compiler uses a simple tree structure patterned after the tree structure in a tree-based code generator [Henry 87, Fraser & Henry 91]. Most nodes contain a node operation type and pointers to left and right children. Node operation types represent conventional arithmetic (IADD, FADD, etc.), control and looping structures (if, else, for, etc.), and so on. References to variables use a node that points to a symbol table entry. Two special node types, hole and root, are used to implement holes; they are discussed below (§3.1.4). An example node structure declaration is shown in Figure 3.3.

Trees are built bottom-up from individual nodes; individual nodes are built using library routines. Each routine allocates new nodes and sets the children to point to subtrees passed as parameters to the library routines; the return from each routine is the root of the new tree. For example, Figure 3.4 shows routines that build some common tree nodes, and Figure 3.5 shows calls used to build $y = y*\boxed{?} + 2$.

The tree copying costs are minimized by keeping each tree node small. Each tree node is implemented as a union type, distinguished by the node operation type (IADD, FADD, etc.). Unions are notorious for introducing debugging problems where fields are accidently written as one type and read as another, and C is notorious for memory allocation errors anyway [Cannon et al. 90]. To help with debugging, each tree node can be represented either as a union or as a large structure with one field for each "union" element. The switchable nature of the nodes was implemented by using C's text replacement (preprocessor) phase to define the pseudo-keyword UNION as either union or struct. The "debugging" representation turned out to be quite useful, as did simply having two implementations and insisting that they have the same behavior, except for execution time. In a "real" VSO system, trees would be manipulated automatically (see Section 2.2), so using just a

```
tree_t *const_int (int value)
tree_t *variable (sym_table_entry_t *sym)
tree_t *binary_op (enum ... op_type, tree_t *left_child, tree_t *right_child)
tree_t *assign (tree_t *lval, tree_t *rval)
tree_t *hole (int holenum)
tree_t *root (tree_t *subtree, int number_of_holes)
```

Figure 3.4: Some library routines that create tree IR.

All routines allocate new nodes. Most nodes also take subtrees and arguments and make those subtrees children of the newly-allocated nodes.

```
/* Build a tree for 'y = y* hole + 2'. */
y_tree = variable (sym_table_entry ("y", type_int))
hole_tree = hole (0)
times_tree = binary_op (IMUL, y_tree, hole_tree)
two_tree = const_int (2)
plus_tree = binary_op (IADD, times_tree, two_tree)
assign_tree = assign (y_tree, plus_tree)
subtree = root (assign_tree, 1)
```

Figure 3.5: Example code to build a tree fragment.

Hand-generated code that builds a tree fragment for a source expression corresponding to y = y* ? + 2. The call "hole (0)" generates an instance of a hole numbered 0. There is a hole in the resulting tree fragment, so root (assign_tree, 1) is used to build an auxiliary data structure that speeds up runtime hole filling.

compact represention would suffice.

3.1.3 Static Tree Generation

In a production system, tree fragments would be generated statically and embedded in the binary as read-only (constant data), to be copied and manipulated at runtime. In the prototype, the hand-generated tree fragments are built using code that simply generates the trees and then stores them in global program variables. To simplify the prototype implementation, the trees are generated at runtime as a part of program startup. They are nonetheless treated as static constant data: all of the trees generated during startup are treated as constants, and all subsequent tree manipulations, which are "real" runtime operations, are performed by first copying the "statically-generated" trees and then operating on the copies. The elapsed time measurements do not include startup.[2]

The application-specific compiler builds tree fragments by simply copying and combining statically-generated tree fragments. As described in Section 2.3.1, however, some constant leaves are created

[2]Both the static-code and VSO'd implementations execute the startup code in order to ensure that the data layout and cache contents are the same; however, in both versions, the timers are off during startup.

at runtime. Dynamic creation of constants is needed where a runtime variable is used as a literal constant in the dynamically-generated code. For example, where the source code fragment "x=y+z" is dynamically replaced with "x=y+2", the leaf node for 2 is generated at runtime, since it is impossible to preallocate trees for all possible values of z. Since the types of the runtime constants are known statically,[3] they can be generated quickly using an allocator that is statically specialized to the needed type. Even the dynamically-allocated constants are typed statically, so runtime allocation is essentially trivial.

Note that expressions that are evaluated at runtime by the application-specific compiler do not need such leaf allocation. For example, if switch (x) is evaluated for x=2, as was shown in Figure 2.8, neither x nor 2 appears in the final (dynamically-compiled) code, so no literal is needed. Since all dynamic compilation takes place in a statically-typed context, no dynamic type creation is required.

3.1.4 Tree Holes

The hole and root node types do not appear in a traditional compiler IR. The hole nodes represent holes that the application-specific compiler fills at runtime with other tree fragments. Each root node is the base of a subtree and points to both a subtree and a list of lists used to quickly find the holes in the subtree.

Conceptually, each tree consists of ordinary nodes, plus some number of holes. Some holes appear more than one place in a tree. For example, code that increments two values by the same amount might be represented by "i += $\boxed{?_x}$, j += $\boxed{?_x}$". Although there are two hole *instances*, there is only one *logical hole*.[4] That is, a hole-filling operation, "tree.hole[x] = ..." should fill both holes $\boxed{?_x}$ with the tree that appears on the right-hand side.

The current implementation provides two tree manipulation facilities that turned out to be unneeded and which slow down tree manipulation. First, the IR operations allow the application-specific compiler to fill holes with subtrees that themselves have unfilled holes, so that updating the parent's holes also updates the child's holes. The current implementation thus provides an encapsulation mechanism where trees are built by a family of application-specific compilers or in multiple passes (§10.5.17). The IR operations also allows subtrees to be deleted and replaced, which would be used by a system that incrementally updates the IR and then recompiles (§10.5.16).

Unfortunately, providing the generality to support these operations adds substantial overhead, because the implementation supports far more dynamic behavior than was necessary. For example, the VSO experiments use tree fragments with a constant number of holes; holes in a tree are filled before the tree is inserted into another tree, and are never modified after insertion; the number of hole instances is already available at hole fill time; and trees are always deleted as a whole, instead of being deleted incrementally. In contrast, the current implementation supports trees with varying numbers of hole nodes, and a given tree may be copied and modified using insertion, deletion and hole remapping, copied again, modified again, and so on. Thus, while VSO could use a constant mapping structure that is never copied at all, the current implementation uses a mutable mapping

[3]If the types were statically unknown, optimizing around the value would also perform type-specific optimization.
[4]Punsters should contemplate this to see how many logical holes they can find.

structure that must be copied, merged and updated along with its the associated tree. Furthermore, since holes may be deleted, storage allocation is more complicated. For example, filling N hole instances with one subtree makes it difficult to tell, when a hole is deleted, whether the subtree should also be deleted. The current system fills N hole instances using N copies of the tree, thus increasing the copying overhead.

The practical consequence is that trees are not implemented as efficiently they should be, and manipulations to copy trees and fill holes are therefore substantially slower than they need to be. As a result, dynamic compilation in the prototype is slower than necessary. The added expense reduces the observed profitability of VSO, and discussed in more detail in Chapter 5

The implementation could be simplified, while still supporting VSO. Each tree built at static compile time could be linearized to simplify and speed up both storage allocation and copying; an auxiliary read-only mapping structure would give the size of the tree and the location of all absolute pointers (pointers that are not represented as offsets). Copying a tree would then be a simple block copy, plus a pass to update any absolute internal pointers by adding the distance the tree has moved. A second auxiliary read-only data structure would have a list of each logical hole instance. Filling a logical hole would then simply require iteration through an array, with patching of the indicated locations.

3.1.5 Memory Allocation

Memory allocation in the application-specific dynamic compiler uses *obstacks* similar to the GNU CC obstacks [Stallman 93, Fre 95] and *arenas* [Hanson 90], but with an interface more like that of the traditional C library `malloc` and `free` [Kernighan & Ritchie 88]. The goal in all cases is to make memory allocation and freeing fast and simple by taking advantage of bounded allocation sizes and/or bounded lifetimes and gang freeing, where all of a group of tree nodes are freed together. For example, the tree built by the application-specific compiler is used to perform code generation and then the tree is freed. Here, node allocation may be as simple as decrementing a pointer and returning its value; similarly, instead of freeing node-by-node, freeing simply resets the pointer to the previously saved value.

Memory allocation and copying costs are also reduced by storing trees so that a tree copy may be implemented as a simple block copy. Block allocation and copying reduces the number of calls to the memory allocator and allows the copy to be implemented as an unrolled loop, with almost no node type examination or dispatching overhead. As mentioned above, however, the cost reduction is not as great as it might be, because the current implementation's support for holes is more complex than needed and because the ability to delete and free parts of a tree interferes with linearized copying.

3.1.6 Retargetable Code Generation

The experiments in this dissertation need a code generator with various features, including: fast code generation, since slow code generation would consume any potential speedups; direct object code generation, since generating assembly code quickly but then calling an assembler would make code generation too slow; retargetable code generation, so VSO could be studied on a variety of platforms; and a good selection of optimizers, to study the effects of runtime optimization on the dynamically-compiled code. Since no such code generator was available, the prototype performs code generation by translating the tree form to C, compiling the resulting code with the vendor-supplied C compiler or GNU CC, and then adjusting the runtime compilers running time to reflect the estimated cost of a fast, locally-optimal code generator (see §A for a more complete discussion of locally-optimal code generation).

C compilers don't usually generate code using fast locally-optimal code generators. In particular, C compilers typically either use a "dumb" strategy that produces code far worse than code from a locally-optimal generator, or else perform relatively expensive intraprocedural optimizations. To ensure that the measured execution times are representative of code from fast locally-optimal code generators, the C compiler's machine code output was checked by hand to ensure that the code used only local transformations plus "dumb" register allocation.[5]

C compilers clearly don't meet the goal of fast code generation; however, we can estimate performance. Measured performance was sacrificed because other features, such as retargetability, are harder to approximate. The compiler cost is estimated at 1,000 instructions per generated instruction at an average speed of 2 machine cycles per instruction. The estimate is conservative both with respect to the fast, locally-optimal code generation literature that reports a few hundred instructions per generated assembly-language instruction [Fraser & Henry 91, Fraser et al. 92] and also with code generators that produce machine code directly, measured at under 500 instructions per generated instruction [Engler & Proebsting 94].

In practice, many fast code generation schemes could have been used. Locally-optimal code generation was used as the standard of of comparison because locally-optimal code generators are known to produce moderately good code fairly quickly [Fraser & Henry 91, Fraser et al. 92, Engler & Proebsting 94] and because the code generator relies purely on local information for producing code, making hand comparisons easier to perform.

3.1.7 Dynamic Linking

The tree-based compilers generate C code, which is compiled to object modules that are then relocated and linked with the running programs. The dynamic linker allocates storage for the final executable code, and is also responsible for instruction cache updating (§2.4, §3.4). Note that in-core code generators can produce code that is ready to run at its final memory location, without relocation or sophisticated linking. An in-core compiler could thus use a faster and simpler dynamic linker.

The experiments in this dissertation used the GNU dynamic linker, dld [Ho & Olsson 91] on platforms where it was available. dld can load, use, and then unload code, and the source was

[5]The "dumb" register allocation strategy assigns parameters to the registers that they are passed in and assigns local variables and compiler temporaries to caller-save registers if they are available, else assigns them to stack slots and uses local (basic block) allocation. The success of this strategy depends on having few live values: as long as there are an abundance of registers, dumb register allocation does as good a job as a sophisticated algorithm.

available, making it possible to ensure that instrumentation was sensible.[6]

For platforms where dld does not work, the needed functionality was implemented using the system loader, ld, the fly instruction cache coherency interface (Chapter 11), and the smallest amount of platform-dependent code that got the test applications running. The vendor-supplied linkers are typically unsuitable because they are designed for dynamic linking of shared libraries rather than for incremental linking and unlinking. Unlinking [Segal & Frieder 89a, Ho & Olsson 91, Segal 92, Keppel 93a] is needed, because one run of an application may compile, use, and discard many code fragments.

The measured linking times are reported using the times on a platform with a full dld implementation. Although the hand-crafted dynamic linkers are less general, they are actually slower than dld, because they are implemented using ld and multiple files, and thus need to start multiple processes, manage I/O, and so on.

Linking times could be much faster if dld supported in-memory linking. Since dld was designed to link from files, there is additional overhead for system calls, conversion between disk and in-memory representations, etc. For example, dld spent typically 60% of its total execution time in the open and read system calls when linking SPARC code used for tree VSO, and spent further time transforming compact disk representations of linkage information. Thus, a linker designed for in-core linking could be an order of magnitude faster.

3.2 Template Compiler Implementation

The template compilers are built by hand, but, again, follow the general transformation procedure that would be used in an automated system (§2.3.2).

3.2.1 Templates and Patching

Each template compiler consists of assembly-level code templates plus static code that concatenates them. Essentially all assemblers lack a notation for holes [Massalin 91]. Therefore, the symbol table information for holes was also created by hand. To simplify the implementation, the symbol table was implemented as code instead of a data structure. It was built by duplicating the table walker and manually partially evaluating the table walking code with respect to each table entry. Relocation information is thus simply another machine code fragment. For example, Figure 3.6 shows a template with a hole and the symbol table entry that updates the hole. (See Section 3.2.2 and Figure 6.1 for a discussion of the runtime sizes of holes.)

The template compiler implementation uses a simple word copy routine to read each template from memory and write it to the new instruction memory. Since the template sizes and values are known statically, a faster implementation could unroll each template copy operation and then inline the template as constant data in the unrolled loop [Leone & Lee 94]. Most symbol table operations

[6] Link times are below the timer resolution on some systems, and so linking was repeated. By examining the source code it was possible to ensure that full linking was being repeated, rather than one linking step followed by a series of fast lookups.

```
bork_tplt:                                ! template start
        ld    (r0) → r1                   ! a relocatable instruction
H0:     shl   r1,0x0 → r2                 ! [0x0] used as a hole placeholder
        st    r2, (r1)                    ! a relocatable instruction
bork_tplt_end:                            ! end of the template

bork_sym:                                 ! Symbol table information, implemented as code
        ld    [offsetof(H0,bork_tplt)](r1) → r3
        or    r2,r3 → r4                  ! patch second instruction
        st    r4, [offsetof(H0,bork_tplt)](r1)
        ret
```

Figure 3.6: Example of a code template.

Simplified code for a template bork_tplt that uses the instruction immediate 0x0 as a hole. Code to copy the template is not shown. The symbol table entry, bork_sym, is implemented as a code fragment; r1 points to a copy of the template and r2 holds the value to patch into the copy's shl hole. Although r1 and r2 are used to manipulate the srl fragment, which also uses r1 and r2, no conflicts arise, since the symbol table code executes at dynamic compile time and thus completes before the execution of the copy of bork_tplt.

```
bork_tplt:                                ! The template is inlined below

bork_sym:                                 ! create and patch a 'bork' template
  set   %instr("ld (r0) → r1"), r3
  st    r3 → (r1+0)                       ! write first instruction

  set   %instr("shl r1,0x0 → r2"), r3
  or    r2,r3 → r4                        ! patch second instruction
  st    r4 → [offsetof(H0,bork_tplt)](r1)

  set   %instr("st r2 → (r1)"), r4
  st    r4 → (r1+8)                       ! write third instruction
  ret
```

Figure 3.7: An optimized version of Figure 3.6.

Instead of copying the template and then manipulating the copy, the body of the template, bork_tplt has been inlined into the symbol table routine so that each word of memory is written only once. Thus, the symbol table entry bork_sym copies the template to the address in r1, simultaneously patching the hole with the value in r2.

can also be inlined in the copy. An optimized version of Figure 3.6 is shown in Figure 3.7. Although such optimizations would be straightforward to automate, coding speed, debuggability and ease of modification were more important for the prototype, so the slower but simpler construction was used.

3.2.2 Memory Allocation

A template compiler runs by allocating storage for the final code, generating code, and performing instruction cache coherency.[7] The size of the template-generated code is simply the sum of the sizes of the individual templates, and can be computed at runtime by executing code identical to the application-specific compiler, but with each instance of "copy (*tplt*)" (see §2.3.2) replaced with "size = size + sizeof(*tplt*). In principle, there are places where factors such as the branch distance affect the final size. In these cases the runtime compiler can either perform detailed computations to pick the size exactly, or, for each template, it can conservatively pick the size of the largest alternative.

The debugger application (§4.1.5) uses a different strategy for allocating space. The application-specific compiler guesses the total space needed, and each copy operation checks the available space. If a copy operation would overflow the available space, the application-specific compiler aborts, allocates a larger region and restarts the entire compilation. This odd guessing arrangement was used to simplify the prototype, which uses separate code generation modules for breakpoint evaluation and displaced instruction simulation (§10.3.4, §10.5.17) and which was put into GDB in the easiest way rather than being integrated in the cleanest way.

3.3 Code Alignment

Minor variations in code layout can dramatically affect a program's execution time [Gupta & Chi 90, McFarling 91, Torrellas et al. 92, Lucco 93, Smotherman 94, Mosberger et al. 95, Mosberger et al. 96]. For example, adding nop instructions pseudo-randomly to an application may increase the dynamic instruction counts by tens of percent and yet introduce alignment effects that reduce overall execution time by tens of percent [Lucco 93]. However, dynamically-compiled code does not generally reside in the cache at the same location as its static counterpart. These code location differences could either hurt VSO's measured performance, or they could alleviate static code instruction cache conflicts and thus lead to "false" performance benefits that could have been achieved simply by reorganizing the static code. That is, it would be unclear whether performance changes were due to VSO or its effect on code placement.

To show that speedups were caused by VSO and not by code placement, the prototype loads all dynamically-compiled code at a memory location that maps to the same locations in the cache hierarchy as the static code (see §3.3). Cache alignment was implemented by building a special-purpose aligning memory allocator that is used only for allocating code space. The allocator uses both the size of the allocation and the location of the static code it replaces. The aligning memory allocator is implemented by internally allocating a very large memory region and returning an address in that region that is the same as the static code, modulo the size of the largest cache in the hierarchy.

[7] Generating new code does not always require instruction cache coherency, but it is required in general. For example, if the allocated region had been used previously then stale instructions may still be present in the instruction cache. Even without reuse, it may be necessary to perform coherency actions in order to ensure that the new code is flushed out of the first-level data cache.

Dynamically-generated code may be larger or smaller than its statically-generated counterpart, which may have a variety of effects. Where the static code has conflicts and the dynamically-generated code is smaller, the VSO'd code may have fewer cache conflicts. These improvements are attributed to VSO. Where the dynamically-generated code is slightly larger, VSO'd code could interfere with static code adjacent to the code that the VSO'd code replaces, simply because of the VSO'd code's slightly larger size. In these cases, the static code is padded so that the VSO'd code will overlap only the static code that it replaces. Any losses due to conflicts with non-adjacent code are attributed to VSO. Where the VSO'd code is much larger than the static code, it may introduce more cache conflicts. These losses are attributed to VSO.

The virtual-to-physical page mapping affects hit rates (conflicts) in physically-addressed caches. The virtual-to-physical page mapping varies from run to run and cannot be controlled by virtual address placement, nor do the systems provide hooks to control or observe the physical page mapping. Mapping variations cause substantial variations in the miss rates and thus the timing from run to run [Uhlig et al. 94]. The reported execution times are therefore averages over several runs.

VSO'd programs allocate more memory than their static code counterparts. Although much of the data is freed before the main (inner) code of the application is invoked, simply allocating and freeing a set of blocks may affect what memory is returned by later calls to the allocator. The prototype avoids cache-related performance effects by having the static and dynamic versions allocate and free all the same data structures. The time spent in allocation and freeing is not counted in the times reported for static-code versions.

3.4 Instruction Cache Coherency

Since VSO allocates and deletes instructions, it could cause the instruction caches to become incoherent. This section describes how instruction cache coherency problems were avoided.

The tree compiler uses dld to implement dynamic linking (§3.1.7). The dld distribution does not ensure instruction cache consistency, so it was modified to use the fly instruction-space coherency package. In the template compiler, the caches are updated explicitly by having the template compiler call fly before returning the newly-created code.

The fly interface is described generally in Chapter 11. This section describes details of the implementation for each platform in the study.

AXP Instruction-space modifications on the DEC AXP (Alpha) make use of the operating system-dependent IMB privileged architecture library call [Sites 92]. Under our version of OSF/1, however, subsequent writes to pages cause an operating system signal to the application. It appears, by experimentation, that the page is being marked unwritable. Instruction caching and flushing appears to work properly if pages are IMB'd and then reprotected as writable using the mprotect operating system call.

MIPS MIPS systems use kernel code to flush the caches. The code is accessed under DEC Ultrix using the cachectl system call. The cost of flushing small regions (tens of bytes) is dominated by the cost of the system call.

SPARC SPARC systems use the flush instruction to signal changes to the instruction space [Sun 91]. The system used in these tests has a unified cache, so all code writes are available immediately, without using flush. Therefore, performance is measured with the fly library configured to do nothing. The library can also be configured to use flush instructions, but some SPARC implementations trap on every flush, even if the system uses unified caches [Sun 91].

3.5 Code Invocation

VSO'd code can, in principle, be substituted for any statically-generated code. An implementation must resolve how the code is invoked (via some kind of an indirect branch), how control returns to the static code, and how storage allocation for the static code interacts with storage allocation for the VSO'd code.

Since the prototype uses the C compiler to generate code, excising parts of routines was not an option. Instead, the prototype dynamically compiles both the key inner code and any other "unaffected" code that appears in the same routine, even if the unaffected code is not optimized by VSO. Using procedures to define replacement boundaries also isolates storage allocation issues, since procedure boundaries have a well-defined set of free (caller-save) registers.

Having the dynamic compiler generate code that could have been generated statically can reduce VSO's overall performance improvement, since the dynamic compiler compiles more code, may optimize it less than a static compiler and the code may not benefit from value-specific optimization. In particular, the dynamically-generated "static" code cannot be inlined statically in its caller. In practice, no degradation was observed, because most routines with VSO'd code spent most of their time in the VSO'd parts of the routines and since the remaining parts of the routines did not benefit from sophisticated optimizations. In addition, the static versions of VSO'd routines did not benefit from inlining, either because they were already invoked via function pointers, because time spent in the routine overwhelmed time spent in the call overhead, or because inlining did not expose additional optimizations.

Chapter 4

Methodology

The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence.

–Fred Brooks, Jr. [Brooks 87]

With what measure ye mete, it shall be measured to you.

–Saint Mark

hard′ware *('härd′wâr'),* n. *The part of the computer you can kick.*

– Anonymous

This chapter describes the methodology used for the measurements in the following chapters: the workload that was used for investigating VSO (§4.1); how the workload was measured (§4.2); and the platforms (§4.3);

4.1 Applications

This section describes the workload used in the dissertation's performance studies, including how the applications were selected, what they do, what parts of them were optimized using VSO and why those parts aren't amenable to static optimization. It also reports what input(s) were used for the reported measurements.

The workload was chosen to satisfy several criteria.

1. **General Interest** The applications were selected from areas with broad use to show that VSO is widely applicable. Choosing from widely-used areas also increases the likelihood that there is already good information about how to make fast static code implementations, and thus helps to ensure that the static code is good. By using high-quality static code for these studies, the results show that VSO is optimizing away overheads that are intrinsic to static compilation.

2. **Static Optimization** Each of the applications suffered unacceptable code growth when extensively specialized, and each application uses enough cases that common-casing didn't give acceptable coverage.

3. **Ease of Implementation** Each application has four implementations, so it was important to use applications that could be built in a modest amount of time, especially because only one of the four implementations is written entirely in a high-level language. Two of the remaining implementations were written using assembly code and the last with manually-constructed trees.

4. **Applicability** Like other optimization techniques, VSO is only applicable to some code sequences. Thus, the benchmarks were selected because they were amenable to VSO's transformations. Such selective analysis is analogous to studying the effects of thread scheduling or dynamic memory allocation policies only for programs where changing the policies is expected to make a difference. It would also be interesting to study how often VSO can be applied, but this dissertation only takes the first step of studying the effects of various VSO policies.

Potential applications are those with one or more variables that are used repeatedly in a critical inner loop and whose values change only occasionally at runtime. Candidates were chosen by estimating features of the inner loop's behavior that determined whether VSO would be applicable: whether optimizations would be produced using VSO, how much dynamic compilation would cost, and how often the key variables would change and thus require recompilation. The code generation cost was usually estimated using the size of the statically-generated code, although VSO'd code can be larger or smaller than its static counterpart.

4.1.1 Bitblt

The bitblt (bit block transfer) algorithm [Ingalls 81, Goldberg & Robson 83] is typically used as a graphics kernel operation, and is an example where VSO has been used successfully in the past [Pike et al. 85, Locanthi 87, Miranda 87, Foley et al. 90, Kaepplein 95]. The bitblt function has several parameters (source and destination alignment; transfer operation) that alter control flow in the innermost loop of a loop nest. These parameters are constant across any given call; therefore, control flow is the same for all iterations in any given invocation. Figure 4.1 shows the core of a "naive" bitblt implementation without any specialization.

Special-casing for transfer operation, source and destination alignment, overlap, and for single-word transfers generates about 2,000 special cases: there are 16 cases for the transfer operation, COPY, AND, OR, XOR, etc.; 2 cases each for whether left and right source and destination are word-aligned ("yes" and "no"; 16 cases total); 2 cases each for whether the source and destination are aligned with each other or overlap in a way that requires right-to-left copying (4 cases total); and 2 cases for whether or not the region fits in a single source and destination word, so that the inner loop is executed only once per line ("character blit" [Locanthi 87]).

Further casing for various reasonable inner loop unrollings would increase the total to about 8,000 special cases. The static code used for these experiments uses 512 special cases based on transfer operation, source left and right edges word-aligned, source and destination aligned with each other, overlap requiring right-to-left copying, and single-iteration inner loops. An example static case is

```
for y = 0..dy-1 {                                       /* For each horizontal line. */
    mask = startmask                                    /* Starting edge. */
    right = left = *(src - step)                        /* Read word at end of line. */

    for x = nw+1..1 {                                   /* Inner loop */
        if (step > 0) {                                 /* If left-to-right copying... */
          left = right, right = *src                    /* ... right → left, new right. */
        } else {                                        /* else right-to-left copying */
          right = left, left = *src                     /* ... left → right, new left. */

        }

        if (lshift == 0) {                              /* Align 'src' to 'dst'. */
          m = left                                      /* ... unless already aligned. */
        } else {
          m = (left ≪ lshift) | (right ≫ rshift)
        }

        switch (op) {                                   /* Do the transfer. */
        case SRC:          *dst = *dst ^ ((m ^ *dst) & mask)
        case ~SRC:         *dst = *dst ^ ((~m ^ *dst) & mask)
        case SRC|DST:      *dst = *dst | (m & mask)
          ... 12 other cases ...
        case ~SRC&DST:     *dst = *dst ^ ((m & *dst) & mask)
        }

        dst = dst + step                                /* Advance to next word. */
        src = src + step

        if (x==2) { mask = endmask } else { mask = ALL_ONES }
    }
    src = src + sw
    dst = dst + dw
}
```

Figure 4.1: An unspecialized bitblt.

An unspecialized bitblt that copies the bitmap src to dst, one word per iteration of the "for x" loop. The ends of a horizontal line might occupy only part of a word, so partial words are copied using the masks startmask and endmask. Lines that fit in a single destination word are handled by setting startmask to the combination of the two masks. Inner iterations use a mask that is all ones (ALL_ONES). Source and destination bitmaps may be aligned on different memory boundaries, so each destination word is generally composed of parts of two source words, left and right. A destination word uses low-order bits from the left word, and high-order bits from the right word (canonicalized around the byte order). To avoid redundant memory reads, only one source word is read per iteration. The right word provides some bits to the destination on one iteration, and then is copied using a register-to-register move to left, in order to provide the remaining bits on the next iteration (left and right are reversed for right-to-left copying). In this and following figures, the notation *ptr means "dereference pointer"; ~, & and ^ mean bitwise negation, AND and exclusive-or, respectively; and ≪ and ≫ mean the left operand is left- or right-shifted by the right operand's value.

```
for y = 0..dy-1 {
      /* First inner iteration. */
      left = *(src-1)
      right = *src
      m = (left << lshift) | (right >> rshift)
      *dst = *dst ^ ((m ^ *dst) & startmask)

      /* Inner iterations. */
      for x = nw..2 {
        right = left
        left = *(src+1)
        src = src + 1
        m = (left << lshift) | (right >> rshift)
        *(dst+1) = *(dst+1) ^ (m ^ *(dst+1))
        dst = dst + 1
      }

      /* Last iteration. */
      right = left
      left = *(src+1)
      m = (left << lshift) | (right >> rshift)
      *(dst+1) = *(dst+1) ^ ((m ^ *(dst+1)) & endmask)

      src = src + sw'
      dst = dst + dw'
}
```

Figure 4.2: One case of a statically-specialized bitblt.

A statically-specialized bitblt that is optimized for the case where the transfer operation is SRC (source overwrites destination); the inner loop takes at least two iterations (the first, last and middle iterations can thus be specialized); the left source and destination edges and right source edge are unaligned both with respect to each other and to memory; and the copy is right-to-left. Memory reference operations such as *(src+1) are assumed to execute with a single machine instruction, thus saving some inner-loop updates of src and dst.

shown in Figure 4.2. Such static replication and specialization causes bitblt to grow to over 25 kilobytes of machine code.[1]

In contrast, the VSO version of bitblt compiles a new version of the function each time it is called. VSO generates an inner loop with all the optimizations of the static code and with complete loop unrolling, except that partial unrolling is used with particularly wide regions. Complete unrolling is probably profitable only up to about 4 iterations for complex inner loops and 8 iterations for simple loops, so the code quality is roughly comparable to the 8,000-case "reasonable unrollings" static code. Figure 4.3 shows a VSO'd bitblt. Compared to the static code, the VSO'd version has the inner loop fully unrolled, eliminating loop overhead, some pointer updates, and iteration-dependent data copying.

[1] Some versions of bitblt have even more functionality: they can merge regions with different bit depths, perform color operations, and so on [Massalin 92b, May 92]. Since case-by-case specialization leads to combinatorial space explosion, less static optimization is possible with these fancier versions of bitblt.

```
for y = 0..dy-1 {
    /* First inner iteration. */
    left = *(src-1)
    right = *src
    m = (left ≪ lshift) | (right ≫ rshift)
    *dst = *dst ^ ((m ^ *dst) & startmask)

    /* Second iteration. */
    left = *(src+1)
    m = (right ≪ lshift) | (left ≫ rshift)      |||
    *(dst+1) = *(dst+1) ^ (m ^ *(dst+1))

    /* Third iteration. */
    right = *(src+2)
    m = (left ≪ lshift) | (right ≫ rshift)
    *(dst+2) = *(dst+2) ^ (m ^ *(dst+2))

    src = src + sw"
    dst = dst + dw"
}
```

Figure 4.3: A VSO'd bitblt.

A bitblt implementation specialized to a variety of parameters: The transfer operation is SRC; the three-word copy is completely unrolled; the left source and destination edges and right source edge are unaligned both with respect to each other and to memory, but the right destination edge is aligned with respect to memory; and the copy is right-to-left. Note that the variables "left" and "right" exchange roles in the second iteration, rather than assigning the value of right to left and then updating right. All memory references of the form "*(src+3)" are assumed to execute with a single machine instruction; src and dst are thus updated only once per outer loop iteration.

Note that the static code inner loop contains a pair of assignments of the "left=right; right=*src". In contrast, the VSO'd code achieves the same effect by simply alternating the roles of left and right. On the second iteration, instead of copying the value of right to left, it simply uses the variable right instead of left. It also assigns to left the value it would have assigned to right, so that on the third iteration there is no need to copy the value from right to left. For static code, achieving the same optimization requires at least three unrolling cases (single-iteration, even loop count, odd loop count) instead of two (single-iteration or not) and thus half again as much code space.

The input data for benchmarking runs were bitmaps. The bitmap values themselves are unimportant; the factors that affect performance are the transfer operation and the size, alignment, etc., of the source and destination regions. The source region x-axes were 192 pixels (bits), corresponding to a region 15-20% the width of a typical screen and using 6-7 total inner loop iterations. The y-axes were 10 pixels (scan lines) tall. Larger bitmaps would emphasize VSO'd bitblt's asymptotically better performance instead of VSO's added compilation cost. However, since bitblt is typically used on small regions, small regions were used for benchmarking. Since aligned and unaligned bitblt calls are both common, reported performance is averaged over two cases: where the source and destination are both word-aligned (and thus relatively aligned) and where the source and destination are neither word-aligned nor relatively aligned. As in previous studies, single-iteration "character blit" [Locanthi 87] cases were omitted.

No assembly-code implementation was used for bitblt, as competing code would have required over 10,000 lines of assembly, which would be impractical for a feasibility study.

4.1.2 Cache Simulation

A uniprocessor cache simulator reads a list of memory addresses and performs a table (cache) lookup to simulate the behavior of a particular cache configuration. The simulation produces cache behavior metrics, such as cache hit ratios. The core table lookup must be flexible, to simulate a variety of cache sizes, associativities, etc.

The static code version is partly specialized for two cases: direct-mapped and set-associative by powers of two. Figure 4.4 shows sample code statically optimized for the direct-mapped case. Other possible specialization parameters include line size, associativity, and total cache size. However, the static code is not optimized for these cases, as the number of cases grows with the product of the number of parameter values. For example, special-casing for just 4 associativities, 8 line sizes, and 16 cache sizes would produce 512 different table lookup cases, yet would still fail to handle some cache configurations found in existing hardware, such as the SPARC 5-way associative instruction cache [Case 91] or the DEC AXP 3-way associative unified cache [Dig 94].

VSO builds a cache lookup function that is optimized to the particular line size, associativity and cache size of the cache currently being simulated [Przybylski 88, Przybylski et al. 88]. Figure 4.5 shows code specialized for a particular line size and total size of a direct-mapped cache; associative caches also use an unrolled search loop.

Note that VSO optimizes on the user-specified associativity and line size, but not on other configuration parameters such as the number of caches or the arrangement of caches in the hierarchy [Przybylski 88, Przybylski et al. 88]. However, table lookup computations are a principal bottleneck, so optimizing around these other parameters would not further improve performance.

The simulator design has not sacrificed generality: reported performance is for a simulator that computes metrics for each memory line, and that supports different line sizes and associativities in each cache, multilevel caches, both unified and split I/D caches, write-through and write-back memory update policies, and invalidations in lower-level caches (closer to the processor) caused by a loss of inclusion in higher-level caches.

The dataset is a uniprocessor instruction and data reference trace collected using MPtrace [Eggers et al. 90]. The data drives a simulation of a 2-level cache hierarchy using unified instruction and data caches. The first-level cache is either a direct-mapped cache with 256 16-byte lines or an associative cache with 4 sets of 64 16-byte lines. In both cases the second-level cache is direct-mapped and has 2048 32-byte lines, and is configured to invalidate closer caches on loss of inclusion.

The trace is converted to a compacted binary format in which each trace record is represented by 30 bits for the address and two bits for the reference type (data read, data write, instruction read,

```
/* Static-code direct-mapped cache lookup. */
hash = (trace_address ≫ cache→hash_shift) & cache→hash_mask
line = cache→line[hash]
tag = trace_address & cache→tag_mask
if (line→tag == tag) {
    line→reads = line→reads + 1
    return (HIT)
} else {
    return (MISS)
}

/* Static-code associative cache lookup. */
hash = (trace_address ≫ cache→hash_shift) & cache→hash_mask
lines = cache→lines[hash]
tag = trace_address & cache→tag_mask
assoc = cache→assoc
set = 0
do {
    if (lines→tag[set] == tag) {
        lines→reads[set] = lines→reads[set] + 1
        return (HIT)
    }
    set = set + 1
} while (set < assoc)
return (MISS)
```

Figure 4.4: Statically-specialized simulation of direct-mapped and 4-way associative cache read. The two code fragments are statically specialized to simulate instruction or data reads from direct-mapped and associative caches, respectively; a third fragment (not shown) handles associative caches whose line size is not a power of two. On a hit, the simulation updates per-line metrics. Misses are serviced by returning to more general code.

special). Compacting the trace increases the cost of trace decoding but reduces the I/O costs, I/O-associated overheads and improves "long line" cache prefetching. Here, using an uncompacted form caused the simulator to spend most of its time waiting for I/O; even with the compact representation, I/O still typically accounts for 25% of the execution time.

4.1.3 Data Decompression

The decompression application uses a compression algorithm, pardoz, that is optimized for fast decompression using table lookup. The table maps bytes in the compressed input data stream to strings of bytes in the decompressed output stream. The static code reads a byte from the compressed input stream and uses it to perform an indirect dispatch to an unrolled copy loop. Example static code is shown in Figure 4.6.

Static specialization on the length would avoid the costs of indirection but would generate more than 4^{251} special cases; a better compressor would use more than 15^{240} special cases.[2] Even restrict-

[2]Derivation: the string table is ordered by length. If there are three strings of two possible string lengths, the

```
/* VSO'd direct-mapped cache lookup. */
hash = (trace_address ≫ 4) & 0x7ff
line = cache→line[hash]
tag = trace_address & not(0xf)
if (line→tag == tag) {
    line→reads = line→reads + 1
} else {
    return (MISS)
}

/* VSO'd associative cache lookup. */
hash = (trace_address ≫ 4) & 0x1ff
lines = cache→lines[hash]
tag = trace_address & not(0xf)
if (lines→tag[0] == tag) {
    lines→reads[0] = lines→reads[0] + 1
    return (HIT)
} else if (lines→tag[1] == tag) {
    lines→reads[1] = lines→reads[1] + 1
    return (HIT)
} else if (lines→tag[2] == tag) {
    lines→reads[2] = lines→reads[2] + 1
    return (HIT)
} else if (lines→tag[3] == tag) {
    lines→reads[3] = lines→reads[3] + 1
    return (HIT)
}
return (MISS)
```

Figure 4.5: VSO'd simulation of a direct-mapped and 4-way associative cache read.

Specialized fragments that simulate reads from direct-mapped and associative caches. Both caches have 16-byte lines and a total capacity of 32KB. All masks, etc. are read directly from instruction immediates instead of being loaded from memory.

ing the specialization to just a "few" common cases would still generate a huge number of special cases. For example, tables of mostly long strings or mostly short strings are rare; a mixture of a few long strings and many short strings is far more likely, but there are millions of such "common" cases.

VSO generates a decompressor inner loop that corresponds to the lookup table used for the current input file. Each input byte is used to dispatch directly to an unrolled copy loop that outputs the appropriate string. In addition, the text of each string can be inlined in each arm of the dispatch, saving a load instruction for each byte appearing in the decompressed output. For

table can have the following lengths: {1, 1, 1}, {2, 1, 1}, {2, 2, 1}, and {2, 2, 2}. The combinations {1, 1, 2}, {1, 2, 1}, {1, 2, 2} and {2, 1, 2} are all impossible, because there is a long string to the right of a short string. To generalize, if there are L possible string lengths (5, in the current implementation) and C cases in the string table (255 in the current implementation, one per byte value and one special case), then the table combinations are $\sum_{i_C=1}^{L} \sum_{i_{C-1}=1}^{i_C} \sum_{i_{C-2}=1}^{i_{C-1}} \cdots \sum_{i_1=1}^{i_2} 1$, which is bounded below by $\frac{(L-1)^{C+1}}{(L-1)!}$, which is in turn bounded below by $(L-1)^{C-L}$.

```
do {
    byte = *in
    in = in + 1
    length = table[byte].length
    string = table[byte].string

    switch (length) {
    case 1:   *out = *string
              out = out + 1
    case 2:   *out = *string
              *(out+1) = *(string+1)
              out = out + 2

      ... cases for other lengths omitted ...

    }
} while ((in < inmax) and (out < outmax))
```

Figure 4.6: Decompressor inner loop — static code.

A decompressor inner loop that reads bytes from a compressed input stream and maps them to strings on the output. Each input byte is mapped to a <string, length> pair, and the decompressor then switches on the length to a suitably-unrolled copy loop. Boundary cases, such as output full or input empty are handled by general code.

example, Figure 4.7 shows a specialized version of the decompressor.

Note that for the other applications described in this chapter, the VSO'd code is about the same size as the static code or is smaller. For decompression, the VSO'd code is an order of magnitude larger than the unspecialized static code in Figure 4.6.

The dataset used with decompression is SPARC machine code compressed using a pardoz compressor that encodes the machine code using five decompressed string lengths (1, 2, 4, 8 and 16 bytes). Overall compression is about 2:1, which is typical for machine code. Note that 2:1 compression implies that the average string length is about 2 bytes and that a main loop dispatch is required about every two output bytes.

4.1.4 Sorting

The sorting application sorts a collection of records, and is described in detail in Section 2.1.1. A previous implementation is described elsewhere [Keppel et al. 91].

The sorting routine calls a record comparison predicate to determine the relative order of a given pair of records. The order of field comparisons is selected by the user, and so is known only at program runtime. For F fields in a record, there are more than $F!$ possible fixed predicates,[3] so statically specializing for each of them would cause code size explosion, even for a small number of fields. For example, just 10 fields gives more than a million such comparison functions. Thus, the

[3] Not all fields need be used in a comparison, so if there are F fields, there are $\sum_{i=1}^{F} \frac{F!}{(F-i)!}$ possible predicates.

```
do {
    byte = *in
    in = in + 1
    switch (byte) {
    case 0:   *out = 'a'
              out = out + 1
    case 1:   *out = 'i'
              out = out + 1
    case 2:   *out = 'a'
              *(out+1) = 'm'
              out = out + 2
    case 3:   *out = 'i'
              *(out+1) = 'f'
              out = out + 2

      ... 251 other string cases omitted ...

    }
} while ((in < inmax) and (out < outmax))
```

Figure 4.7: VSO'd decompressor inner loop.

A specialized decompressor inner loop, with dispatch performed directly on the byte value instead of the string length. Each arm of the dispatch corresponds to a particular string, and string values are stored in instruction immediates instead of being loaded from memory. For example, cases 0 through 3 correspond to the strings "a", "i", "am" and "if".

static code uses a routine that can be parameterized to emulate any of the predicates. Once the user selects a field order, however, the predicate does not change during a given sort, so VSO can generate and use the predicate specific to the user's sort order.

The dataset used for the benchmarking is a collection of four-element records with an integer, a character, a floating-point, and a pointer-to-string field, in that order. The unsorted dataset is generated using a pseudo-random number generator and with four equally-likely values for each field, except for the string field which has five equally-likely values. Thus, a predicate invocation on two randomly-selected records should give a 75% likelihood of determining the record order after just the first comparison, a 94% chance after the second, 98.5% odds after the third, a 99.7% likelihood after the fourth, with an 0.3% chance that the records are equal; as the records are sorted, adjacent records are more and more similar and therefore as sorting progresses, each predicate invocation will compare more and more fields.

For simplicity, the dataset was sized to fit entirely in primary memory, but the results should apply for databases where sorting and I/O can be overlapped.[4] Records were sorted by moving them directly, rather than by sorting a collection of pointers to records. Larger records with the

[4]Optimizing sort for large datasets still makes sense where the cost of sorting is greater than the cost of I/O. For small records and some kinds of large records, the cost of computation should overwhelm that of I/O. Where records are large and data is used sparsely in the comparison (e.g., only four of forty fields might be used in the comparison), it should be possible to scan the real records once to build "mini records" with the fields of interest, sort the mini-records, and then sort the real records in a final pass.

same data are more expensive to copy and thus VSO would show lower speedups. Sorting an array of pointers to records (instead of the records themselves) reduces copying costs substantially and thus improves VSO speedups for large inputs, though the reduced copying costs are offset somewhat by the increased data sizes (both the pointer and the record are now referenced) and the decreased cache locality (because records are not clustered, as they are when the actual records are moved).

4.1.5 Debugging

Debuggers implement conditional breakpoints, which halt a program at a user-specified *breakpoint location* only if certain program variables satisfy a user-specified *breakpoint condition*, represented by an expression, such as "y!=12". Breakpoints are implemented by patching the program's breakpoint location with a trap or jump to code that evaluates the breakpoint condition. If the condition is false, the instruction that was originally at the breakpoint location, called a *displaced* instruction, is executed and the program continues. If the condition is true, the program is halted and the debugger is restarted.

Fast conditional breakpoints are implemented by placing some debugger code in the application's address space, so that the code can be invoked directly, without the overhead of operating system involvement. Most previous fast conditional breakpoint studies [Aral & Gertner 88, Kessler 90a, Chase & Hood 91, Bedichek 92] have compared VSO against static code implementations that require several protection domain crossings (kernel traps) on each invocation of the conditional expression; most of the static code's overhead has been in the trap handling code. Even with faster IPC mechanisms, crossing costs would still dominate. The debugger used here implements both static and VSO'd versions, using code in the debuggee. Thus, the speedups measured here are due entirely to VSO. The debugger used in this study is a modified version of the Free Software Foundation's GDB debugger [Stallman 87] that maps static code read-only into the application's address space, so that the conditional expression executes in the debuggee, without domain crossings [Paxson 90b].[5] The debugger code is thus invoked directly, without the overhead of operating system involvement.

The debugger is otherwise fairly conventional. Conventional debuggers do some code modification, typically to patch breakpoint instructions with a `trap` instruction. The debugger implementation used here patches breakpoint instructions with a branch instruction to the static code.[6] When a condition evaluates to true, the debugger code that is mapped into the application simply traps to the debugger for further handling.

Static code must be flexible enough to handle a variety of expressions and values for the breakpoint condition; likewise, it must be be able to execute (simulate) any displaced instruction with any

[5]The debugger code is provided to the debuggee in the form of a debugging library that is linked with the debuggee at build time; the debugger checks for the library by looking for a symbol with a particular "magic" name that cannot be generated directly by the C compiler. Using a debugger library also ensures adequate space in the debuggee for the dynamically-generated code. It would also be possible to dynamically map the code into an unused part of the process address space, with the advantage that programs would not need to be compiled with the debugging library, but with the disadvantage that applications may request particular memory regions, those requests might conflict, and there is no robust way for the debugger to detect such requests in order to remap the debugger code.

[6]Some locations cannot be patched with branches [Kessler 90a, Paxson 90b]. Those that can are set to branch to jump to one of N stubs that identify which breakpoint is being serviced. The number of simultaneous fast breakpoints is thus limited by the number of stubs; additional breakpoints are implemented conventionally.

operands and execute it out-of-line, at a different address than the address where it was originally compiled to run.[7] For example, Figure 4.8 shows code and read-only data used to implement a breakpoint and conditionally execute the displaced instruction. Although the static code performs no type-related dispatching operations, it nevertheless has the form of an interpreter, because the variable addresses and comparisons may take on so many values. Specializing the static code to the particular application would substantially increase the static code size but only slightly improve the performance, because static specialization must allow for arbitrary breakpoints at arbitrary instructions.

In contrast, VSO code evaluates the conditional and displaced instruction directly using simple in-line code. For example, Figure 4.9 shows the same breakpoint condition evaluated using VSO, which fully unrolls the inner loop and specializes each iteration.

Even the "portable" version of fast breakpoints relies on some machine-dependent code. The most significant problem is that all registers are potentially live where the breakpoint is set. Therefore, the handler must restore all registers before restarting the application. In general, doing so uses a register to hold the target address in order to restart the application. The SPARC implementation performs a branch and then uses a "load double" instruction in the delay slot to both reload from the stack the register that holds the branch target and, simultaneously, to restore the stack pointer. The MIPS and AXP platforms lack both hardware and OS support for equivalent operations, so the debugger was never ported to the other machines.

The dataset for debugging was a program with a small inner loop and a conditional breakpoint of the form "stop at *line* if x == 1000". Increasing the size of the loop would decrease the relative cost of breakpoints and thus the speedup of VSO. Using a more complex breakpoint expression, such as "stop at *line* if x[i] == x[j]", would increase the relative cost of breakpoints and thus increase the speedup of VSO.

4.1.6 Further Optimization

Cache simulation and sorting both have additional VSO opportunities that were not explored in this dissertation. For example, in both applications, the VSO'd code appeared in an inner loop and could have been dynamically inlined into the loop. In contrast, bitblt and decompression spend the bulk of their time in VSO'd code, so there is little benefit from inlining in the caller. For debugging, inlining is constrained because of the arbitrary way in which the application may be patched.

Performing larger-grain optimization would have provided additional opportunities for VSO, but would tend to increase the cost of dynamic code generation. In the prototype system, dynamically-generated code was built from manually-constructed trees, and writing the outer loops in the tree form was judged too awkward to make it worthwhile, since one of the goals was to keep the prototype simple.

[7] In principle, the displaced instruction may sometimes be patched back to its original location and then excised again before the next invocation of the breakpoint location. In practice, patching and re-excising works for fewer cases, will have poor performance on machines with non-snooping instruction caches [Keppel 91], and may introduce new debugging problems, since the text segment must be writable to achieve acceptable performance.

```
loop {
    cmd = *thread
    thread = thread + 1
    switch (cmd) {
    case FETCH_ABS:      address = *thread
                         thread = thread + 1
                         push (*address)
    case CONST:          push (*thread)
                         thread = thread + 1
    case NEQ:            push (pop( ) != pop( ))
    case CBREAK:         if (pop( ) != 0) {
                           restart_debugger( )
                         }
    case READ_REG:       push (regfile[*thread])
                         thread = thread + 1
    case PLUS:           push (pop( ) + pop( ))
    case STORE_REG:      regfile[*thread] = pop( );

                       ... other operations ...

    case DONE:           return (*thread)
}

commands_breakpoint_zero[ ] = {
    FETCH_ABS, &y,       /* Fetch value of 'y'. */
    CONST, 12,           /* Fetch '12'. */
    NEQ,                 /* Compare 'y' to '12'. */
    CBREAK,              /* Stop if not equal. */
    READ_REG, 4,         /* Read register 4. */
    CONST, 1,            /* Fetch '1'. */
    PLUS,                /* Add. */
    STORE_REG, 5,        /* Store to register 5. */
    DONE,                /* Done simulating displaced instruction. */
    &displaced_instr + 4 /* Place to restart debuggee. */
}
```

Figure 4.8: Static-code debugger inner loop.

Statically-generated code in a debug library and data downloaded by the debugger into the application in order to implement the conditional breakpoint "break at *line* if y!=12" when "y" is a global variable and the displaced instruction is "add r4,1 → r5". Here, the operator & returns the memory address of the operand. This example does not show either the code that sets up for the breakpoint or the code that restarts the application.

```
push (y)
push (12)
push (pop() != pop())
if (pop() != 0) {
  restart_debugger()
}
push (regfile[4])
push (1)
push (pop() + pop())
regfile[4] = pop()
return (&displaced_instruction)
```

Figure 4.9: VSO'd debugger inner loop.

Breakpoint handling code specialized for a particular breakpoint condition and displaced instruction. In practice, the "push()" and "pop()" operations are inlined; in the VSO'd code, inlining enables optimizations that are not possible in the statically-generated code.

4.1.7 Size of Static Specialization

Will VSO eventually lose its size advantage? As memories grow, some static cases that are "too large" today will eventually fit into memories. However, there are situations where reasonable static specializations exhaust all forseeable storage. For example, specializing data decompression just with respect to the string lengths generates more than 4^{251} cases; more cases than elementary particles (about 10^{80}) in the known universe. On top of that, the current decompression implementation also specializes with respect to the actual string values, multiplying the number of cases even further, with anywhere from 255 to 255^{16} possible string values for *each* of the 4^{251} cases! The upshot is that ongoing advances in storage technology are unlikely to solve the space explosion problems any time soon.

4.1.8 Rejected Benchmarks

Two benchmarks that initially seemed like good candidates were rejected because each had a core algorithm that could be recast as two-stage algorithm, with an approximate first stage serving as a "filter" for an exact second stage that handled the cases passed on by the first stage. In both cases, the first stage was the performance bottleneck, and in both cases, the first stage was not amenable to VSO.

String Matching

String matching [Beeler et al. 72, Boyer & Moore 77, Standish 80, Consel & Danvy 89, Haertel et al. 90] appeared to be an ideal candidate for VSO, because the search engine has an inner loop that iterates over the pattern string and, during any given run of the matcher, the pattern (string) is constant [Beeler et al. 72].

The matcher's inner loop is VSO'd by duplicating and specializing the inner loop for each character in the pattern. As a result, a comparison of the form "`c == pat[j]`" is transformed to a series of comparisons "`c ==` 'pat_j'", where each pat_j is a (runtime constant) character from the pattern. The VSO'd code encodes each pat_j as an instruction immediate, removing loads from the inner loop. Similarly, each iteration for a partial match decrements a counter, with zero indicating a successful match. By unrolling the loop, the decrement and the test for zero can be executed once by the application-specific compiler, instead of being executed for each iteration.

However, the apparent payoff is not realized in practice. Although the basic Boyer-Moore algorithm can be VSO'd as above, the inner loop can be rewritten statically to search quickly for just the last character of a match[8] and then revert to the general match [Boyer & Moore 77, Haertel et al. 90], which removes inner loop tests and dependence on the (candidate) specialization variable. Since mismatches are the common case, matching table lookups are also arranged to be idempotent, so that the inner loop can be unrolled and the match test performed only occasionally, instead of being done on every iteration [Haertel et al. 90]. The inner loop is thus an average of less than five RISC-like instructions and contains only computations that depend on input data (see §B.17 for details).

For typical matching problems, over 80% of the program's execution time is spent in the optimized loop [Boyer & Moore 77]. Pathological cases are limited by the performance of the non-optimized loop, but pathological cases depend on both the pattern string and on the input data, are rare in practice, and would show only modest speedups anyway.

Regular Expression Matching

Regular expression matching finds a sequence of characters that match a pattern specified using a regular language; the language constructs manipulate only characters, so no runtime type-related optimizations are possible. Regular expression matching is appealing, because each unit of work in a regular expression is small, and thus dispatching between cases may have substantial overhead [Thompson 68].

As with string matching, simple algorithms are improved by VSO [Thompson 68, Leone & Lee 96b] but faster algorithms implemented with static code can handle most of the cases are not amenable to VSO. First, regular expressions that contain strings can be matched quickly by using a string matcher to guide application of the regular expression matcher [Hume 88, Haertel et al. 90]. Second, VSO tends to improve the performance of NFA pattern matchers but not of DFA matchers. DFA pattern matchers have exponential worst-case size, which appears to make NFA matchers more important. However, an incrementally-built DFA matcher can operate in bounded space by dynamically converting from an NFA representation to the larger DFA representation and throwing away parts of the DFA when space runs out [Hume 88, Haertel et al. 90].[9] The incrementally-built space-friendly DFA form can be implemented using just a single indirection per input character

[8] The Boyer-Moore algorithm searches from right to left; the important observation here is that the rightmost character normally mismatches, and so most match tests are performed on the rightmost character; thus, that match test is further specialized [Boyer & Moore 77]. Oddly, this optimization is not shown in most textbooks, though it substantially improves performance; perhaps it is omitted because it does not improve the computational complexity.

[9] This construction by Alfred Aho, circa 1975 [Hume 88].

(§B.17), which is no more than is required by the VSO'd NFA matcher. Although the DFA matcher may "thrash" in a finite space, such cases are rare, and the advantage of a VSO'd NFA matcher is correspondingly small.

VSO may be more valuable where the pattern is more general than can be expressed with a regular expression [Haertel et al. 90]; such cases were not investigated.

4.2 Methodology and Metrics

This section describes how the workload was measured and what metrics were used to compare the performance of the various implementations. Section 4.2.1 describes the measurements that were used to determine the time taken by static and VSO versions of each program, and also describes how the dynamically-generated code was analyzed and compared to the static code. Section 4.2.2 describes measurements based on tracing of *event counts*, such as the number of loads and stores and cache misses, and on trace-driven simulation results.

4.2.1 Elapsed Time

The timing metrics are (1) the dynamic compile time, (2) the asymptotic speedup and (3) the breakeven point at which the total execution time of two implementations is equal.

Using elapsed time captures a variety of effects, such as I/O and paging overhead, cache behavior, processor architecture, processor implementation, and so on. Elapsed times also report *that* performance differs but does not always explain *why* performance varies across applications and implementations. However, the elapsed time does accurately report the behavior on a real machine, including the effects of complex interactions between application and architectural features; these interactions are otherwise hard to model [Bedichek 95b].

The *asymptotic speedup* is the performance improvement for very large inputs. Alternatively, it can be viewed as the speedup on processing each data item after the breakeven point has been reached. The *breakeven point* is the input size at which statically-optimized code and VSO'd code have the same cost. For smaller inputs, the VSO'd code is slower; for larger inputs, faster. To determine the asymptotic speedup and breakeven point, the execution time was measured with a variety of input sizes.

The measurements are of total execution time and thus include time that is spent in code that is not improved with VSO. Overall speedups are used, because they best reflect the contribution of VSO to application performance. Raw speedups of just the VSO'd code can ignore important aspects of an application's overall behavior, such as time spent executing statically-optimized code and cache and I/O effects. For example, sorting divides its time between record comparison and the basic sorting algorithm, but only record comparison is VSO'd (§4.1.4, §4.1.6); in one case, the sorting application using tree VSO had a 1.40 asymptotic speedup but the VSO'd predicate alone had a 2.20 asymptotic speedup (§5.4). As an alternative, *microbenchmarks* are considered briefly in Section 5.4. Microbenchmarks show just the time spent in the VSO'd code and thus show how code quality was improved due to VSO alone, divorced from the effects of caching, I/O and so forth.

Measurement

The dynamic compile time is measured differently for the tree and template compilers. For the template compilers, the elapsed compile time was simply measured. For the tree compilers, the execution time was composed of the tree generation time, the code generation time, the dynamic linking time, and the instruction cache updating time. The tree generation, dynamic linking and cache updating times are measured directly, but as discussed above (§3.1.6), the code generation time is estimated.

Timing measurement is tricky, because measurement tools have limited resolution, change the execution time (intrusion), and may change the code generation strategy. For example, in some of the programs over 30% of the execution time is spent in the profiler. Profiling also sometimes affects the code generation strategy; for example, it sometimes turns leaf routines into non-leaf routines. In addition, most coverage tools typically do not profile dynamically-generated code.

To minimize measurement side-effects, the timing measurements use elapsed time (wallclock) timers, and the short-running phases were placed inside of a loop to execute them repeatedly, increasing the execution time relative to the timer accuracy. Repeating a phase artificially improves cache hit ratios and, therefore, artificially reduces execution times. However, such "warming" effects occur for both static and VSO'd code.

Analysis

Measuring the execution time shows when VSO produces speedups, but it does not explain *why* improvements are or are not achieved. In order to understand why performance changed, the statically-compiled code and dynamically-compiled counterparts were examined by hand. Hand-examination was also required to ensure that the code generator was producing code equivalent to that produced by a locally-optimal code generator (see §3.1.6).

A disadvantage of hand-examination is that it must be repeated for each application, input and optimization level. Thus, the benchmarks were first run over varying input datasets to discover representative datasets, and then features that affected the generated code were fixed. For example, for decompression the compressed file size was varied, but the decompression table was fixed; for sorting, the number of records was varied but the field comparison order was fixed. The performance of bitblt and cache simulation varied enough with the dataset that two points were used for each. Cache simulation exposes enough underlying details that the two cases are reported separately; for bitblt, the results of two extreme cases (fully-aligned and fully-unaligned) were averaged.

The build time, breakeven, speedups and event counts are all data-dependent. In particular, the compile time and the generated code both depend on the data values used to drive VSO. It is possible to construct pathological cases with especially good and bad behavior. For example, the build time for sorting depends on which fields of a record are being compared, and the VSO speedup depends on the record size, the values of the fields in each record, and the initial order of the records. A variety of plausible input values were used for each application, and, in practice, the data-dependent variations were typically small.

Static Code Implementations

The static code was compiled with whatever single optimization level gave the best performance (shortest execution time) and the static code for the VSO implementations was compiled at the same level. On the MIPS, the static code was not compiled using the u-code optimizer, because a single executable image was used for both the static and VSO'd runs[10] and the u-code optimizer is unable to optimize programs that use dynamically-bound code. Although the u-code optimizer might have generated better code, simple experiments showed the optimizer took advantage of "benchmark constants" that would have been variables in real programs.

Many remaining inefficiencies in the static code were removed by the time-honored technique of iteratively examining the compiler output and editing the source code in order to improve the generated code. Although human intervention is not generally practical, it allowed our state-of-the-practice compilers to produce state-of-the-art code.

Typically, the static AXP code showed benefit from high levels of optimization; the fastest code was sometimes produced by GCC [Stallman 93] and sometimes by the vendor compiler. The MIPS and SPARC static code was usually fastest with the next-to-highest level of optimization, using either the GCC or vendor compilers.

Microbenchmarks

The microbenchmarks were implemented by extracting just the VSO'd code and calling it repeatedly from a "driver loop" that runs the code over a fixed dataset. In places where the execution time is strongly data-dependent, the code was run over several fixed datasets, although the microbenchmark datasets do not necessarily cause the same path frequencies as the macrobenchmark datasets. The microbenchmark speedups include time needed to execute "driver" code that repeatedly calls the measured static or VSO'd code. The VSO'd code is faster, so the fixed driver overhead is a relatively larger overhead; thus, reported times are underestimates of VSO's relative speedup.

4.2.2 Tracing and Simulation

Execution time measurements are precise but present only an aggregate picture of VSO's effects. Thus, the workload was also measured by tracing each application and using the traces to drive analysis. The analysis helps explain what factors contribute most to VSO's performance improvement and also what machine features most limit VSO's performance, thus giving insight about how VSO will affect performance across a variety of machines.

The workload was traced using instruction-level program tracing, which yields a list of *events*, such as loads, stores and branches. The events are then used to analyze the instruction mix, drive cache simulators, and so on. Although execution time estimates derived from event traces are less accurate than measured execution times, event traces provide a finer-grained view of application behavior. They are thus better at explaining the principal bottlenecks to overall performance and in turn better at explaining what architectural features most strongly affect performance.

[10] For example, the program was invoked with the `-vso` or `-static` flags.

Tracing Methodology

The workload was measured using analyzers built on top of Shade [Cmelik & Keppel 93, Cmelik & Keppel 94], which instruments existing binaries and gathers metrics for dynamically-generated V8 SPARC machine code. Unfortunately, no equivalent AXP or MIPS tracing tools were available.

For these studies, just the inner loop code for each application is traced, so the statistics are isolated from the details of initialization, code generation, and so on. Tracing was restricted by setting start and stop "watchpoints" in the traced code and tracing only between watchpoints.

Shade traces only user-level code, which isolates the traces from details of the operating system, but which also excludes the effects of code executed by the operating system on behalf of the application, for example, for I/O. As a result, the state of the simulated machine – caches, branch prediction tables, etc. – does not exactly mimic that of the real machine [Uhlig et al. 94, Bedichek 95b, Rosenblum et al. 95]. In particular, I/O-intensive programs have user-level misses caused by operating system cache accesses. The effect of such I/O interference is estimated using two bounds: by simulating system calls assuming that they cause no state change, and so that they completely flush the caches.[11]

Finally, note that the cache simulations use virtually indexed and tagged caches, whereas many real systems use physically indexed and tagged caches. Although most cache studies assume that both produce equivalent results, the inaccuracies may actually be substantial [Uhlig et al. 94]. Unfortunately, tracing physical addresses requires system-level traces or system-level support; neither was available.

Instruction Mix

The *instruction mix* is a breakdown of instructions by category, such as load, store, arithmetic, and so on. The instruction mixes reported in Chapter 6 were gathered using the ifreq analyzer [Cmelik 93] built on top of Shade.

Although many RISC instruction sets are largely similar, there are a few variations that were handled specially. One variation is that the SPARC architecture [Sun 91] supports *anulled* branch delay slot instructions. Although VSO substantially changes the absolute number of anulled instructions, most of the workload uses only a small total number of them, and thus VSO does not dramatically change the percentage of anulled instructions in the overall instruction mix. Thus, the reported mixes are based on the number of instructions completed rather than the number of instructions issued.

The SPARC architecture also supports *register windows*, which cache some stack state. In the normal case, the save and restore instructions behave like ALU instructions. When the register cache overflows or underflows, however, the register window instructions may perform stores and loads. For these applications, a realistically small register cache eliminates most spills and restores, so save and restore appear in the results only as ALU operations. A related issue is that register windows may eliminate some saves and restores that occur in other architectures; these effects could not

[11] Such simulation does not quite bound the real behavior, since interrupts also affect processor state.

be traced because AXP and MIPS tracing tools for dynamically-generated code were unavailable. However, VSO was mostly used to optimize either (a) routines that ran for a long time between entry and exit, or (b) typically used only a small number of registers. Therefore, all platforms should have similar behavior for load and store traffic for inner loop calls and returns.

Contributions To The Execution Time

An application's execution time depends on the number of instructions it executes and the time taken by each instruction. The event traces were used to drive a simple timing simulator in order to determine the factors that contributed towards the time required by each instruction.

The contribution of each factor is expressed as the average over all instructions, called the *cycles per instruction* or *CPI*. The timing simulator assumes a base cost of one cycle per instruction. Most events, such as read misses, are assumed to have a fixed cost, such as "each mispredicted branch costs N cycles". In real systems, the costs are not truly independent, because instruction-level parallelism may overlap several such costs. However, the processor implementations used in this study make relatively little use of low-level parallelism other than pipelining, and so fixed costs appear to be a good estimate except as noted below.

For events with a fixed cost, each event's contribution to the simulated CPI is ΔCPI (Equation 4.1), which is simply the frequency of the event times the cost. For example, if loads are one quarter (0.25) of all instructions and the data read miss rate is one percent (0.01) and the cost for a miss is 30 cycles, then the contribution to the CPI is $0.25 \cdot 0.01 \cdot 30 = 0.075$, or just under a tenth of a cycle per instruction.

$$\Delta CPI = \frac{\#\text{events}}{\text{total instructions}} \cdot \text{cost} \tag{4.1}$$

Correspondingly, the estimated CPI is computed as one – the base cost for an instruction – plus the cost from each type of fixed-cost event (Equation 4.2). Fixed-cost events include load conflicts, which cost one cycle; instruction and data read misses, which cost 30 cycles; and mispredicted branches, which cost one cycle.

$$CPI = 1 + \sum_{i=1}^{\#\text{ event types}} \Delta\text{CPI}_i \tag{4.2}$$

Stores (memory writes) are modeled by a fixed cost estimate. The estimate is an approximation because the cost depends strongly on the timing of preceding stores. Each store goes to a store buffer. When the buffer has space, the store completes immediately; otherwise the store stalls until there is buffer space. In the worst case of back-to-back stores, each store takes the memory write time, 15 cycles. The fundamental problem here is that neither event tracing nor the simple timing simulator provide sufficient timing information to distinguish between delays due to a single long-running store and delays due to a collection of other factors.

The store-induced stall cost is thus approximated by computing two alternatives. The worst-case cost is when all store instructions appear back-to-back and all stores but the first few are delayed by

Table 4.1: A brief summary of the platforms and their processors.

Platform	Processor	MHz (Width)
DEC 3000/400 [Dutton et al. 92]	AXP [Sites 92]	133 (2)
DECstation 5000/125 [DEC 92]	MIPS R3000 [Kane 87]	25 (1)
SPARCstation-2	SPARC V8 [Sun 91]	40 (1)

The column "MHz (Width)" gives the clock rate and issue width; the peak instruction issue rate per second is the product of the two numbers.

the memory write time. Alternatively, the stores may be evenly distributed, so that the buffer is kept at the same average fullness, and buffering-induced delays are overlapped with other instructions as much as possible. Note that an even distribution of writes is not a best-case estimate, since instruction costs vary.

Averages

For each of the above metrics, Chapter 6 reports values for each application and also reports a *composite*, which is the unweighted average across all applications. The composite helps to point out overall trends across applications and where specific applications differed from the average. An unweighted average is used, because there was more than an order of magnitude variation in the number of instructions executed by each application; weighting would bias the composite towards those applications that executed the most instructions.

4.3 Platforms

The three platforms used in these studies are similar in many ways. For example, all use RISC processors with 32 integer registers and have direct-mapped caches. The platforms are summarized briefly in Tables 4.1, 4.2 and 4.3. Although there are a variety of minor differences, such as the size of instruction immediates, the studies in Chapters 5 and 6 show the following features are important:

- The MIPS and SPARC platforms are older machines and have little or no branch prediction penalty. The AXP has up to a 5-tick penalty for mispredicted branches; with up to two instructions issued each cycle the effective penalty can be as great as 10 instructions.

- The SPARC system has a unified I/D cache, while the MIPS and AXP systems do not.

- The AXP system has a 4-word write buffer, the MIPS system has a larger 7-word write buffer, and the SPARC platform uses a smaller 2-word write buffer. The MIPS may thus execute store instructions in larger clumps without saturating the memory bandwidth, while a much smaller number of closely-spaced writes will overflow the SPARC write buffer.

Table 4.2: Cache miss costs for each platform.

Platform	MHz (Width)	L1 Miss nSec	L1 Miss cycles	L1 Miss instr	L2 Miss nSec	L2 Miss cycles	L2 Miss instr
DEC 3000/400	133 (2)	90	12	24	225	30	60
DECstation 5000/125	25 (1)	1000	25	25		–	
SPARCstation-2	40 (1)	750	30	30		–	

Cache miss costs expressed in nanoseconds, as a number of cycles, and as the number of instructions that might have been issued during the miss. The number of instructions is computed by attributing N instructions to each cycle in which a N-wide issue might stall. Miss costs were not listed by the manufacturers and so were determined experimentally.

Table 4.3: A brief summary of the memory hierarchy for each platform.

Platform	L1 Line Size	L1 Cache Sizes I	L1 Cache Sizes D	L1 Policy	L2 Cache
DEC 3000/400	32B	D-M 8KB	D-M 8KB	write-through	D-M 512KB
DECstation 5000/125	4B	D-M 16KB	D-M 16KB	write-through	–
SPARCstation-2	16B	D-M 64KB unified		write-through	–

"D-M" is "direct-mapped". None of the caches are sectored (subblocked).

- The MIPS and SPARC systems have larger first-level caches (16KB I, 16KB D and 64KB unified, respectively), while the AXP system has smaller first-level caches (8KB I, 8KB D).
- Compared to the AXP platform, the MIPS and SPARC systems have relatively larger caches and relatively smaller worst-case miss penalties.
- The MIPS and SPARC systems single issue, while the AXP system can dual issue some instructions.
- The AXP has 64-bit registers and datapaths; the MIPS and SPARC are both 32-bit architectures.

The importance of these features is discussed in the following chapters.

Chapter 5

VSO Performance Results

O mighty Caesar! dost thou lie so low?
Are all thy conquests, glories, triumphs, spoils,
Shrunk to this little measure?

–Mark Anthony, in William Shakespeare's *Juilius Caesar*

This chapter presents various component and overall execution time measurements for both the static and the VSO implementations of the applications. The results demonstrate that VSO can produce code that is faster than even the best possible hand-crafted static assembly code and show that a variety of application and implementation factors influence its profitability.

Section 5.1 examines the speedups and breakeven points of the static-code and VSO implementations, including both portable and hand-coded assembly versions of each application. Section 5.2 reports the observed speedups across various platforms and describes application and architecture features that cause widely varying speedups, both across applications and across architectures. Section 5.3 studies the effects of varying the degree of runtime optimization and shows that for tree VSO, fast local code generation provides code as good as that produced by a more sophisticated but slower intraprocedural optimizer. Section 5.4 presents microbenchmark results that show VSO's effects on just the optimized code, rather than the effects on the application as a whole.

The results presented here are all based on measured execution times; Chapter 6 gives additional measurements based on tracing and simulation.

5.1 Comparing VSO Implementation Strategies

This section quantifies tradeoffs between various VSO strategies by comparing four versions of each program. Key inner loops of each program are implemented using hand-tuned C code, hand-crafted static assembly, tree VSO and hand-crafted template VSO. The results show the realized speedups from VSO, the cost of producing code dynamically and their effect on the breakeven point, which is the smallest dataset size for which the VSO'd implementation is profitable. In addition, since static hand-written assembly code represents the best code that can be produced statically, faster VSO implementations show that VSO can produce code that is better than the best possible static code.

Assembly-code implementations were written only for SPARC platforms, so this section presents SPARC measurements only. Cross-architecture results are presented below, in Section 5.2.

5.1.1 Breakeven & Speedup

For bitblt, the build time, speedup, and breakeven all depend on the size, shape and alignment of the source and destination regions. Although the VSO'd code shows asymptotic speedups (Table 5.1; see pg. 41 for the cases that were measured) – that is, better overall code quality for the inner loop – the tree build time is more than the static code execution time on a typical screen region. Hence, VSO would pay only for atypically large regions or if the VSO'd bitblt could be cached for later reuse. VSO'd bitblt was profitable in other experiments [Pike et al. 85, Locanthi 87, Miranda 87, Foley et al. 90], because they used less-optimized static code and their compiler was hand-coded and machine-dependent and thus faster; yet even there, small regions used static code.[1]

For cache simulation, the asymptotic speedup for tree VSO is smaller, only 5-10%. However, the breakeven point is also small, about 1,000 references; since cache simulators typically consume millions of references from a trace, VSO thus "always" pays for cache simulation.

The debugger payoff depends on how the conditional breakpoint is being used. The data in Table 5.1 shows behavior for a simple conditional expression. Using a more complicated expression would tend to improve the advantage of VSO'd code; setting a breakpoint in a more complicated loop would tend to diminish VSO's asymptotic speedup, but would not change the breakeven point. In the other applications, reuse of the VSO'd code – and thus the breakeven point – is computed easily from information such as the size of the input file. For the debugger, it is difficult to predict how many times the VSO'd code will be used. However, compilation times are small compared to the user response times, so VSO does not affect the observed time to set a breakpoint. The static code debugger used here maps static code into the debuggee and executes it without domain crossings (§4.1.5), so the reported improvements are due entirely to VSO. Previous conditional breakpoint studies have compared VSO against static code that require several protection domain crossings on each invocation of the conditional expression; although these systems have used VSO, the principle speedup was from eliminating domain crossing overheads. For example, the GNU GDB debugger [Stallman 87] is three orders of magnitude slower, due to the domain crossing overhead.

Decompression and sorting provide speedups, but only for larger inputs. Thus, decompression and sorting must decide at runtime whether to use static or VSO'd code. In contrast, cache simulation and debugging were always profitable. Template compilers dramatically reduce compilation costs and thus help to reduce the breakeven point and in turn increase the number of situations in which VSO may be applied. For example, sorting with tree VSO pays off when more than about 1,200 records are being sorted; with template VSO when more than ten are sorted.

For each application, template VSO'd code is faster than tree VSO'd code. The template VSO'd code and the static assembly code were crafted by hand and thus benefitted from transformations that were missed by the retargetable code generators used in the static compiler and in the tree

[1] The asymptotic speedup of bitblt is lower than previously reported [Keppel et al. 93] because a static code performance bug has been fixed.

Application	Implementation	Asymptotic Speedup (m)	Breakeven Point
Bitblt	static C	1.00	–
	tree VSO	1.14	megapixels
Cache–1	static C	1.00	–
	tree VSO	1.06	1000 refs
	static assembly	1.09	–
	VSO template	1.14	75 refs
Cache–4	static C	1.00	–
	tree VSO	1.10	1000 refs
	static assembly	1.07	–
	VSO template	1.14	150 refs
Debugger	static C	1.00	–
	tree VSO	1.50	1500 checks
	static assembly	1.25	–
	VSO template	5.40	700 checks
Decompression	static C	1.00	–
	tree VSO	1.13	1.19MB
	static assembly	1.28	–
	VSO template	1.56	24KB
Sorting	static C	1.00	–
	tree VSO	1.44	1200 records
	static assembly	1.20	–
	VSO template	1.65	10 records

Table 5.1: Asymptotic speedups and breakeven points for the five applications.

All speedups are computed relative to static C, to show overall code quality. The speedup of template over assembly is computed by dividing the template's asymptotic speedup by the assembly's asymptotic speedup. For example, the relative speedup of the assembly implementations of the debugger is 5.4/1.25, which is 4.3. Breakeven points are computed against static code using the same code generation technology: tree VSO breakeven points are computed relative to static C code, and template breakeven points are computed relative to static assembly. Entries marked "–" are base cases for computing breakeven points. The debugger entry marked *traditional* is for a conventional debugger implementation with high IPC overhead. Bitblt was not implemented in assembly, see §4.1.1.

VSO'd code. When static assembly is derived using a compiler, the code quality is lower; in much the same way, templates derived automatically by the compiler (§2.3.2) would have code quality similar to the tree VSO'd code. Template VSO, however, would retain the advantage of smaller runtime compile times, discussed further in Section 5.1.2.

In all cases, hand-crafted VSO is faster than hand-crafted static code, showing that VSO exposes optimizations that fundamentally cannot be performed statically. For cache–4, debugger and sorting, even tree VSO produces code that is faster than the best static assembly; these results show that

VSO'd code that is produced using slower retargetable code generation technology can sometimes be faster than the best hand-crafted static code.

5.1.2 Code Generation Costs

VSO's profitability depends on both the final code quality and on the cost of generating the code at runtime. As shown in the previous section, code generation costs are large enough that breakeven points are nontrivial. As described in Section 1.1, faster code generation directly improves the breakeven point at which VSO shows benefit. This subsection analyzes component costs of runtime code generation.

Table 5.2 summarizes VSO's runtime costs using both the tree and template compilers. The tree compiler costs are broken down into tree build, code generation and linking costs. As described in Section 3.1.6, the tree code generation cost is estimated by multiplying the number of generated instructions by 1,000 instructions per generated instruction and the SPARC instruction issue rate (Section 4.3), assuming 2 cycles per instruction. The template compiler is one-pass, so the costs cannot be subdivided. In all cases, cache flush times were negligible (§3.4). For the debugger, there is an additional cost (reflected in the *total* column) to move code or data from the debugger to the debuggee.

The tree VSO compiler's tree build, code generation and link costs usually have similar ratios, with code generation time typically the smallest. As described above, it should be possible to substantially reduce the build and link times, because the tree form supports hole manipulation operations that are far more general than needed for VSO (§3.1.4), and because the linker manipulates files, rather than code that is generated in-core (§3.1.7). Improving build and link times would substantially reduce the startup cost and thus move the breakeven point to smaller inputs.[2]

Template compilers are much faster than fast retargetable code generators, even though the latter take only hundreds of instructions per generated instruction [Fraser & Henry 91, Engler & Proebsting 94]. Template compilers are fast because they produce machine code directly, are one-pass and are specialized to both the application [Henry 87] and the target machine.

The template compilers used in this study execute 35-45 machine cycles per generated instruction. The template compiler costs reported here are conservative, because the compilers were built to emphasize final code quality and ease of development; compiler speed was largely ignored. For example, templates are small, aligned, and their size is known statically, yet the templates were copied using a general-purpose copy routine instead of using an unrolled and inlined copy. Similarly, template holes are often patched using calls to small out-of-line routines instead of using inlined code (§3.2).

The right column of Table 5.2 shows that programs with similar breakeven points may spend vastly different amounts of time performing code generation. For example, at breakeven, code

[2] The payback varies depending on the application. For many applications, the number of VSO'd code invocations is proportional to the data set size, so reducing the code generation cost by 50% reduces the breakeven point by 50%. Sorting, however, typically calls the VSO'd predicate $O(n\ lg_2 n)$ times, so reducing the code generation costs by 50% would reduce the breakeven from 1,200 records to about 650 records, which is about 45%.

Application	Implementation	Compile Time (ms)				% of RT at
		build	cgen	link	total	breakeven
Bitblt	tree VSO	12	2	12	26	N/A
Cache–1	tree VSO	2	1	7	10	6%
	template				0.02	5%
Cache–4	tree VSO	10	3	10	23	9%
	template				0.05	7%
Debugger	static C				4	–
	tree VSO	6	2	6	17	40%
	static assembly				5	–
	VSO template				17	83%
Decompression	tree VSO	63	45	11	119	12%
	template				3	18%
Sorting	tree VSO	5	10	8	23	29%
	template				0.033	27%

Table 5.2: Measured startup costs for VSO implementations.

The tree compiler costs are broken down into tree build (*build*), code generation (*cgen*) and linking (*link*) costs. Template compiler costs cannot be subdivided. The costs are measured using wallclock timers, except for code generation cost for the tree VSO compiler (*cgen*), which are estimated (§3.1.6). Debugger startup costs include the time to download from the debugger to debuggee. The figures for template compilers reflect breakeven compared to static assembly code. Entries marked – are base cases for computing breakeven points. The right column, *% of RT at breakeven* is the percentage of execution time spent in code generation for program runs that reached breakeven; it is equal to $1 - \frac{1}{speedup}$. Note that template compiler numbers reported as the breakeven relative to static assembly code, whereas speedups in Table 5.1 are reported relative to C code.

generation is less than 10% of the time for cache simulation but is 40% to 80% of the debugger's time. Yet both applications reach breakeven with a relatively small number of invocations of the VSO'd code. Cache simulation is profitable because the code is reused many times. On the other hand, debugging has high enough speedups that few uses are needed in order to pay back its compile costs.

5.2 Observed Optimizations and Cross-Architecture Comparisons

The benefit of VSO varies widely across architectures (Table 5.3). For example, sorting showed asymptotic speedups of 1.4 on the SPARC platform, 1.8 on the MIPS platform, and 2.5 on the AXP platform; and cache–1 showed a slowdown (speedup less than 1.0) on the AXP platform. Debugging is not reported because the AXP and MIPS platforms lack the needed support for user-level restart (§4.1.5).

Even the range of speedups (the ratio of the largest and smallest speedups for an application)

Application	AXP	MIPS	SPARC	Spread
Bitblt	1.40	**2.05**	1.14	1.80
Cache–1	0.96	**1.10**	1.06	1.15
Cache–4	1.02	**1.10**	**1.10**	1.08
Decompression	1.04	0.95	**1.13**	1.19
Sorting	**2.53**	1.92	1.40	1.81

Table 5.3: VSO's asymptotic speedups over static code for each application and machine. The column *Spread* shows the ratio of the best and worst speedups using VSO. Entries in **bold** show the best speedup for the application.

varied widely across applications, and there was no single machine that showed consistently good or bad VSO speedups. The wide variation in speedup is perhaps surprising, given that the various architectures are all RISCs and are thus similar in many respects, and given that VSO typically performs only simple optimizations.

This section focuses on a variety of architectural and application characteristics that produced the observed behavior. The section first describes the optimizations that VSO performed across all architectures; then it explains why the optimizations caused varying speedups across the platforms.

5.2.1 Observed Optimizations

VSO enables a variety of optimizations, summarized in Figure 5.4. The optimization *constant folding* takes advantage of runtime variables that are constant over the lifetime of the VSO'd code, folding their values with static constants and other runtime constants. *Dead code elimination* discards both code that is dynamically unreachable and also eliminates branches around that code. *Load elimination* promotes runtime constants from memory variables to instruction immediates, reducing both the number of memory references and the number of instructions executed. *Loop unrolling* eliminates branches, operations needed to compute loop termination, and loop-dependent operations such as updates to array indicies. Except for some bitblt cases, VSO unrolled the loops completely, eliminating *all* looping overhead. VSO also sometimes caused degradation, labeled *Self-Interference*.

5.2.2 Load Elimination

VSO treats variables as constants and therefore replaces memory-referencing load instructions with instructions that use immediates, as shown in Figure 5.1. Replacing loads with instruction immediates is an example of strength reduction, where an expensive operation, in this case a memory reference, is replaced by a cheaper one, use of an immediate. The immediate is cheaper because it is prefetched with the instruction [Deutsch & Schiffman 84], occupies no space in the data cache, cannot cause data misses, can sometimes be merged with a subsequent instruction (Figure 5.1b), and is available without a delay slot.

Application	Constant Folding	Dead Code Elimination	Load Elimination	Loop Unrolling	Self-Interference
Bitblt				√	
Cache–1	√		√		√
Cache–4	√		√	√	
Decompression			√		√
Debugger		√	√	√	
Sorting		√	√	√	

Table 5.4: Optimizations enabled due to VSO's better runtime information. *Self-Interference* shows degradation caused by specialization-induced cache interference.

```
ld   0(r4) → r5        add  r6,#imm → r6    set  #imm → r5     low  #imm → r5
add  r6,r5 → r6   ⇒                         add  r6,r5 → r6    hi   #imm → r5
                                                               add  r6,r5 → r6

      (a)                    (b)                  (c)                 (d)
```

Figure 5.1: A static code fragment and three possible VSO'd equivalents. In (a), 0(r4) changes rarely and is replaced by #imm in the VSO'd code. Various instruction sequences are needed depending on the relative size of the variable and the sizes of immediates. In all cases, however, the VSO'd code has one less data reference.

Small data values can fit in a single instruction immediate, but large values such as floating-point values and pointers may require several instructions, as shown in Figure 5.1d. However, such expansion never occurred in the applications used here. Expansion would be more likely when using a machine with particularly small immediate fields, such as the KSR-1 [KSR 94], which had 8-bit immediates.

Except for expansion, replacing memory instructions with immediates should, intuitively, always produce speedups. However, the effectiveness of load elimination depends on the cache architecture and on the instruction issue implementation.

For example, the SPARC platform used here has a unified instruction and data cache that is single-ported. Thus, each load can interfere with prefetching the subsequent instruction and may cause loads to take two cycles. In contrast, the MIPS platform has split instruction and data caches, so cache hits can be serviced as fast as they are issued. Also, there are often sufficient instructions to fill the load delay slots. These factors are one reason why VSO shows such different speedups on the MIPS and SPARC versions of decompression: even though the static code and VSO'd versions of the inner loop are similar on all machines, removing load instructions from the SPARC inner loop eliminates more overhead than it does on the MIPS. Conversely, for sorting, most of the instructions that get eliminated are non–load instructions, and the remaining load instructions are relatively more expensive on the SPARC platform, so the speedups are correspondingly smaller.

The effectiveness of load elimination is also influenced by the instruction issue implementation. For example, the AXP has 2-wide superscalar issue that can issue one ALU operation and one memory operation every cycle. In one part of the statically-compiled cache simulation code, the AXP issues pairs of memory reference and arithmetic operations. Although the corresponding VSO'd code has 20% fewer instructions, it uses the same number of arithmetic instructions as the static code and therefore takes the same number of cycles to issue. Since the AXP also has a single memory unit, programs could, in principle, benefit if VSO converted memory-intensive – and thus single-issue – code into dual-issue mixed code.

5.2.3 Loop Unrolling

Static loop unrolling usually *partially* unrolls loops, so that loop control tests are executed less frequently. For loops with a small iteration count, VSO can *fully* unroll the loops, eliminating all iteration overhead. VSO did full loop unrolling for sorting, associative cache simulation, and sometimes for bitblt.

As with static unrolling, dynamic unrolling both eliminates looping overhead and also enables other optimizations. For example, full unrolling of the sorting application's inner loop improves use of the AXP's superscalar issue by removing ALU operations used for loop control. The resulting code has a better mix of memory and ALU operations and is thus dual-issued some places that the static code only single-issues. Since the tree compiler generates code at runtime, it may need to include a (simple) instruction scheduler in order to take advantage of superscalar issue; in contrast, templates may be scheduled statically.

Since full unrolling is applied to loops that iterate just a few times, full unrolling eliminates those loop-control branches that are the hardest to predict. For example, a loop that executes 100 times can have the loop control branch statically predicted with 99% accuracy. By comparison, an inner loop that executes 4 times for each outer loop iteration can be statically predicted with only 75% accuracy, and simpler dynamic schemes may only reach 50% prediction accuracy on such a loop [Yeh & Patt 91].

VSO's loop unrolling removes ALU operations from bitblt's inner loop, but removes no writes. VSO thus makes the inner loop more write-intensive,[3] and performance is limited by the performance of the write-through caches on each platform. The SPARC platform has a relatively shallow write buffer and frequent writes stall the processor; the speedup is thus limited principally by the memory subsystem's bandwidth. The AXP and MIPS platforms have deeper write buffers, which allow writes to "stack up" during write bursts and finish during parts of the loop that are less write-intensive.

[3]For some common cases, every other instruction is a write.

The AXP uses wider words and for bitblt it often performs fewer writes; also, the write-back second-level cache can service writes relatively more quickly (Table 4.2). For these reasons the write buffer depth is less important.

For bitblt, the AXP's 64-bit datapath interacts with loop unrolling in two ways. First, since bitblt is basically a copy operation, wider words reduce the number of iterations needed to copy a given region, and fixed overheads in the outer loop becomes relatively more important, reducing the benefit of full unrolling. Second, since there are fewer iterations of the inner loop, branch prediction is worse, improving the benefit of full unrolling. For bitblt, the first effect is most important, so overall VSO speedups were reduced by the wider datapath.

For associative cache simulation, full unrolling is only partly successful at removing branches. Although the inner loop is small, it contains a tag match conditional which unrolling cannot eliminate. On all platforms, both the static and dynamic code generators lay out the code so that forward tag match conditional branches are usually taken. The AXP statically predicts forward branches as not taken, and so both the static and VSO'd code mispredict tag comparison branches. The MIPS does not use prediction. The SPARC predicts branches as taken, so both the static and VSO'd code predict tag match branches correctly.[4]

5.2.4 Side-Effects of Dynamic Optimization

Recall that VSO works by dynamically specializing general-purpose code around key runtime values. Two effects of specialization are increased code size and code duplication caused by specialization. Both interact with the cache configuration, though in different ways. Thus, although optimizations such as constant folding will, in principle, either improve performance or have no effect, using VSO to enable these optimizations can sometimes cause code expansion and duplication that reduce the code's performance.

Dynamic specialization may either increase or reduce the size of a routine (Table 5.5) and may thus change the working set of the inner loop. For cache–4 and decompression, the VSO'd code is larger than the static code; for bitblt, cache–1 and sorting, the VSO'd code is smaller. The smallest VSO'd fragment is less than 1/3 the size of the static code it replaces, while the largest is over 13 times larger than its counterpart, a code expansion variation of more than 40:1.

Fully-unrolled code may be smaller than the static code that it replaces, because loop-related code is removed, each inner iteration is simplified, and because the unrolling factor is typically small. For example, both bitblt and sorting fully-unroll inner loops, yet the VSO'd code is smaller.[5] Conversely, VSO performs no loop unrolling on decompression, yet decompression shows the most code growth.

[4]It is always possible to convert cycle-free, bounded-length instruction sequences with branches to branch-free sequences, but the branch-free sequences may be longer. VSO makes the iteration length a compile-time constant and thus enables such branch-free transformations [Massalin 87, Granlund & Kenner 92], but their profitability was not studied.

[5]Note that VSO'd bitblt on the AXP shows more percentage shrinkage and is notably smaller overall than the VSO'd bitblt on the other machines. The reason is that the AXP has wider data words, so there are fewer unrolled iterations.

Application	AXP			MIPS			SPARC		
	Static	VSO	exp	Static	VSO	exp	Static	VSO	exp
Bitblt	284	88	0.31	176	156	0.89	176	156	0.89
Cache–1	96	80	0.83	92	80	0.87	92	76	0.83
Cache–4	132	184	1.40	136	140	1.03	132	136	1.03
Decompression	1292	13804	10.7	570	3636	6.38	520	6868	13.2
Sorting	440	152	0.35	404	160	0.40	312	136	0.44

Table 5.5: The size of VSO'd code versus the static code that it replaces.

Absolute sizes, in bytes of a static code fragment and of the VSO'd code that replaces it. Code sizes are given for each application and platform. Note that the code sequences may contain loops and/or conditionals, so not all instructions are executed with equal frequency. The column "exp" is the ratio of the sizes of the two implementations, showing the expansion caused by VSO; expansion less than one shows code that shrunk when VSO'd.

Decompression uses indirect (table lookup) dispatching that maps the input byte to one of several statically-unrolled copy loops. VSO converts an indirect 6-way dispatch operation into a faster direct 256-way dispatch operation. VSO thus generates up to 255 copy loops. On the MIPS and SPARC platforms, the first-level cache is large enough that it holds all of the decompressor's VSO'd inner loop. On the AXP, however, the unrolled loop is over 12KB and the instruction cache is only 8KB. Although the inner loop contains a switch statement and is thus executed sparsely on each iteration, and although some switch cases are more common, the large inner loop still causes self-interference.

Special casing can also replicate code in ways that lead to self-interference, even when the VSO'd code is smaller overall. For bitblt, decompression and sorting, a specialized version of the code is created, used, and then discarded. Thus, there exists only one specialized version at a time. Cache simulation, however, uses specialized cache lookup routines, one for each cache, and so several specialized fragments exist concurrently. For each reference, the simulator executes code to probe the simulated first-level cache.[6] On a miss, the simulator invokes code to probe the simulated second-level cache. Cache–1 simulates direct-mapped first and second level caches. The static code implementation uses a single simulator code fragment to probe both the first-level and second-level caches. The VSO'd version, however, uses different specialized code fragments for probing each cache, and there were conflicts between the code fragments for the first- and second-level caches.

The cache conflicts arise in part because these experiments use an aligning code allocator (§3.3) that places all VSO'd routines at the same cache location as the static code that they replace. Thus, VSO'd routines for the first-level and second-level caches appear at the same cache location as the static code that they replace. If the simulated first-level and second-level caches are both direct-mapped, the VSO'd simulation routines will both appear in same location in the real machine's

[6]For simplicity of explanation, assume that only a single first-level cache is being simulated. That is, assume that either the simulated first-level cache is a unified cache, or only data or only instruction references are being simulated. If split caches were used, VSO would also suffer host conflict misses on every simulated data reference.

instruction cache. Thus, each time the simulator executes code to simulate the second-level cache, the real machine will experience an instruction cache miss, as the first-level simulation code is replaced with the second-level simulation code. Likewise, the immediately following memory reference will call the first-level simulation code, and again the real machine's instruction cache will miss, as the second-level code is replaced with the first-level code.[7]

The effect of VSO, therefore, is to cause two sets of instruction cache misses in the real machine each time there is a miss in the simulated first-level cache. Since misses on the real machine are an order of magnitude more expensive than hits (see Figures 4.2 and 4.3), specialization-induced misses can cause substantial performance loss. Although the aligning allocator causes worst-case conflicts, VSO's code replication would cause some conflicts even with an oracle allocator that could optimize code placement. The problem is similar to a phenomenon observed with inlining, where a commonly-used routine is inlined several places and each one of the sites has poorer locality than the original (commonly-used) routine [Mosberger et al. 95, Mosberger et al. 96, Pittman 95].

The "two version" code interference occurred with the cache–1 simulation but not with the cache–4 simulation. Cache–4 simulates an associative first-level cache and a direct-mapped second level cache; since the static code for direct-mapped and associative simulation are not aligned in the cache,[8] neither are the specialized routines produced by VSO.

The interference observed with cache simulation is partly an artifact of the code placement strategy, which attempts to place all runtime-compiled code versions in the same cache location as their static counterparts. Such code placement is useful for measurement and comparison because it minimizes the effects of caching differences when comparing statically-compiled and VSO'd code (§3.3). Other placement strategies might reduce conflicts (§B.6); without further study, however, the effects are unclear. Note also that, while specialization-induced conflicts are worst on machines with direct-mapped caches, the same effects can appear in machines with associative caches.

5.3 Effects of Varying Dynamic Optimization Level

Faster dynamic compilation directly improves VSO's breakeven point, but very fast compilers achieve their speed, in part, by omitting some optimizations. As a result, VSO compilers that are "too fast" may produce poor code that is unprofitable compared to static code.

[7] The simple handmade dynamic linker (§3.1.7) was sometimes restricted to linking one code fragment at a time and the aligning memory allocator (§3.3) was most simply implemented across multiple platforms by limiting aligned allocation to only a single region at a time. Multi-level cache simulation works around these limitations by using VSO'd code for the top-level cache and general-purpose static code for the lower level caches. Although performance is marginally worse, the simulated first-level cache has a miss rate of less than 10%, and since the static code is only about 10% slower, using static code for the second-level cache should make the performance at most 1% worse. The interference in the real machine's cache, however, is the same: rather than using the specialized code *instead of* the general code, as in the other applications, the cache simulator uses specialized code *in addition to* the general code. There are thus cache conflicts between the code for simulating the first-level cache and the general-purpose code that simulates the second-level cache.

[8] It is simply coincidence that the code for associative and direct-mapped simulation are not aligned. Since both fragments are small compared to the cache size, and since there are only a few such fragments, non-alignment is the likely result. However, if the static code fragments *were* aligned then both the static and VSO'd code would suffer alignment-induced conflicts. Notice that further specialization of static code increases the number of code fragments and thus the likelihood of alignment conflicts, even when using code alignment tools [McFarling 89, Gupta & Chi 90, McFarling 91, Smotherman 94].

In principle, then, a major challenge is determining which optimizations are worth performing at runtime, and which are of marginal benefit and should thus be omitted. The results presented in this section show a perhaps surprising result: that simple optimizations can produce high-quality code and that sophisticated optimizations do not typically further improve the code. The insight is that VSO replaces variables with constants and in doing so replaces uncertain information with definite information. In other words, reducing the complexity of input to the compiler also reduces the benefits of sophisticated analysis.

There are at least two benefits of using simple code generators. First, they are faster, which improves VSO's breakeven point. In particular, since locally-optimal code generators do a good job, it is possible that template compilers can also do a good job, and template compilers are much faster. Second, determining which optimizations to use is a potentially thorny problem, since there are many candidate optimizations and an exponential number of combinations of them. Being able to use a simple code generator thus defuses an explosively complex static optimization problem.

Table 5.6 shows application speedups with tree VSO when using various code generation strategies. The goal of `-O0` optimization is "to produce code quickly" [Stallman 93]. Code is generated on a statement-by-statement basis and has many redundant loads and stores; it is thus worse than code produced by a fast, locally-optimal code generator. `-O1` optimization performs common optimizations, such as CSE, jump optimization, register allocation, and delay slot filling. Although `-O1` does not generate code using a locally-optimal code generation algorithm, the output was checked by hand in each case to verify that the results were like that of a locally-optimal code generator with simple, fast register allocation. The exception, decompression, is described below. The `-O2` optimization level is more expensive and performs more sophisticated optimizations, such as strength reduction and induction variable elimination and (in some cases) instruction scheduling.

Intuitively, better optimization should take longer but produce more efficient code. However, all three code generation strategies produced similar code, except that `-O0` had more loads, stores and unfilled branch delay slots. The advanced optimizations of `-O2` were of little use when applied after VSO, because VSO had already applied or obviated them in the application-specific compiler. For example, induction variable elimination is of no use for code in which loops have been fully unrolled.

In many cases, there was no or only trivial difference between the `-O1` and the `-O2` code. For example, sorting on the MIPS under `-O1` had a subtract followed by a register-to-register move, while under `-O2` the subtract placed the result directly in the final register and the move was replaced with a `nop` instruction. As shown in the column *Ratio* `-O2`:`-O1`, turning on additional optimizations sometimes caused a minor performance loss. The execution times vary slightly from run to run due to, for example, unrelated system activity; even where `-O1` and `-O2` produce identical code, the reported times may vary slightly. Cases where the code quality depended substantially on optimization level are discussed below.

For bitblt, associative cache simulation and sorting on the AXP, the `-O2` optimizer replaced dual-issue instruction pairs with two single-issue instructions, leading to minor slowdowns. This is an artifact of both `-O1` "happening" to get a good instruction order and of the `-O2` optimizer having

Application	Platform	Runtime Optimization Level				Ratio
		`-O0`	`-O1`	`-O2`	`-OX`	`-O2:-O1`
Bitblt	AXP	0.25	1.40	1.30		0.93
	MIPS	0.24	2.05	2.01		0.98
	SPARC	0.30	1.14	1.15		1.01
CacheD	AXP	0.69	0.94	0.96		1.02
	MIPS	0.90	1.10	1.02		0.93
	SPARC	0.80	1.06	1.06		unity
Cache–4	AXP	0.73	1.00	1.02		1.02
	MIPS	0.95	1.06	1.07		1.01
	SPARC	0.80	1.10	1.10		unity
Decompression	AXP	0.54	1.00	1.04		1.04
	MIPS	0.69	0.88	0.95		1.08
	SPARC	0.53	0.97	0.97	1.13	N/A
Sorting	AXP	1.20	2.53	2.42		0.96
	MIPS	1.24	1.92	1.86		0.97
	SPARC	0.72	1.40	1.44		1.03

Table 5.6: Asymptotic speedups of the VSO'd code versus level of runtime optimization.

The execution times of the VSO'd code, shown as a ratio compared to the static code execution times, for each architecture and application. The notation `-O`n indicates the level of runtime optimization, with higher n indicating more optimization. `-OX` shows the result of turning off the tail merging optimization; the result is faster but larger code. The column *Ratio* `-O2:-O1` shows the ratio of the speedups of `-O2` to `-O1` or `-OX`; numbers greater than one indicate the `-O2` code is faster, while numbers less than one indicate that the `-O1` code is faster; unity indicates no change, and N/A for decompression on the SPARC indicates that the results are not comparable (see text). The static code optimization is the same as in the preceding experiments.

an incomplete pipeline model. These examples suggest that instruction scheduling is probably a valuable runtime optimization for multiple-issue machines, especially for tree compilers. Since VSO tends to promote variables to constants, there may be enough free registers that simple scheduling algorithms can work well.

For decompression, the code generators often performed an optimization called *tail merging*, which discovers when a collection of basic blocks all end with the same instruction sequence and then rewrites each block to end with a jump to the common sequence. Tail merging produces smaller code that is slower because of the extra jumps. Tail merging is not an optimization performed by locally-optimal code generators, but was applied by `-O1` optimization, because code generators could not be forced to omit it.

In some cases, other optimization levels produced code without tail merging and thus more like that of a locally-optimal generator. For the AXP, `-O2` with the vendor compiler turned off tail merging and produced larger faster code; `-O2` also allocated common constants to registers on a procedure-wide basis. Thus, `-O1` code is slower than code produced by a locally-optimal code

Application	Macro Speedup			Micro Speedup		
	AXP	MIPS	SPARC	AXP	MIPS	SPARC
Cache–1	0.96	1.10	1.06	1.11	1.14	1.44
Cache–4	1.02	1.10	1.10	1.17	1.45	1.90
Sorting	2.53	1.92	1.40	2.65	2.41	2.20

Table 5.7: VSO speedup of core routines.

VSO speedups over static code for each application and machine. Microbenchmark results are not shown for Bitblt or Decompression; see the text for details.

generator, `-O2` is faster. For the MIPS, `-O0` produced code substantially worse than locally-optimal, and all higher levels of optimization performed tail merging, and thus produced code that was slower than that that would be produced by a locally-optimal code generator. For the SPARC, the vendor C compiler at a low level of optimization (noted as `-OX`) avoided tail merging and otherwise produced code like that of a locally-optimal code generator.

Decompression speedups on the AXP were limited in part by the AXP's use of multi-instruction sequences to manipulate bytes. These operations were present in both the static and VSO'd code and reduced the percentage of inner loop code that was optimized away by VSO.

Note that various machine features make cross-machine comparisons difficult. In particular, I/O rates depend on details of the disk, the system bus architecture, sector layout, and so on [Cockcroft 93]. For all programs, input files were placed on local disks that were largely empty (in order to allow good block placement) and the systems were run otherwise unloaded to avoid both CPU and I/O contention. Nonetheless, some programs were sensitive to I/O rates. For example, direct-mapped cache simulation on the SPARC typically spent 30% of its waiting for I/O, so switching to a machine with even slightly faster relative I/O rates or using overlapped I/O[9] would have increased the time spent executing code and thus the relative speedup of VSO.

5.4 Microbenchmarks

This section presents *microbenchmark* results. Microbenchmarks execute just the core code and thus show execution time without the overheads of I/O, cache misses or program fragments that are not optimized by VSO. Therefore, they represent the maximum improvements possible with VSO.

Table 5.7 shows both the overall application speedups from Table 5.3 and the VSO speedups for the inner loop. No results are shown for bitblt or decompression: for bitblt, nearly all time was spent in the VSO'd code, so the microbenchmark speedups are the same as the macro speedups; for decompression, there are many paths through the inner code and the code structure makes it unclear how to build a representative microbenchmark that avoids cache- and I/O-related overheads.

[9]For example, memory-mapped files with sequential-access prefetching. However, using `vadvise` crashed the system.

For sorting, some differences between micro and macro speedup occurred because there were several paths through each code fragment, and speedups depended on which path was executed. For example, there are five paths through the sorting predicate (for the records and sort order used in this study), and SPARC micro speedups ranged from 1.64 to 2.65, depending on the path. Indeed, in sorting and elsewhere, VSO increased the data-dependent variations in execution time, by making fixed-cost parts faster and thus increasing the relative performance impact of data-dependent variations. The microbenchmark results are thus presented as an average of path values, weighted by an expected path frequency. The purpose of the microbenchmarks is to measure improvements in code quality, in isolation from other effects such as cache behavior. In order to avoid cache effects that might reduce performance, microbenchmarks use special datasets; they thus thus use somewhat different average paths than for the macrobenchmarks.

Some variations between micro and macro speedups occur because the relative cost of the non-VSO'd part of the code varies across platforms. For example, the VSO'd inner loop of sorting uses both VSO'd and static code. Some of the static code performs copying; on the SPARC platform, copying has both load conflicts (§5.2.2) and write stalls (§5.2.3), both of which make the static code more of a bottleneck than on other machines, thus limiting the speedup.

5.5 Summary

The results in this chapter demonstrate that VSO can produce code that is faster than even the best possible hand-crafted static assembly code. The results also show a variety of ways in which VSO payoff depends on the application, the dataset, the architecture and the VSO implementation.

The performance of VSO'd code was compared against carefully-constructed static assembly code and the VSO'd code was faster, indicating that VSO exposes optimizations that are unavailable statically. In addition, even tree VSO sometimes produced code faster than the best static assembly, showing that even retargetable code generators can sometimes produce code better than the best hand-crafted static code.

The runtime compiler performance sometimes affects VSO's profitability. For example, when VSO is implemented with a template compiler, the runtime compile times are much lower and, consequently, the breakeven points are achieved much sooner than with the tree compilers. In some cases this difference helps to make VSO applicable to a much wider variety of inputs; in others, tree VSO more than pays off for all reasonable inputs, so the template compiler is of no additional benefit.

The benefit of VSO also varies widely across architectures. For example, sorting showed asymptotic speedups of 1.4 on the SPARC platform, 1.8 on the MIPS platform, and 2.5 on the AXP platform. Furthermore, there was no single machine that showed consistently good or bad VSO speedups. Thus, while VSO profitability depends on the architecture, these results do not point to a "VSO architecture", because architectural features that VSO exploits for one application are of no particular use with another application.

In principle, a major challenge is determining which optimizations are worth performing at runtime, and which are of marginal benefit and should thus be omitted. The results presented in this chapter show a perhaps surprising result: that simple optimizations can produce high-quality code and that sophisticated optimizations do not typically further improve the code. The insight is that VSO replaces variables with constants; in doing so, it replaces uncertain information with definite information, and thus allows simple analysis to produce good optimizations.

Chapter 6

Event Count Studies

"This is a most unusual event."

–Opening speaker at most any event.

This chapter reports on how VSO's optimizations and speedups are affected by particular hardware features. Event counts, such as the number of memory, ALU and branch instructions, were used to gain insight into architectural factors that influenced VSO's speedups.

Event traces are particularly useful because they can drive simulation studies that isolate and evaluate the effects of specific hardware features. Simulation-driven studies can vary just one component while leaving the others constant. In contrast, comparing program execution times on two platforms is complex because a pair of machines will have several differences that can together affect overall execution time. For example, the MIPS and SPARC platforms are generally similar in that they are both workstation-class RISC-based machines, but they nonetheless have different instruction sets, relative processor and memory speed, memory hierarchy and I/O performance.

This chapter's results are based on traces of SPARC versions of the applications, because only the SPARC tools could trace dynamically-changing code (§4.2).

Section 6.1 compares instruction counts and execution times and describes how VSO speedups were limited by increases in the average cost to execute an instruction. Section 6.2 reports on how VSO changes the kinds of instructions executed by each program. Section 6.3 shows which factors cause the increase in the average cost per instruction and also how those costs are affected by some current architectural trends.

6.1 Instruction Counts and Execution Time

VSO reduced the number of instructions executed by the programs. However, performance did not improve proportionally, because VSO also increased the average time needed to execute an instruction. Table 6.1 shows the total number of instructions executed by each version of each application, the mean *cycles per instruction* or *CPI* and the overall application speedup. The CPI was computed by multiplying the execution time and the processor speed, then dividing by the instruction counts.

Table 6.1: Number and cost of instructions executed by the core of each application.

Application	Instructions			CPI			Speedup
	Static	VSO	Ratio	Static	VSO	Increase	
Bitblt	2470	1440	1.71	1.27	1.90	1.51	1.14
Cache–1	1070	1010	1.06	3.44	3.44	1.00	1.06
Cache–4	2100	1800	1.16	2.44	2.59	1.06	1.10
Decompression	164	152	1.08	2.03	1.94	0.96	1.13
Sorting	890	525	1.70	1.56	1.89	1.21	1.40

The number of instructions executed by the core of each application and the measured CPI. *Instructions* shows millions of instructions executed in the inner loop for long runs of each application, with instructions for program initialization and cleanup omitted. *Ratio* is the first column divided by the second, which is the inverse of VSO's reduction in instruction counts; the inverse is used to show the correspondence between instruction counts and asymptotic speedup. *CPI* shows the average number of cycles taken to execute an instruction; reported times include cache miss and I/O overheads. *Increase* is the increase in CPI using VSO. *Speedup* is the measured asymptotic performance from Table 5.3.

For all applications, the static versions executed more instructions than the VSO'd versions. For cache simulation, speedups closely matched the reduction in the number of instructions. However, for bitblt and sorting, measured speedups were not as good as might be predicted by the drop in instruction counts, because the average cost per instruction increased. Conversely, for decompression, the speedup was actually greater than the number of instructions that were removed, because the average cost per instruction went down.

The cost per instruction depends on a variety of factors. First, VSO does not remove fixed overheads such as I/O, so the fixed overheads are amortized over a smaller number of instructions, increasing the average cost. Second, VSO changes the kinds of instructions that are executed. For example, load instructions typically take more than one cycle, so removing many load instructions, as was done in decompression, tends to decrease the average cost per instruction. However, leaving the load instructions and removing relatively faster integer arithmetic instructions, as in bitblt, tends to increase the average cost per instruction. The following sections explore these effects.

6.2 Instruction Mix

This section investigates how VSO changed the *instruction mix* executed by each program. The instruction mix is the dynamically-weighted percentage of instructions executed from each of a set of categories, such as load, store, and so on. Table 6.2 shows how instructions are categorized and Table 6.3 shows the metrics used to compare the static and VSO versions of the programs. Table 6.4 shows results for each program and Table 6.5 shows averages over all programs.[1] The results show

[1]The averages are unweighted averages of the percentages. Since the percentages are normalized counts, the effect is to weight the averages so that the results are not biased by the number of instructions executed by each workload.

Table 6.2: How instructions in Tables 6.4 and 6.5 were categorized.

Category	Representing
`alu`	Arithmetic operations, such as add, subtract, shift, and so on.
`alucc`	Same as `alu`, but also sets the condition codes. Instructions in `alucc` are usually paired with conditional branches (`cbr`).
`cbr`	Conditional branches.
`jmp`	Unconditional branches, including procedure and system calls.
`ld`	Loads.
`st`	Stores.

Table 6.3: Categories and metrics for instruction mix comparisons in Tables 6.4 and 6.5.

Column	Representing
Category	The instructions categories shown in Table 6.2.
Static Percent	The dynamically-weighted instruction mix for programs compiled statically.
VSO Percent	The dynamically-weighted instruction mix for programs using VSO.
Change in Mix	The ratio of the static and VSO percentages, computed by dividing *VSO Percent* by *Static Percent*. Numbers less than 1.0 indicate that the VSO'd version executed fewer of its instructions from the given category. Since VSO reduces the total number of instructions executed by a program, VSO may eliminate instructions from a given category (*Percent of Instructions Removed*), yet still increase the fraction of such instructions executed by a program.
Percent of Instructions Removed	Of all instructions removed by VSO, the percentage removed from each category. For example, "50" for `alu` indicates that half of instructions removed by VSO were from the `alu` category.

that VSO removed different components of the instruction mix for each application, but that, on average, VSO had a relatively small effect on the overall instruction mix.

ALU Operations Through constant propagation, VSO removed a large number of ALU operations. The total number of ALU operations was so great, however, that ALU operations were removed less than proportionally. VSO thus tended to increase the percentage of ALU operations executed by the applications.

ALU operations marked as `alucc` set condition codes and typically occurred in tandem with conditional branches. As shown in Figure 6.1, VSO may remove the ALU operation, the branch, or both. Although Tables 6.4 and 6.5 show equal numbers of such instructions being eliminated, the profiles do not indicate whether VSO merely eliminates the need for the condition codes, as in

Table 6.4: Measured instruction mix for each application (continued in Table 6.5).

Application	Category	Static Percent	VSO Percent	Change in Mix	Percent of Instructions Removed
Bitblt	alu	61.3	61.0	1.00	61.7
	alucc	8.8	2.5	0.28	17.0
	cbr	8.8	2.5	0.28	17.0
	jmp	0.2	0.2	0.95	0.2
	ld	12.0	18.6	1.55	3.6
	st	8.8	15.2	1.73	0.6
Cache–1	alu	47.9	51.0	1.07	0.0
	alucc	6.1	6.5	1.07	0.0
	cbr	5.8	6.2	1.07	0.0
	jmp	6.8	7.3	1.07	0.0
	ld	27.0	22.2	0.82	100.0
	st	6.4	6.8	1.07	0.0
Cache–4	alu	43.5	46.2	1.06	28.9
	alucc	10.8	9.9	0.92	15.4
	cbr	9.8	8.8	0.89	15.4
	jmp	7.2	7.4	1.02	6.4
	ld	23.3	21.3	0.92	33.8
	st	5.4	6.4	1.19	0.0
Decompression	alu	32.5	38.2	1.17	9.3
	alucc	16.5	16.3	0.99	17.4
	cbr	16.5	16.2	0.99	17.4
	jmp	7.7	10.1	1.31	-2.3
	ld	20.6	11.4	0.56	58.2
	st	6.3	7.8	1.24	0.0
Sorting	alu	36.3	39.2	1.08	32.2
	alucc	17.9	15.5	0.87	21.1
	cbr	18.7	15.5	0.83	23.0
	jmp	5.2	7.7	1.49	1.7
	ld	18.4	15.7	0.86	22.0
	st	3.7	6.3	1.73	0.0

Figure 6.1(b), or whether VSO removes the ALU operations altogether, as in Figure 6.1(c). Hand-examination of the applications indicated that VSO typically removed both the ALU operation and the conditional branch, as in Figure 6.1(c).

Branches VSO enabled loop unrolling and dynamic dead code elimination, which eliminated both conditional and unconditional branches. For most applications, 15-20% of the instructions eliminated by VSO were branch instructions. For bitblt, cache–4 and sorting, VSO reduced the overall percentage of branches in the VSO'd instruction mix; however, for cache–1 and decompression, it increased the fraction of branches in the VSO'd mix. In addition, VSO turned some conditional

Table 6.5: Measured instruction mix for each application (continued from Table 6.4).

Application	Category	Static percent	VSO percent	Change in mix	Percent of Instructions Removed
Composite	alu	44.3	47.1	1.08	26.4
	alucc	12.0	10.1	0.82	14.2
	cbr	11.9	9.9	0.81	14.6
	jmp	5.4	6.5	1.17	1.2
	ld	20.3	17.9	0.94	43.5
	st	6.1	8.5	1.39	0.1

The columns are described in Table 6.3. The sum of the entries in some columns does not total 100% due to rounding; however, all instructions are included.

```
subcc  r4,#1, r5        sub  #N,#1, r5      op  #N-1, ...      ba  L15
bne    L15         ⇒    op   r5, ...
op     r5, ...
```

(a) (b) (d) (d)

Figure 6.1: A static code conditional branch and three VSO'd equivalents.

A static code conditional branch, (a), and three VSO'd equivalents that can arise when the value in r4 is promoted to a constant. In (a), subcc sets condition codes used by the conditional branch, bne, and also produces a result in r5. If the branch is not taken, the fall-through case, "op ..." is executed. In (b), VSO promotes r4 to a constant other than one and thus and the bne is eliminated; however, the value in r5 is also used in a subsequent instruction and so the subcc is converted into a sub but is not eliminated. In (c), the result of the sub can be encoded in the op instruction's immediate field, so the sub is eliminated. In (d), VSO promotes r4 to the constant one and thus the bne is replaced by an unconditional branch.

branches in decompression into unconditional indirect branches; these can be harder to predict, potentially increasing the CPI on those branches [Popescu et al. 91, Yeh & Patt 91].

Loads VSO eliminates loads by promoting variables to constants and embedding the constants in instruction immediates (§5.2.2). For most applications, VSO removed load instructions in greater proportion than they appeared in the static program. For example, loads were about a quarter of the instructions executed by cache–4, but about a third of the instructions removed by VSO. As a result, VSO made loads a smaller component of the final instruction mix in most applications. Bitblt is an exception: VSO promoted variables that were otherwise held in registers, so few of the instructions removed from bitblt were load instructions.

Stores VSO rarely removes store instructions. Although VSO promotes variables to constants and thus improves the quality of information available to the optimizer, the constant values are quickly combined with variables, which tends to localize the improved information. Thus, most computations eliminated by VSO are those that produce short-lived results. Most short-lived results are kept in registers, though, and even static code has few stores of short-lived results. For example, the associative search loop in cache–4 used computations to control iteration. Although VSO unrolled the loop and thus removed those computations, their results were already kept in registers for their whole lifetime; consequently, there were no stores to remove. Conversely, the simulated cache tags were live over an indefinite number of iterations of the outer loop (a potentially infinite lifetime), and using VSO to compute the cache tags more efficiently did not eliminate the need to save them.

Table 6.5 shows unweighted averages of the values in Table 6.4. Comparing the columns *Static percent* and *VSO percent*, the results show that, on average, VSO tended to slightly decrease the percentage of loads and conditional branches, slightly increase the percentage of stores and ALU operations, but overall have only minor effects on the instruction mix.

The averages are potentially misleading since they obscure variations between applications. For example, on average, loads and ALU operations were removed in roughly equal numbers (*Percent of Instructions Removed*). It is thus intuitively appealing to imagine that each time a variable was promoted to a constant, VSO removed both a load and an ALU operation that depended on that value. In practice, however, VSO had this effect only for cache–4 and sorting. For bitblt, nearly all of the eliminated instructions were ALU operations or branches, and for cache–1 and decompression relatively few ALU operations were removed. Likewise, VSO eliminated many loads on average, but for some applications the fraction of loads went down substantially (cache–1, decompression), for others the fraction was relatively unchanged (cache–4, sorting), and one had the fraction increased (bitblt).

The profiles in this section also do not capture some of VSO's effects. For example, in decompression, VSO enabled constant propagation and removed several ALU operations from the inner loop's dispatch operation. VSO also enabled load elimination and replaced loads with ALU operations (set immediate) in copy loops. Thus, although the two optimizations have large effects at the basic block level, they tended to cancel each other overall.

6.3 CPI

As noted above (§6.1), VSO tends to change the CPI, so that application speedups do not always match the reduction in the number of instructions executed. Studying the factors that affect the CPI provides a mechanism for understanding the role played by processor architecture. The results show that write stalls and fixed overheads, such as data cache misses and time spent waiting for I/O, were the major causes. In contrast, effects from unified cache conflicts, instruction cache misses, and mispredicted branches were all much smaller. This section also shows that VSO also would have only modest effects as instruction caches got smaller and as branch penalties increased.

Table 6.6: CPI component costs for each application, as reported in Tables 6.7 and 6.8.

Column	CPI Contribution From
U cont.	Load instructions that hit in the cache but contend for access to the unified cache and thus delay issue of subsequent instructions (§5.2.2).
I miss	Instruction cache misses.
D miss	Data cache misses.
Mis. br.	Mispredicted branches.
Write stall	Rapid (back-to-back) writes, which can overflow the write buffer and cause stalls. The CPI from write stalls depends on the timing of the writes (§4.2.2), but event simulation gives only the sequence of writes, not their times. Stall costs are thus approximated two ways. First, assuming that writes are distributed evenly throughout the application and thus conflict with each other minimally (column *Stall Est.* is *Low*), and second, assuming that writes are all back-to-back and thus essentially all writes stall (*Stall Est.* is *High*). Both are unrealistic extremes; for lack of a timing-accurate simulation, averaging the high and low values probably provides a reasonable estimate.
Est. CPI	Estimated CPI derived using simulation. The estimated CPI is the sum of the above columns plus a base cost of one cycle per instruction.
Unac. CPI	Difference between *Est. CPI* and the measured CPI.
Meas. CPI	Measured CPI.

6.3.1 Accounting For The Observed CPI

Each version of each application was simulated as described in Section 4.2.2, in order to compute low-level component costs that contributed to the CPI. In particular, cache contention, cache misses and mispredicted branches were simulated using fixed costs per event. The effects of system-induced cache perturbation were approximated by simulating caches that were flushed at each system call (see Section 4.2.2). The effects of write buffer stalls were approximated by computing the stall costs based on various distributions of store instructions. The effects of system activity, including both instructions executed in the operating system and time spent blocked on I/O, were approximated by subtracting the estimated cost (the sum of all estimated CPI sources) from the measured cost (Table 6.1).

Table 6.7 shows the estimated and measured CPI for each version of each application, and Table 6.8 shows the unweighted averages across all applications. The columns *Flush* and *Stall Est.* show the simulation methodology used for each application. The remaining columns show the CPI contribution from each source; Table 6.6 describes the categories. The remainder of this section analyzes how VSO affected the CPI contribution from each of the sources.

Unified Cache Contention The simulated CPI from unified contention was typically 0.2 CPI. That was as much as 10% of the total CPI for an application. From an architectural standpoint,

Table 6.7: Estimated CPI for each application, continued in Table 6.8.

App.	Impl.	Flush	Stall Est.	U cont.	I miss	D miss	Mis. br.	Write stall	Est. CPI	Unac. CPI	Meas. CPI
Bitblt	static	N	Low	0.12	0.00	0.00	0.03	0.00	1.15	0.12	1.27
			High	0.12	0.00	0.00	0.03	0.88	2.03	-0.76	1.27
		Y	Low	0.12	0.00	0.00	0.03	0.00	1.15	0.12	1.27
			High	0.12	0.00	0.00	0.03	0.88	2.03	-0.76	1.27
	VSO	N	Low	0.19	0.00	0.00	0.00	0.34	1.53	0.38	1.91
			High	0.19	0.00	0.00	0.00	1.53	2.71	-0.80	1.91
		Y	Low	0.19	0.00	0.00	0.00	0.34	1.53	0.38	1.91
			High	0.19	0.00	0.00	0.00	1.53	2.71	-0.80	1.91
Cache–1	static	N	Low	0.27	0.01	0.06	0.01	0.00	1.36	2.08	3.44
			High	0.27	0.01	0.06	0.01	0.64	1.99	1.45	3.44
		Y	Low	0.27	0.02	0.23	0.01	0.00	1.53	1.91	3.44
			High	0.27	0.02	0.23	0.01	0.64	2.16	1.28	3.44
	VSO	N	Low	0.22	0.30	0.07	0.01	0.00	1.60	1.84	3.44
			High	0.22	0.30	0.07	0.01	0.68	2.27	1.17	3.44
		Y	Low	0.22	0.30	0.24	0.01	0.00	1.78	1.66	3.44
			High	0.22	0.30	0.24	0.01	0.68	2.46	0.98	3.44
Cache–4	static	N	Low	0.23	0.06	0.04	0.04	0.00	1.37	1.07	2.44
			High	0.23	0.06	0.04	0.04	0.53	1.90	0.54	2.44
		Y	Low	0.23	0.07	0.13	0.04	0.00	1.47	0.97	2.44
			High	0.23	0.07	0.13	0.04	0.53	2.00	0.44	2.44
	VSO	N	Low	0.23	0.07	0.04	0.03	0.00	1.38	1.21	2.59
			High	0.23	0.07	0.04	0.03	0.62	2.00	0.59	2.59
		Y	Low	0.23	0.08	0.15	0.03	0.00	1.49	1.10	2.59
			High	0.23	0.08	0.15	0.03	0.62	2.11	0.48	2.59
Decom- pression	static	N	Low	0.21	0.00	0.00	0.08	0.00	1.30	0.73	2.03
			High	0.21	0.00	0.00	0.08	0.60	1.90	0.13	2.03
		Y	Low	0.21	0.01	0.11	0.08	0.00	1.42	0.61	2.03
			High	0.21	0.01	0.11	0.08	0.60	2.02	0.01	2.03
	VSO	N	Low	0.19	0.01	0.01	0.09	0.00	1.29	0.65	1.94
			High	0.19	0.01	0.01	0.09	0.65	1.94	0.00	1.94
		Y	Low	0.19	0.04	0.11	0.09	0.00	1.43	0.51	1.94
			High	0.19	0.04	0.11	0.09	0.65	2.08	-0.14	1.94
Sorting	static	N	Low	0.21	0.01	0.26	0.08	0.00	1.56	0.00	1.56
			High	0.21	0.01	0.26	0.08	0.34	1.90	-0.33	1.56
		Y	Low	0.21	0.01	0.26	0.08	0.00	1.56	0.00	1.56
			High	0.21	0.01	0.26	0.08	0.34	1.90	-0.33	1.56
	VSO	N	Low	0.18	0.01	0.43	0.07	0.00	1.69	0.20	1.89
			High	0.18	0.01	0.43	0.07	0.57	2.26	-0.37	1.89
		Y	Low	0.18	0.01	0.43	0.07	0.00	1.69	0.20	1.89
			High	0.18	0.01	0.43	0.07	0.57	2.26	-0.37	1.89

Table 6.8: Estimated CPI for each application, continued from Table 6.8.

App.	Impl.	Flush	Stall Est.	U cont.	I miss	D miss	Mis. br.	Write stall	Est. CPI	Unac. CPI	Meas. CPI
Com-posite	static	N	Low	0.21	0.02	0.07	0.05	0.00	1.35	0.80	2.15
			High	0.21	0.02	0.07	0.05	0.60	1.95	0.20	2.15
		Y	Low	0.21	0.02	0.15	0.05	0.00	1.43	0.72	2.15
			High	0.21	0.02	0.15	0.05	0.60	2.02	0.13	2.15
	VSO	N	Low	0.20	0.09	0.09	0.04	0.00	1.43	0.92	2.35
			High	0.20	0.09	0.09	0.04	0.77	2.20	0.15	2.35
		Y	Low	0.20	0.10	0.18	0.04	0.00	1.53	0.82	2.35
			High	0.20	0.10	0.18	0.04	0.77	2.30	0.05	2.35

however, unified cache contention may be eliminated by using split instruction and data caches.[2] Also, the CPI contribution from unified cache contention was affected only slightly by VSO. Thus, the effects of unified caches are largely irrelevant except for attributing causes of the measured CPI.[3]

Instruction Cache Misses Instruction cache miss costs were generally small except for cache simulation, and only cache–1 showed a substantial CPI increase from VSO. As described in Section 5.2.4, the dramatic increase for cache–1 occurred because VSO generated multiple specialized versions of the core code and the VSO prototype aligned all of the versions, thus causing conflict misses.

Data Cache Misses Data cache miss costs varied widely across applications. Predictably, bitblt, which performed in-core copying of small regions, showed good cache hit ratios. In contrast, sorting had fewer loads than most other applications (Table 6.4), but it accessed a larger data set and the access patterns were less regular, so prefetching due to long cache lines [Smith 82] tended to be less effective. For sorting, VSO removed references to dispatch control variables; the variables were used on every predicate invocation and thus had good locality. The remaining loads referenced fields in records and had poor locality. Thus, the overall cache miss *costs* tended to remain about the same, but with VSO they were amortized over a smaller number of instructions, increasing the average cost.[4]

VSO also tended to increase the data miss CPI for cache simulation and data decompression, because I/O-induced miss costs were averaged over shorter program runs. The inner loop of these

[2]Coherency of split caches is then needed, but the added overhead would not materially affect the cost of VSO for any of these applications or platforms.

[3]Note that even with split caches, load elimination can still be useful, because it prefetches data with the instructions and because it eliminates data references and thus reduces data cache contention.

[4]The behavior for sorting fits with intuition, which says that if a particular variable is used often enough to be profitably inlined into a particular code fragment, then that variable probably already had good cache locality. A given value may be inlined into several fragments, but the overall body of code must still have good enough locality to pay for the cost of dynamic optimization and compilation. This intuition is less likely to be true for applications that span multiple protection domains [Torrellas et al. 92, Mosberger et al. 95, Mosberger et al. 96].

applications typically stepped through an input buffer, reading each element once. When the application reached the end of the buffer, it called the operating system to refill the buffer; when I/O completed, the application then repeated its walk through the buffer. The I/O-induced operating system effects were difficult to measure directly (§4.2.2), so each operating system call was simulated two distinct ways: first, by having the caches[5] unaffected by system calls (noted with column *Flush* as *N*); and second, by having system calls completely flush the caches (*Flush* is *Y*), which simulated the effects of refilling the input buffers.

Since bitblt and sorting made few system calls, they were mostly unaffected by flushing. Cache simulation and decompression had low miss rates without flushing, which shows there were few conflict-induced misses. The miss rates were much higher with flushing, due to simulated I/O. The application buffer sizes were the same order of magnitude as the cache size, so full flushing closely modeled flushing done by the real system, whether or not the real system performed full flushing.

Mispredicted Branches Mispredicted branches contributed little overhead, mostly because the host machine's misprediction penalty was small (one cycle). The effects of larger branch prediction penalties and the effects of varying prediction strategies are discussed below (§6.3.2).

Write Stalls The platforms used in this study all use write-through first-level data caches that buffer writes so that execution can overlap with writes to the second-level caches. When rapid (back-to-back) writes overflow the buffer, the later writes stall until a prior write completes and makes space in the write buffer. Write stalls can be particularly expensive compared to cache misses because misses are typically amortized over several loads, whereas write stalls with write-through caches may affect every write in an inner loop. For example, if writes take ten cycles to complete, an inner loop containing two store instructions can never execute faster than twenty cycles per iteration. The SPARC platform used here (see Chapter 4.3) is especially sensitive to write stalls, because the write buffer is small compared to the frequency of writes; overflows are thus more likely than with the other platforms.

VSO tended to exacerbate write stalls because it increased both the average rate of writes and the burstiness of writes. The increase in burstiness is particularly important, as stalls are most often caused by write bursts. An extreme example is bitblt. The static version's inner loop was already write intensive and VSO eliminated branches and ALU operations, so the write stall costs were averaged over fewer instructions. The other platforms have deeper write buffers and relatively faster stores, and so bitblt was less hampered by writes and the speedups were better, more closely matching the reductions in instruction counts. In contrast, the other applications executed a smaller percentage of writes (Table 6.7); there, VSO had less effect on the write distribution and thus added less stall-induced overhead.

The estimated CPI was derived by adding together the estimated costs and a base cost of one

[5] "Caches" here refers to the caches of the machine running the applications; one of the applications happens to be a cache simulator, but that's not the cache of interest here.

cycle per instruction. For bitblt and sorting, which are not I/O intensive, averaging the write stall times gave an estimated CPI that was close to the measured CPI, indicating that the simulation model was reasonable. For the others, the estimated CPI was much less than the measured CPI. However, attributing the remaining time to I/O is consistent with profiles of the programs, which show that a large fraction of each program's execution time is spent waiting for I/O.[6]

It is clear that the simulation model is inaccurate in at least one way: measuring decompression showed that the CPI decreased markedly (Table 6.1), while simulation showed the CPI as nearly unchanged. Likewise, the additional contribution from I/O should have gone up, because fixed I/O overheads were amortized over shorter execution times. It seems likely that the problem is instruction-level parallelism that is not modeled [Bedichek 95b]; in particular, on the real machine, reducing the number of loads allowed buffered stores to complete more quickly.

To summarize, VSO can substantially increase the CPI of applications, for two major reasons. First, VSO tends to increase the density and frequency of write bursts and thus increases the cost due to write stalls. Second, VSO does not remove overheads such as I/O and data cache misses. Those overheads are amortized over shorter program runs and are thus more significant. Although these results may seem obvious in retrospect, it is interesting the degree to which other factors improved the execution time without substantially affecting the CPI.

Finally, event-driven simulation did not measure the effects that VSO had on superscalar issue. As described in Section 5.3, such effects are clearly important, but simulation of a processor's superscalar issue and pipeline were beyond the scope of this study.

6.3.2 Branch Misprediction CPI

Section 6.3 showed that branches contributed little to the overall CPI; however, the costs reflect a machine with a small misprediction penalty. This section analyzes VSO's branch misprediction costs on processors with large branch misprediction penalties. The results show that, while VSO removes some branch misprediction costs, the reduction is partly offset by reductions in total instructions, so that the average CPI is only slightly improved. The exception is bitblt, where the inner loop contains only branches that control definite iteration; here, VSO dramatically lowers the branch misprediction costs.

This section evaluates the effects of VSO on branch misprediction penalties using two simple, static branch prediction policies. The policy *BTFNT* is *Backwards Taken, Forwards Not Taken.*[7] The policy *SPARC* predicts branches as taken. This section considers only the costs of misprediction; it does not consider unconditional and indirect branches, nor correctly-predicted static code branches that are removed by VSO.

This section evaluates misprediction *costs* rather than the misprediction *rates*, because the contribution of branches to the overall execution time depends on both the number of branches and the

[6]Profiling shows, e.g., 30% of cache–1's time is spent waiting for I/O, versus about 45% in Table 6.7. However, profiling increases the execution time and thus tends to underestimate the percentage of time spent in I/O.

[7]Although BTFNT is like the static part of the AXP's branch prediction strategy [Sites 92], actual AXP implementations use both static and dynamic prediction [Gwennap 95b]. Thus, the costs listed here are not the same as those paid by AXP versions of the programs that are measured in Chapter 5.

overall misprediction rate. For example, VSO removed 70% of the branches in bitblt. It happens that VSO improved the average prediction rate, but even if the prediction rate were unchanged, most of the costs of mispredicted branches would have been eliminated, simply by eliminating most of the branches.

Table 6.9 shows the CPI contribution of branches in each implementation of each application. The branch prediction penalty ranged from one, typical of older implementations of today's processors, up to 16, representative of processors that are becoming available at the time of writing [Gwennap 95a].

The results show that VSO was moderately successful at reducing the CPI contribution of unpredictable branches, even though the VSO'd versions executed fewer instructions and thus the remaining misprediction costs were amortized over fewer instructions. The only application that showed dramatic improvements was bitblt, which has an inner loop that was fully unrolled by VSO, thus eliminating the unpredictable last-iteration branches (§5.2.3). In contrast, the inner loops of cache–4 and sorting were also unrolled, but unrolling did not remove unpredictable branches used for conditional loop exit. For cache–1, no branches were removed, while for decompression a fairly predictable branch was removed.

6.3.3 Instruction Cache CPI

Section 6.3.1 notes that instruction cache misses are a small component of the overall execution cost. However, the base system used a 64KB cache, which is large compared to today's first-level caches. This section analyzes how VSO affects miss costs for smaller caches.

Table 6.10 shows that, with decreasing cache size, cache–1 implemented with VSO had consistently high miss costs. The overhead arose because, for this application, VSO created multiple conflicting versions of the inner loop (§5.2.4). However, the added cost of VSO did not change with decreasing cache size. Although the cache–1 conflicts disappeared with the 128KB cache, the change is an artifact of the VSO prototype's aligning code allocator, which positioned code for the real machine's 64KB cache (and thus for smaller cache sizes), but which did not position it in the larger simulated cache.[8] Cache–4's miss costs also dropped when using the larger cache, but both the static and VSO'd versions experienced improvement; here, the drop is probably due to the elimination of real conflicts.

For decompression, the VSO'd SPARC code size was about 6KB, which causes minor increases in conflict rates for the 8KB simulated cache. For the 2KB cache, the static code miss rates rise substantially and are worse than those of the larger VSO'd code; this anomaly is probably because the VSO'd code, though larger, is executed sparsely and, by coincidence, heavily-used paths of the VSO'd code are at different addresses than the corresponding parts of the static code. Note that the AXP's VSO'd code was about 12KB and thus the AXP's cache behavior cannot be extrapolated

[8]The drop occurs because the simulated cache is larger than the 64KB real cache for which the positioning code allocator was configured. Allocators are configured separately for each platform, so the execution times of the previous chapter are measured consistently. For example, the SPARC allocator was configured only for a 64KB cache, but for the AXP, the allocator was configured to preserve code position even in the AXP's larger 512KB second-level cache.

Table 6.9: CPI contribution for each version of each application.

Application	Impl.	Policy	Branch Penalty (cycles) 1	2	4	8	16
Bitblt	static	SPARC	0.03	0.06	0.12	0.23	0.46
		BTFNT	0.01	0.03	0.06	0.11	0.23
	VSO	SPARC	0.00	0.00	0.00	0.01	0.01
		BTFNT	0.00	0.00	0.00	0.01	0.01
Cache–1	static	SPARC	0.01	0.02	0.03	0.06	0.12
		BTFNT	0.03	0.06	0.13	0.25	0.50
	VSO	SPARC	0.01	0.02	0.03	0.07	0.13
		BTFNT	0.03	0.07	0.13	0.27	0.54
Cache–4	static	SPARC	0.04	0.07	0.15	0.30	0.60
		BTFNT	0.05	0.10	0.20	0.40	0.80
	VSO	SPARC	0.03	0.07	0.14	0.28	0.56
		BTFNT	0.04	0.08	0.17	0.34	0.68
Decompression	static	SPARC	0.08	0.17	0.34	0.67	1.35
		BTFNT	0.08	0.16	0.32	0.64	1.27
	VSO	SPARC	0.09	0.18	0.37	0.73	1.46
		BTFNT	0.07	0.15	0.29	0.58	1.17
Sorting	static	SPARC	0.08	0.16	0.32	0.64	1.28
		BTFNT	0.08	0.17	0.33	0.66	1.33
	VSO	SPARC	0.07	0.14	0.28	0.55	1.10
		BTFNT	0.08	0.15	0.30	0.60	1.20
Composite	static	SPARC	0.05	0.10	0.19	0.38	0.76
		BTFNT	0.05	0.10	0.21	0.41	0.83
	VSO	SPARC	0.04	0.08	0.17	0.34	0.67
		BTFNT	0.05	0.09	0.19	0.37	0.74

CPI contribution for each version of each application, using various prediction policies (*Policy*) and misprediction costs (*Branch Penalty*). The columns under *Branch Prediction* show the CPI that is due to mispredicted branches. Each column is simply a multiple of the 1-cycle cost; numbers vary slightly due to rounding. To compute the per-branch penalty, divide listed entries by the corresponding `cbr` entries in Table 6.4. For example, the average misprediction cost per branch for the sorting application, compiled statically, using a one-cycle penalty with SPARC prediction is $\frac{0.08}{0.187} = 0.43$, where 0.08 is the entry from the appropriate entry of this table and 0.187 is the decimal form of 18.7% from the appropriate entry of Table 6.4.

directly from the results here.

Finally, Table 6.10 shows that, overall, VSO had little effect on instruction cache behavior. With decreasing cache sizes, VSO increased cache miss costs only for particularly small caches. However, with small caches, even the static code was victim to circumstantial effects of code placement.

Table 6.10: CPI contribution for various sizes of instruction caches.

Application	Impl.	Flush	2KB	8KB	32KB	64KB	64KB U	128KB
Bitblt	static	N	0.01	0.00	0.00	0.00	0.00	0.00
		Y	0.01	0.00	0.00	0.00	0.00	0.00
	VSO	N	0.00	0.00	0.00	0.00	0.00	0.00
		Y	0.00	0.00	0.00	0.00	0.00	0.00
Cache–1	static	N	0.55	0.00	0.00	0.00	0.01	0.00
		Y	0.56	0.01	0.01	0.01	0.02	0.01
	VSO	N	0.81	0.28	0.28	0.28	0.30	0.00
		Y	0.82	0.29	0.29	0.29	0.30	0.01
Cache–4	static	N	1.86	0.05	0.05	0.05	0.06	0.00
		Y	1.87	0.06	0.06	0.06	0.07	0.01
	VSO	N	2.20	0.06	0.06	0.06	0.07	0.00
		Y	2.21	0.07	0.07	0.07	0.08	0.01
Decompression	static	N	0.00	0.00	0.00	0.00	0.00	0.00
		Y	0.01	0.01	0.01	0.01	0.01	0.01
	VSO	N	0.27	0.01	0.00	0.00	0.01	0.00
		Y	0.29	0.04	0.04	0.04	0.04	0.04
Sorting	static	N	0.28	0.11	0.00	0.00	0.01	0.00
		Y	0.28	0.11	0.00	0.00	0.01	0.00
	VSO	N	0.19	0.19	0.00	0.00	0.01	0.00
		Y	0.19	0.19	0.00	0.00	0.01	0.00
Composite	static	N	0.54	0.03	0.01	0.01	0.02	0.00
		Y	0.55	0.04	0.02	0.02	0.02	0.01
	VSO	N	0.78	0.11	0.08	0.08	0.09	0.00
		Y	0.79	0.12	0.10	0.10	0.10	0.01

All caches have a 16-byte lines. The column *64KB U* is the unified cache used for Tables 6.4 and 6.5. All other entries are for instruction-only caches.

6.4 Summary

This chapter presented a detailed component analysis of the effects of applying VSO. First, VSO removed up to 40% of the instructions from applications. The instructions that occurred frequently in the instruction mix were mostly removed in proportion to their occurrence, and so VSO had relatively little effect on the overall mix. Second, VSO's speedups were limited by fixed overheads that got amortized over fewer instructions. The primary overheads were cache misses for programs with poor data locality, I/O overhead for those that are I/O-intensive, and limited write-through bandwidth on machines with write-through caches. Third, VSO removed unpredictable branches but also removed other instructions, so even with high branch penalties, VSO did not tend to reduce the average cost per instruction. Finally, since VSO generates new code, it potentially increases the instruction cache contention; however, only unrealistically small caches suffered substantially increased miss costs.

Chapter 7

Future Work: Automating Recognition and Transformation

Making something variable is easy. Controlling duration of constancy is the trick.
–Alan Perlis [Perlis 82]

This chapter introduces one important area of future work — how VSO might be automated. The chapter focuses on characteristics of an application that make it amenable to VSO, on the decisions that determine whether a *possible* use of VSO is indeed *profitable*, and on the basic structure and transformations needed by a compiler that automates VSO. Here, the overall task of the static compiler is to recognize program components that are used heavily, identify variables that do not change or change slowly during execution of those components, and to estimate runtime compilation costs and speedups in order to determine breakeven points for VSO candidates. Put another way, the goal is to discover candidate code that might be VSO'd, build a $y = mx + b$ model for the static code and each candidate, statically discard candidates that are provably unprofitable, and then minimize the runtime cost of considering the remaining candidates. The analysis is applied by hand the workload in Section 4.1; for each application, the analysis produced an estimated breakeven point that was conservative and typically close to the measured breakeven point.

Note that many of the issues and observations discussed here apply both to VSO and also to other dynamic optimizations, including language translation [Hansen 73, Johnston 79, Chambers 92, Hölzle 94], instruction set simulation [Magnusson 93, Cmelik & Keppel 94], and application-controlled virtual memory and I/O. In many cases, the term *dynamic optimization* is used; it refers both to VSO and to other kinds of runtime optimization.

The rest of this chapter is organized as follows: Section 7.1 describes a mechanism for automating profitability analysis, based on estimated compilation costs, execution time improvements, and data-dependent reuse rates. Section 7.2 briefly considers the complexity of recognizing VSO opportunities. Section 7.3 investigates how errors in the static compiler's cost estimates affect the final cost of the application. Finally, Section 7.4 hand-applies the techniques in the previous sections to the dissertation workload; a promising result is that, for these applications, estimation errors tended to cancel, producing moderately accurate estimates of when to apply VSO.

7.1 Profitability Analysis

In principle, it is possible to specialize any code with respect to any variables used in that code. Thus, the key to automating VSO is *profitability analysis* – deciding where the cost of runtime specialization is paid back by improved performance. This chapter divides VSO profitability analysis into three major steps: discovering code and variables that may be used for dynamic optimization; generating timing models for each alternative implementation in order to discard provably unprofitable optimizations; and, finally, staging the profitability analysis in order to minimize the runtime cost of deciding whether to use static or VSO'd code. This section examines the analysis step-by-step.

7.1.1 Discovering Candidate Code

Many programs spend a large part of their time executing a small part of their code [Knuth 71], and that part is where dynamic optimizations should be applied. One reason is simple: as described by Amdahl's Law [Amdahl 67, Hennessy & Patterson 96], performance is only improved substantially by eliminating such bottlenecks. In addition, dynamically-optimized code must be reused to pay back the cost of dynamic optimization. Code that is used frequently can be profitable even in the face of high runtime compilation costs and small asymptotic speedups. Code that is used less often requires lower runtime compile costs or higher speedups in order to be profitable.

The "hot spots" where most time is spent may be discovered using static techniques such as program analysis, heuristics based on previous runs [Wall 91] or runs of other programs [Calder et al. 95], or hints. Dynamic identification of dynamic optimization sites is also possible [Hansen 73, Hölzle et al. 91, Chambers 92, Blair 93, Cutts et al. 94, Hölzle 94], but adds substantially to the cost of recognizing optimizations thus limiting static planning for them and thus increases the cost of performing them. Thus, dynamic identification of optimizations tends to work only for dynamic optimizations that have high payback compared to the cost of the static alternatives. The focus of this chapter is on static identification of optimization sites, with a dynamic decision of whether to runtime-optimize any given statically-chosen site.

Hot spots are generally inner loops. Note that program loops may appear entirely in a single function (e.g., §4.1.1, §4.1.3) or may be spread across several routines (e.g., §4.1.2, §4.1.4) and thus require interprocedural analysis. Loops may arise through iteration, recursion and through user-defined iteration constructs, continuation passing, and so on. The workload of this dissertation uses only straightforward iteration.

7.1.2 Discovering Candidate Variables

VSO works by optimizing code with respect to slowly-changing variables, referred to here as *candidate variables*, or by optimizing with respect to variables that change frequently but take on a limited number of values. Only the former cases are considered in the chapter (but see §B.4). Here, "slowly" means that, over some part of the program's execution, the candidate code is executed many more times than the variable changes value.[1] Good candidate variables are read by the inner loop but modified only outside of the inner loop (but see §B.16). Candidate variables are discovered using loop-invariance analysis such as is used to perform static code motion. Aliasing and other such problems make it harder to discover which variables are loop-invariant; VSO thus benefits from ongoing improvements in alias analysis.

The compiler may consider each variable as a whole, without considering changes to component fields, or it may consider individual fields as separate variables. Generally, all variables larger than one bit have component fields. Optimizing with respect to a whole object requires recompilation whenever any part of the object changes. Optimizing with respect to particular components can thus improve reuse rates. For example, the cache simulator used in this dissertation (§4.1.2) is optimized with respect to structure fields that change rarely, but the structure also contains fast-changing fields, such as the data for the simulated cache. However, considering each field separately complicates analysis. For example, deciding whether to specialize with respect to all of an array might take just one decision, while considering specialization with respect to each element can increase both the static and runtime costs.

The applications used in this dissertation specialize with respect to whole variables, except for cache simulation (§4.1.2), which specializes with respect to individual fields in a structure variable. Only bitblt specializes with respect to simple scalar variables; decompression (§4.1.3) and sorting (§4.1.4) both specialize with respect to entire arrays.

7.1.3 Static Optimization

Hot spot and candidate variable analysis may reveal code that is also amenable to static specialization. Profitable candidates are those in which code is a "hot spot" and there are few possible or few common values for the specialization variables, so that the code explosion is acceptable (§B.1). The practical upshot is that any code that is identified as a candidate for VSO is also a candidate for static specialization, and should thus be checked to see if it benefits appreciably from full or common casing. Where it does, static optimization is preferred.

7.1.4 Candidate Variable Reuse Rate

Once candidate variables have been identified, the next step is to determine which can be used profitably for runtime optimization, and which to leave as conventional variables. Since candidates are used to specialize an inner loop, the key metric is the *reuse rate* of a loop, which is the number of iterations a loop is executed before the candidate is changed and the loop must be recompiled.

Reuse rates depend on the optimizations being applied [Przybylski 88]. For example, a loop "for i=1..y" may have a loop body reuse rate of y. If y is constant over the loop, the loop may be unrolled y times, making the reuse rate of the unrolled loop body just one.

[1] "Slowly" could instead be measured relative to other events, such as the number of instructions that get executed, rather than the number of invocations of the candidate code. The two will likely be similar, as inner loops comprise the majority of all instructions that are executed by a program.

The reuse rate may also vary in complex ways. Figure 1.1 shows graphs of execution time versus "problem size" in which the relationship is linear. There are, however, applications with nonlinear behavior, for example an inner loop whose bounds depend on the current outer loop iteration. Nonlinearities may make the reuse rate more difficult and expensive to compute.

The reuse rate may also be implied. For example, an inner loop that iterates over a list will likely use indefinite iteration, with reuse implied by the list length. Pretraversing the list to discover its length adds runtime overhead, and, even if the length is stored explicitly in another program variable, the compiler will likely have trouble associating the length variable with the list.

VSO'd code may depend on several variables; if they change value at different times, they may cause lower reuse rates.

Finally, the reuse rate may depend on values that are discovered during execution, for example a loop that exits when a computation converges. Thus reuse rates sometimes can only be determined exactly by actually running the loop.

Despite a variety of such complications, some cases are simple to analyze. Consider, for example, a doubly-nested loop where the inner loop iterates over `y` ("for i=1..y") and the outer loop iterates over `z`. If the inner loop is specialized with respect to a variable set inside the outer loop, the loop body reuse rate is `y`; specializing with respect to a variable set outside of the outer loop gives a reuse rate of `y*z`. If the inner loop is unrolled by `y` and specialized with respect to a variable set in the outer loop, the loop body reuse rate is one; unrolling by `y` and specializing with respect to a variable set outside the outer loop gives a reuse rate of `z`. In addition, since low reuse rates rarely pay off, such cases may be discarded heuristically (but see §10.5.20). Finally, some reuse information may deduced from, e.g., type information.

Some cases are not amenable to analysis but are still predictable. For example, the sorting application (§4.1.4) has a reuse rate that is roughly $\mathcal{O}(NlgN)$ but depends on the number of data items and their values. Sorting thus suffers from a reuse rate that is complex, implicit and dependent on inner loop data values. Nonetheless, loop-invariance analysis can determine that the number of data items is constant over each call and profiling can readily predict how reuse varies with the number of data items.

7.1.5 Execution Time Analysis

The reuse rate is combined with the estimated execution time of each implementation in order to decide which implementation is fastest. This section briefly considers the general task of estimating the execution cost of a code fragment. The following subsections apply it to runtime compilers and to the static and VSO'd versions of a code fragment.

It is generally difficult to predict the execution time of a code fragment, even for machine code for which all runtime data is available [Wagner et al. 94, Bedichek 95b]. Moreover, VSO optimizers require more precise cost estimates than static optimizers, which typically need only relative timing ("which is faster") rather than absolute execution times. Yet even static compilers have difficulty estimating execution times, and are notorious for sometimes producing worse code when more optimizations are turned on.

Nonetheless, dynamic optimization may reasonably operate with approximations. First, slight errors in the estimated breakeven point tend to cause only slight loss of performance (§7.3). Second, a system may invoke VSO conservatively instead of invoking it right at the breakeven point. Third, profitability analysis depends some places on relative times rather than absolute times, so some absolute errors may cancel each other. Finally, VSO only needs to pay on average; occasional mis-estimations are tolerable.

The result of static timing analysis is generally an equation. For example, the execution time of an if-then-else construct is the probability of the true path times the cost of the true path plus the probability of the false path times the cost of the false path.

Execution times may be estimated several ways, each presenting tradeoffs. IR analysis exposes high-level features, such as loops and conditionals but is typically poor at providing low-level timing details. Machine code analysis is better for low-level estimates but worse for analyzing high-level flow, though machine code may be augmented with IR to raise the level of information [Lucco et al. 95]. Instructions should be analyzed in groups to account for inter-instruction dependencies, exposing the effects of dual-issuing, stalls, etc. Instrumentation provides less predictive information but can provide highly accurate low-level information [Baker 92a].

VSO profitability analysis may combine cost estimates. For example, IR analysis may reveal that a loop runs X times and machine code estimates or instrumentation may predict the cost at Y cycles. The final cost estimate is then XY, computed at runtime once the value of X is known.

Evaluating cost functions at runtime poses two challenges. First, some values may be unavailable, even at runtime (§7.1.4). Second, the cost of runtime evaluation may be high. Replacing runtime variables in cost equations with approximated constants solves both problems. For example, consider a switch with N alternatives: where the N costs are all similar, they might simply be averaged; or, the largest or smallest cost can be used to conservatively bound the execution time. Note that for static code, it is conservative to *under*estimate the execution time.

Since VSO is valuable where there are few common cases, it might appear that instrumentation will rarely measure representative cases, and thus will be useless for estimating performance. However, some situations are readily measured. First, instrumentation can measure low-level component costs. Second, different specialized cases may have similar costs. Third, more general code typically has costs that are less data-dependent than a heavily-specialized counterpart. Fourth, even where there are too many cases to profile, it may be possible to build a model and then calibrate it against the measured cost of particular cases.

Another consideration is that optimization may change the low-level structure of the VSO'd code, so that the final cost is less than the sum of the component costs. For example, the cost of an unrolled loop might be estimated as the cost of one iteration times the unrolling factor. However, even simple optimizations such as constant folding can improve the execution time compared to the isolated low-level costs. The effect is most pronounced for dynamically-generated code (rather than the static code or the dynamic compiler) because timing estimates cannot generally use final code (§7.1.7). It is unclear how much optimizations skew the estimates; however, the effect is to conservatively underestimate the benefit of VSO.

Consider, for example, the code in Figure 2.7. It consists of a loop, a statement that is executed on every iteration and a `switch` to one of `N` statements. IR analysis shows that the code's cost is the number of iterations times the cost of the first statement, the `switch`, and the statement for whichever case is taken. Machine code analysis and/or instrumentation provide the cost of each low-level component. Once all component costs are known, computing the final cost estimate requires knowing the `switch` control value. The static cost estimate may thus be an equation executed at runtime, with a table lookup to determine the cost of the particular case; or, it may use a typical or maximum value for the `switch` case, in order to provide a fully-static cost estimate.

This subsection has described cost estimates in general. The following subsections concentrate on applying cost estimation to the runtime compiler and the code generated by the runtime compiler.

7.1.6 Cost of Runtime Compilation

The runtime compiler is itself static code (but see §10.5.16) and so the runtime compilation cost may be estimated as described in the previous section. Although the runtime compiler calls other routines, many of those routines are simple and regular. For example, the called routines manipulate either trees or templates, both of which have easily-computed sizes. Likewise, the typical operations are hole filling and IR copying; the operations are simple and thus their execution times are easy to deduce.

Tree code generation is more sophisticated and thus the execution time is harder to estimate. However, simple estimates may suffice. For example, each subtree's nodes are known statically, so the cost may be estimated from the number and types of nodes in the tree, using a static code generation cost estimate for each node type. Finally, costs may be measured by running the code generator over each "leaf" subtree and using that cost to help compute the cost of composite trees.

The cost of instruction cache coherency is typically easy to estimate given the size of the region that is being made coherent (Chapter 11). Thus, an estimate of the final code size (e.g., derived from the number of tree nodes) can be used to provide a cache coherency cost estimate.

7.1.7 Cost of Executing The VSO'd Code

Cost estimates for the VSO'd code are more difficult because the VSO'd code does not exist during static analysis. Fortunately, VSO'd code is derived in a relatively straightforward way, so costs may be estimated by using the static code costs and subtracting the cost of application operations that are performed in the VSO compiler. Alternatively (or additionally), the cost may be estimated by adding together estimated costs for each of the components used to build the VSO'd code.

For example, the boxed code in Figure 2.7(a) is being removed from the static-code version. The VSO'd code cost can thus be computed as the static cost minus the cost of the removed operations. Alternatively, the boxed code in Figure 2.7(b) is what will be left after VSO. The VSO'd cost can thus be computed using the static estimates of the boxed operations. In both cases, removing the control flow may enable optimizations between code fragments that span the control flow operation. These optimizations may improve the VSO'd code performance but lead to conservative performance estimates (§7.1.5).

Transformations such as loop unrolling tend to produce faster but larger code. The benefit of unrolling tends to decrease with increasing loop body size and the cost of compilation increases with the number of unrolled iterations, so VSO will tend to profitably unroll only those loops that produce small code. Thus, in order to simplify or augment profitability analysis, an implementation might set an upper limit on generated code size (see also §B.6) and use that as one of the metrics that decide which implementation to use. Code size is a useful metric since it can be easier to estimate or measure than some other metrics, and since a single test for code size can subsume a set of several more-specific tests.

7.1.8 Comparing Costs

Conceptually, a VSO'd application decides whether to invoke static or generate and invoke VSO'd code by reading the values of key variables, evaluating cost functions for each implementation alternative, and then choosing the least expensive implementation.

It is both convenient and appealing to think of VSO in terms of asymptotic speedups. For example, "the VSO'd alternative is asymptotically X times faster than the static code." However, choosing an implementation requires time units, rather than simple ratios. For example, static and VSO'd alternatives that take 100 and 80 cycles per data item, respectively, have the same asymptotic speedup as code taking 200 and 160 cycles. The latter, however, pays off a given runtime compilation cost in half as many invocations.

Nonetheless, using execution times instead of ratios still allows for costs to be normalized in order to simplify the runtime calculations. In addition, even though absolute times are used, cost estimation errors can still cancel if the errors appear consistently in both time estimates.

7.1.9 Comparing Costs Statically

The costs of each alternative are conceptually computed at runtime, but a static compiler can eliminate alternatives statically if it shows that other alternatives will always have better performance. A static compiler can also order the runtime cost computations in order to minimize the runtime overhead. These static optimizations are important, because runtime cost comparisons are an added overhead that, unlike the startup cost (Figure 1.1, pg. 2), increase the execution time even when the static (non-VSO'd) code is used.

As a corollary, good static information reduces the runtime cost of profitability analysis, so approximate static information that is merely "good" can sometimes be more useful than exact runtime information (§7.2, §B.12). In addition, some candidate implementations are never profitable because, for any given problem size, other implementations are always faster. In addition, the static compiler can order the tests to minimize the number of evaluations needed to determine whether static code should be used, so that the extra overhead of choosing among VSO alternatives is amortized over VSO's improved performance. Finally, static compiler can also reduce the runtime overhead by rephrasing the cost functions in terms of the runtime variables. For example, a cost comparison $0.3x + 10 < 0.8x$ may be solved for x to yield an equivalent but faster comparison, $x > 20$.

Unfortunately, the above analysis oversimplifies, since, in general, the compilation cost and speedups are functions of all specialization variables. Reuse rates also differ for differ implementations. Thus, each implementation's cost is potentially a complex multidimensional topology rather than a straight line. Despite this potential complexity, the applications used in this dissertation had straightforward performance characteristics and thus decisions that should be straightforward to evaluate, as evidenced by the hand-analysis performed in Section 7.4.

7.1.10 Summary

In principle, any code may be specialized at runtime with respect to the values of variables used in that code. However, naive application of dynamic specialization can easily lead to excessive recompilation rates and thus worse overall performance. Thus, the key to automating VSO is profitability analysis. For VSO, the analysis consists of discovering code that makes use of variables that are accessed in a way that indicates they are slow-changing; building timing models for each alternative implementation; and, finally, building code to perform runtime profitability analysis.

7.2 Exponential Costs in Analysis

Recognizing VSO opportunities is also tricky because analysis costs can grow exponentially: N candidate variables yield $2^N - 1$ possible runtime optimizers.[2] As a result, a modest number of candidates can generate too many optimizers to consider even during static compilation. A heuristic is to consider optimization with respect to each variable individually, and then consider combinations of particularly profitable variables. Such analysis can prune the number of cases and may provide linear-time analysis in the best case. However, specializing with respect to two variables simultaneously may provide far better performance than the sum of the benefits of individual specialization; for example, `a+b` can be eliminated if the values of both `a` and `b` are known. Thus, pruning may sometimes eliminate profitable alternatives.

Conversely, failing to prune may leave as many as 2^N runtime computations needed in order to determine which strategy is most profitable. Eliminating candidates statically can thus be very important and so good static estimates may sometimes be more useful than exact dynamic information (§B.12).

Finally, exponential growth limits the practicality of splitting variables in order to analyze their components (§7.1.2).

Fortunately, inner loops typically reference a modest number of loop-invariant variables, so analyzers should discover relatively few candidate variables and static compiler execution costs due to

[2] As a trivial detail, it is not clear whether the 2^N'th case, which specializes with respect to no candidate variables, should represent static code or construction of a runtime compiler that generates unspecialized code.

VSO profitability analysis should tend to be small. Thus, performance will be good for most cases, and the static compiler must simply be prepared to use timeouts, etc., to punt on the occasional slow cases.

7.3 Error Analysis

VSO predicts the execution times of various implementations and then selects the fastest alternative. The predictions are, however, inexact. This section investigates how prediction errors affect the choice of implementation and the runtime performance. The results show that a given level of mis-estimation of the execution time has a wide range of effects on actual execution time. These results in turn indicate that execution time computations need further study.

7.3.1 Model Derivation

Errors in execution time estimates cause the analysis to compute a false breakeven point. For problems sizes between the true breakeven point and the false breakeven point, the analysis will select the wrong (slower) implementation and thus increase the execution time.

Execution times are modeled as a slope-intercept form $y = mx+b$, where (approximately) y is the overall execution time, m is the cost per item, x is the problem size, and b is the runtime compilation cost. Errors are expressed as *ratios*, rather than absolute times. For example, a compilation cost error of 0.667 means that the estimated execution time is two-thirds of the real cost, and the ratio 1.0 means "no error".

To compute ratios more simply, the execution time equations are normalized, so that the breakeven point is $x \equiv 1$ and the static code slope is unity, $m_s \equiv 1$. For static code, the normalized cost is thus simply $y_s = x$. For VSO'd code, the cost is $y_v = m_v x + b_v$. Since the breakeven point occurs at $x = y = 1$, the compilation cost, b_v, may be re-expressed in terms of the slope of the VSO'd code, $b_v = 1 - m_v$.

The *asymptotic speedup* is defined as the static code execution time divided by the VSO'd execution time for large problems (§4.2.1). The asymptotic speedup is thus the ratio of the slopes, m_s/m_v. Since the normalized slope of the static code is just 1, the asymptotic speedup is the inverse of the slope of the VSO'd code. Speedups are thus slopes less than one, and larger asymptotic speedups correspond to smaller (shallower) VSO'd slopes.

In the following, the error in the compilation cost is noted Δ_{b_v}. For a given error, the erroneous breakeven point is the value of x where the VSO'd line $y = m_v x + (b_v \cdot \Delta_{b_v})$ intersects the static code line, $y = x$. Rearranging and substituting $y = x$ yields an erroneous breakeven point $x = (b_v \cdot \Delta_{b_v})/(1 - m_v)$.

In the following, the error ratio for the breakeven point is noted as Δ_B, and the error due compilation cost errors is noted $\Delta_{B,b}$. $\Delta_{B,b}$ is the x value of the erroneous breakeven point divided by the real breakeven point, $x \equiv 1$, so $\Delta_{B,b} = (b_v \cdot \Delta_{b_v})/(1 - m_v)$. Substituting $b_v = 1 - m_v$ from above yields Equation 7.1, which shows that errors in the compilation cost estimate, Δ_{b_v}, lead to proportional errors in the breakeven point $\Delta_{B,b}$. Figure 7.1(a) shows an example.

$$\Delta_{B,b} = \Delta_{b_v} \tag{7.1}$$

The analysis for slope errors is similar, but slope errors cause nonlinear errors in the computed breakeven point. Each breakeven point lies on the static code line $y = x$; if Δ_{m_v} is the VSO slope error, then $y = (m_v \cdot \Delta_{m_v})x + b$. Substituting $y = x$, solving for x, substituting $b = 1 - m_v$ and dividing by the real breakeven point, $x \equiv 1$, yields Equation 7.2.

$$\Delta_{B,m} = (1 - m_v)/(1 - m_v \cdot \Delta_m) \tag{7.2}$$

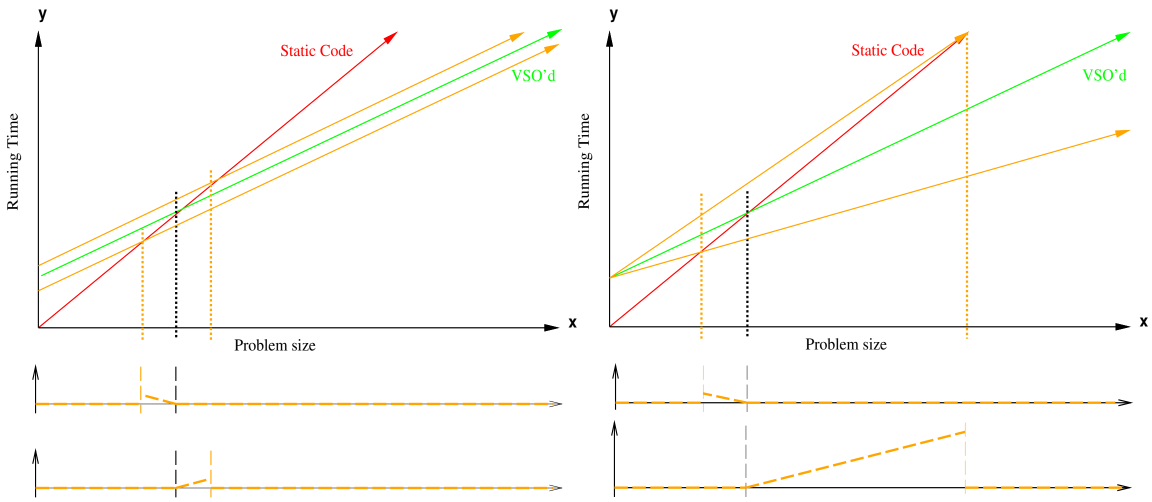
Equation 7.2 tells us several things. First, execution time underestimates ($\Delta_{m_v} < 1$) cause low false breakeven points and overestimates ($\Delta_{m_v} > 1$) cause high false breakeven points. Second, errors such that $m_v \cdot \Delta_{m_v} \geq 1$ show VSO as being asymptotically slower, so the breakeven point is never reached. For example, an asymptotic speedup of 1.10 is $m_v = 1/1.10 = 0.91$; if $\Delta_{m_v} = 1.25$, the erroneous slope $m_v \cdot \Delta_{m_v} = 0.91 \cdot 1.25 = 1.14$, and the VSO estimate never reaches the breakeven point. Finally, errors have a larger effect on the breakeven point with decreasing asymptotic speedup. Figure 7.1(b) shows an example of modest errors in the estimated speedup that lead to wide variation in the breakeven point.

The above shows how errors in the estimated compilation cost and execution time affect the erroneous breakeven point ratio Δ_B; how do these errors affect the overall execution time? For underestimates, the largest execution time error ratio Δ_R occurs at the low erroneous breakeven point Δ_B. It is the cost of the VSO'd code divided by the cost of the static code that should have been used: y_v/y_s. For overestimates, the largest execution time error occurs at the high false breakeven point and is the cost of the static code divided by the cost of the VSO'd code that should have been used, or y_s/y_v. These are equivalently expressed as $\Delta_R = 1 + |y_v/y_s - 1|$.

The cost of a general code fragment is $y = mx + b$. The real breakeven occurs at $x \equiv 1$, the static code slope is $m_s \equiv 1$, and the static code has no compilation cost, so the static code cost at an error breakeven Δ_B is $y_s = m_s(\Delta_B \cdot 1) = \Delta_B$. A VSO'd code fragment costs $y_v = m_v(\Delta_B \cdot 1) + (1 - m_v)$, where the second term is the compilation cost. Substituting in the preceding expression for Δ_R and simplifying yields Equation 7.3, which shows the erroneous execution time ratio as a function of the error in the breakeven point.

$$\Delta_R = 1 + |(m_v + (1 - m_v)/\Delta_B) - 1| \tag{7.3}$$

Combining Equation 7.3 with Equations 7.1 and 7.2 and then generalizing to include errors in the running time estimates of both the static and VSO'd code yields (derivation omitted) Equation 7.4. It shows the maximum increase in the execution time as a function of Δ_{m_s}, the error in the static cost estimate (§7.1.5); Δ_{m_v}, the error in the VSO'd asymptotic cost estimate (§7.1.7); and Δ_{b_v}, which is the error in the runtime compilation cost estimate (§7.1.6). Equation 7.4 is used below to show the effects of various errors.



(a) (b)

Figure 7.1: Cost estimation errors that lead to false breakeven points.

Errors in cost estimation may lead to errors in the computed breakeven point, leading the application to select a slower implementation. Figure (a) shows a range of errors when the recompilation cost estimate is wrong by 25%. Figure (b) shows the range of errors when the asymptotic speedup estimate is wrong by 25%. The breakout bars show increases in the execution time due to errors in errors in the breakeven point; the upper bars shows how underestimates cause VSO to be used before it is profitable, and the lower bars show losses from using static code even after VSO becomes profitable.

$$\begin{aligned}\Delta_R &= 1 + |y_v/y_s - 1| \\ &= 1 + |m_v/m_s + (m_s \cdot \Delta_{m_s} - m_v \cdot \Delta_{m_v})/(m_s \cdot \Delta_{b_v}) - 1| \qquad (7.4)\end{aligned}$$

7.3.2 Execution Times

Table 7.1 shows several examples of mis-estimation, the resulting errors in the estimated breakeven point and the effect that has, measured as the largest increase in the execution time. Compile cost estimation errors tend to produce maximum errors that are smaller than the estimated error and the asymptotic speedup but which grow with increasing speedup.

Asymptotic performance estimation errors can cause relatively large errors when the asymptotic speedup is small. For VSO cost overestimates, the loss is bounded by the asymptotic speedup; if VSO is 1.1 times faster, then failing to use VSO leads to at most a 1.1 dilation over a VSO implementation. Cost underestimates, however, can lead to larger errors because the cost of compilation may greatly

Table 7.1: Example estimation errors and their effects on overall execution time.

Asymptotic Speedup	Estimation Error Is In	Underestimate By	Underestimate Δ_B	Underestimate Worst Δ_R	Overestimate By	Overestimate Δ_B	Overestimate Worst Δ_R
1.10	compile cost	1.25	0.80	1.02	1.25	1.25	1.02
1.50	"	1.25	0.80	1.08	1.25	1.25	1.07
2.50	"	1.25	0.80	1.15	1.25	1.25	1.12
1.10	asymp. perf.	1.25	0.33	1.18	1.25	∞	1.10
1.50	"	1.25	0.71	1.13	1.25	2.00	1.17
2.50	"	1.25	0.88	1.08	1.25	1.20	1.10
1.20	all; errors cancel	1.10	1.00	none	1.10	1.00	none
1.20	all; errors reinforce	1.10	0.44	1.21	1.10	∞	1.20

overwhelm the potential gains from switching to a VSO implementation. As shown by Equation 7.2, the sensitivity to errors decreases with increasing asymptotic speedup. Although wrong choices tend to hurt performance more, there is also a smaller "window" in which analysis selects the wrong implementation.

Where the static, VSO'd and runtime compile costs are all mis-estimated, errors may cancel, leading to accurate prediction and no loss of performance. Where errors reinforce, the execution time error can exceed both the asymptotic speedup and the estimated error.

Which errors are most significant? Arguably, errors are more serious when they cause slowdown relative to the static code, when the peak discrepancy is greatest, when the performance loss covers the greatest number of cases, or when errors cause the greatest average loss. Perhaps high error breakeven points are better because they have bounded performance loss? For example, if the faster code is asymptotically 1.1 times faster, then choosing the wrong implementation leads to at most a 1.1 dilation. On the other hand, low error breakeven points may cause a larger *percentage* loss but a smaller absolute loss. For example, the execution time might increase from one second to four, a 400% increase, but the absolute impact is small compared to an error of "only" 1.1 that nonetheless increases the execution time from 1,000 seconds to 1,100 seconds.

7.4 Application

This section shows the results of hand-applying profitability analysis to the dissertation workload in order to show how the cost estimation ideas can be applied and how cost estimation errors affect performance. Keep in mind that the workload used here was chosen because it showed speedups, which limits the applicability of these results.

Candidate discovery is done as above (§7.1.1, §7.1.2). No cost estimate is computed for parts of the inner loop that are always implemented using static code. Omitting the always-static code prevents asymptotic performance calculations for the whole loop, but still allows breakeven analysis, because the VSO compilation cost is paid back by a certain *number* of invocations, irrespective of

Table 7.2: Estimated reuse rates and incremental costs.

Application	Impl.	Rate of Reuse Via	Rate of Reuse Value	Incremental Cost Via	Incremental Cost Value (cycles)
bitblt	Static	I	$wid \cdot hgt$	I+M	$18 + 50/wid$
	VSO	I	*height*	I+P, avg	$14 \cdot wid + 40$
cache–1	Static	P	$> 100,000$	I+M+P, avg	48
	VSO		(same)	I+P, avg	42
cache–4	Static	P	$> 100,000$	I+M+P, avg	68
	VSO		(same)	I+P, avg	60
decompression	Static	P	$1,000 - 10,000,000$	I+M+P, avg	52
	VSO		(same)	I+P, avg	44
sorting	Static	P	$1000–100,000$	I+M+P, avg	60
		I	$\#items..\#items^2$		
	VSO		(same)	I+P, avg	48

The column *Via* indicates how the information was discovered: *I* is IR analysis, *M* is machine code analysis, *P* is profiling, and *avg* is averages over several paths. *Value* for *Rate of Reuse* is the number of iterations of the application inner loop; the candidate code may be only a part of the inner loop (§7.1.1). *Value* for *Incremental Cost* is the estimated number of machine cycles for one inner iteration. For bitblt, *wid* is the width in words.

their contribution to the overall execution time.

Where control flow is sufficiently regular (data-independent), the rate of reuse of is computed using IR analysis. Reuse for irregular cases is deduced by profiling and the reuse rate is here taken as a constant equal to the smallest profiled reuse rate. For sorting, it appears that analysis would be able to produce conservative upper and lower bounds on the number of predicate invocations as a function of the number of data items. Table 7.1 shows the reuse rates for each application.

The execution-time estimates are computed as an *incremental cost* that is paid once for each use of the code. The static code cost is estimated by combining IR and machine code analysis (§7.1.5). Where the execution path depends on specialization variables, the estimate is a function of those variables. Otherwise, varying path costs are computed as an average of costs based on profiles. For example the cache–1 cost is the weighted average of the costs for simulating a hit and simulating a miss. The low-level cost of the code is assumed to be two cycles per instruction with an extra cycle for each load or branch. Table 7.2 shows the incremental cost for each application's static code.

The runtime compile cost is estimated as the size of the IR assembled by the runtime compiler times a code generation cost of 2,000 cycles (1,000 instructions per generated instruction times two cycles per instruction). Table 7.3 shows estimated costs for each application. Comparing against a real system is difficult, because the code generation costs for the results in Chapter 5 are also estimates. However, those results estimated the cost as 1,000 instructions per *generated* instruction and also include measured costs for IR generation and for linking.

The VSO'd code does not exist during analysis, so the cost is estimated by analyzing the VSO compiler trees that are used to build the final code and by using analysis and profiling of the static

Table 7.3: Estimated runtime compile costs.

Application	Compile Cost		
	Via	Value (cycles)	mSec @ 40MHz
bitblt	I+P, avg	$(12 \cdot wid + 24) \cdot 2,000$	$0.6 \cdot wid + 1.2$
cache–1	I	$36 \cdot 2,000$	1.8
cache–4	I+P	$(13 \cdot assoc + 44) \cdot 2,000$	$0.7 \cdot assoc + 2.2$
decompression	I+P	$4,700 \cdot 2,000$	24
sorting	I+P, avg	$48 \cdot 2,000$	2.4

For columns *Via* and *Value*, see Table 7.2. The cost *mSec @ 40MHz* is *Value* multiplied by the cost per cycle on the same platform used for Table 5.2; see that table's *total* cost.

code to determine how many of each type of tree will appear. In the static code, the cost over dynamically-uncertain paths was averaged. In the VSO'd code, the uncertain paths may appear both as uncertainty about which subtree is being used and as uncertain paths through the subtrees. However, since there is a direct correspondence between static code paths and the subtrees and subtree paths that replace them, averaging is still straightforward. The low-level cost of each node is estimated by counting the number of each type of node in a subtree. For example, each integer add is assumed to take two cycles, while each pointer add is presumed to appear as an addressing mode and thus take zero cycles.

Table 7.4 shows, for each application, the computed breakeven and static compiler implementation decision. Where reuse depends on known runtime variables, the static compiler emits a runtime test (§7.1.8). However, where no profiled values reach VSO profitability, as with bitblt, the static compiler chooses static code in order to avoid the overhead of a runtime test. Where reuse rates come from profiles, the static compiler assumes the smallest profiled value is the lowest reuse value that will be seen in practice and chooses VSO only if it would have been profitable for all profile runs. For example, the cache–4 reuse rate is presumed to be 100,000, so VSO is used only for associativities that would be profitable for inputs of 100,000 records, even though wider associativities might show speedups on longer traces.

The results are encouraging: comparing Table 7.4 to the measured breakeven points in Table 5.1 shows that cost estimation predicts breakeven points fairly accurately given the quality of available information. For example, sorting is profitable with VSO for more than about 1,250 records; automated analysis indicates that sorting is profitable only when there are more than 8,000 records. Thus, slower static code is used for a range of inputs, with dilation ranging from negligible to a dilation of 1.3 over the faster VSO'd code. In addition, for all applications, all loss comes from failing to take advantage of faster VSO'd code, rather than from choosing VSO where static code would have been faster.

Table 7.4: Estimated breakeven points.

Application	Breakeven	Static Decision	Worst Δ_R
bitblt	$4 \cdot wid \cdot hgt + 10 \cdot hgt$ $-24,000 \cdot wid > 48,000$	never	none
cache–1	12,000 refs	always	none
cache–4	$3,250 \cdot assoc + 11,000$ refs	$assoc < 27$	none
decompression	1.17 MB	never	1.13
sorting	8,000 predicate invocations	$\#items > 8,000$	1.3

Breakeven is the cost computed by the static compiler. *Static Decision* is the decision made by the static compiler based on the analysis and profiles of the code. *Worst* Δ_R shows the worst performance loss that would be caused in practice by using VSO as indicated by analysis.

7.5 Conclusions

Profitability analysis is the key to automatic application of VSO, because VSO transformations can be applied to almost any code fragment, but only some of the applications are profitable. For VSO, profitability analysis consists of discovering code and variables that might be optimized with VSO; building timing models for each alternative implementation; and, finally, discarding provably unprofitable alternatives and, for the remaining, building code to perform runtime profitability analysis.

This chapter shows that there are many details to profitability analysis. For example, it is sometimes difficult to determine when the values of variables change, and it is generally difficult to accurately predict the performance of a given code fragment. However, many of the steps have special cases that work with much simpler treatment. A key question, then, is whether using simplified treatment for each step can still generate a system that is generally useful, or whether the resulting system recognizes too few of the profitable cases. Although this chapter does not answer those questions, it does describe some of the needed components, investigates how errors in the analysis can affect profitability of the resulting code, and hand-applies the analysis to the dissertation workload; for the dissertation workload, the estimated breakeven points were close to the measured values, were slightly conservative, and would work well in practice.

Part II

Runtime Code Generation

Chapter 8

Runtime Code Generation

Part II of this dissertation is about *Runtime Code Generation* or *RTCG*, which is general runtime construction and modification of code within an executing program. Part I focused specifically on improving performance over the best statically-typed, statically-compiled code. Part II studies RTCG more generally. It provides working definitions for studying RTCG systems, then examines a variety of RTCG design and implementation issues and techniques for a wide variety of RTCG systems. The study thus serves as a framework for identifying and categorizing uses of RTCG.

Chapter 9 provides a brief look at the fundamentals, providing definitions for basics such as "runtime", "code" and "generation". Chapter 10 examines a wide variety of code modification issues, RTCG design and implementation techniques, reports on some traps and pitfalls that have plagued RTCG over the years, and provides references to a broad base of related work. Chapter 11 investigates instruction cache coherency. Finally, Chapter 12 concludes.

Chapter 9

Definitions

There a side o' me I show — a side I keep hidden
Sometimes they flip-flop — hey! No kiddin'
I use my left brain and then I use the right
I'm the day — I'm the night
My personality has a plurality
and that's why they call me "Mr. Duality"
That's me... that's me too

–"Mr. Duality", The Bobs

The etymology of the word code *is most interesting. It seems evident that in its original form it has reference to the paginated format of a book as distinguished from the earlier scrolls. Books with leaves fastened together at a common edge were apparently used in Roman times to record compilations of laws, and this is the basis for the original meanings of* code, *as a systematic collection, compilation, or formulation of laws, principles, rules or regulations.*

–[Patterson 60]

Typical dictionary definitions for "code" and "data" are vague enough that two people can reasonably disagree about whether a given system is manipulating code or data. In the same way, "runtime" and "code generation" are sufficiently vague to cause problems. This chapter explores some ambiguities of the existing definitions, provides more precise definitions and describes the particular focus of the dissertation.

This chapter is organized as follows. Section 9.1 defines "code" and "data". Section 9.2 treats "runtime". and Section 9.3 explores "code generation". Section 9.4 combines these to provide definitions for "compiler", "runtime code generation" and so on. Finally, Section 9.5 discusses some aspects of typing that are different when applied to code and data.

9.1 Code

What is code? Many definitions are possible; for example, some processors execute "machine code" using hardware; others execute microcode that in turn manipulates "machine code" as data. Is the "code" the machine instructions or the microcode? A reflective [Smith 84, Wand 88] view presents computing as an infinite tower of processors, with each but the lowest interpreting the "code" of the processor below it. In short, all code is data. Conversely, the bit pattern `0011` means "three" to the integer ALU only because the ALU *interprets* the bit pattern as a command to set the ALU state to represent "three". However, the ALU state need not use the literal representation 0011; nor does 0011 mean "three" to the floating-point ALU. Since the meaning of a bit pattern depends on interpretation, all data may be seen as code.[1]

These views — all code is data and all data is code — are amusing, but not useful. The real reason for distinguishing "code" from "data" is that doing so allows us to make broad statements and reason simply about computer systems. For example, within a given context, "code" typically represents commands, where "data" is typically descriptive. Similarly, "code" typically modifies data but is not itself modified, thus simplifying reasoning about both the behavior of code and about the extent of modifications performed by stores. Nonetheless, these uses are only guidelines. For example, in declarative languages such as Prolog "code" is a description of the desired solution rather than commands on how to achieve it. Likewise, a variety of tools modify code, for example, debuggers and dynamic linkers.

"Code" is any thing which we intend to feed to some *execution engine* and interpret as commands for that engine. In short, "code" is anything fetched via some kind of program counter. For example, virtual machine code is "code" for the interpreter that executes it but "data" to the hardware that is running the interpreter. Similarly, 80386 instructions are "native machine code" for the Intel 80386 processor as a whole, but not for the core unit that executes microinstructions. Conversely, the microinstructions are not "code" for the processor as a whole. The insight, then, is that "code" is defined by the engine that executes it, whether that engine exists in hardware or software.

> **code** (kōd), *n.* **1** The representation of a set of commands. **2** A set of inputs to an engine that carries out those commands.

Defining code precisely requires attention to a variety of details. Whose view of "code" should we use? In Smalltalk, a user's command is compiled to bytecodes [Goldberg & Robson 83] that are translated to machine code [Deutsch & Schiffman 84] and may then be compiled to microcode [Ditzel & McLellan 87, Ditzel et al. 87a, Ditzel et al. 87b]. The user's view is that the command text is "code". However, the view of the microengine is that only microinstructions are code; not the expression, the bytecodes or the machine code. A given form, then, may simultaneously be code from one viewpoint and data from another. In addition, Turing equivalence tells us that code in one form may be translated to equivalent code in another; likewise, tools such as disassemblers and decompilers can transform executable code back into source text – that is, data [Halpern 65].

Some tools, such as compilers, manipulate several kinds of code. For example, an optimizer will execute constructs in the input, such as "if (false) ...". Likewise, the output form is "code", constructed around an execution model. Finally, the compiler is itself implemented using some kind of code; few compilers, however, are reflectively "aware" of, or able to manipulate, their own code form [Rodriguez Jr. 91]. Although it is usually clear which engine is implied with a given "code", some historically-common bugs come from using the wrong engine. Examples include:

[1] For completeness: uninterpreted bit patterns represent neither "code" nor "data", exactly because they are uninterpreted.

using module definitions for host libraries instead of for the target; and performing compile-time expression evaluation using host arithmetic instead of target arithmetic. Additional problems arise in retargeting compilers that execute the code they generate [Appel 94].

What about data values embedded in instructions — so-called *literals* or *immediates* [von Neumann 45] — are they "code", too? Although they are part of the language accepted by the engine, we don't normally think of them as "code". However, `add r0,r1`, "add the contents of registers zero and one", is a different instruction than `add r0,3`, "add three to the contents of register zero". The effect of the literal is to transform or specialize an instruction. Thus, although `3` is not itself a machine code instruction it is *part* of such an instruction, in much the same way that bit number five of the instruction opcode is not itself an opcode but still comprises a *part* of it. Although immediates are eventually used as data, they are first interpreted as a part of the instruction. For example, the National Semiconductor 32000 [Meissner 95] encodes literals in a big-endian byte order [James 90], even though other values are encoded little-endian.

Finally, the execution engine must be named explicitly because a single fragment may have different meaning depending on context. For example, early computer designers recognized that code and data could be stored in the same memory as long as there was a mechanism for the computer to "distinguish a number from an order". That is, code does not differentiate itself,[2] instead, the computer must have an explicit mechanism for separating code from data [Burks et al. 46]. Particularly amusing examples of context dependence come from the folklore programming challenge to write nontrivial programs that are simultaneously valid programs in each of several programming languages (§10.7); Figure 9.1 shows an example of one such fragment.

The focus of this dissertation is on *machine code*, described by the instruction set architectures put forth in processor architecture manuals. This dissertation also has occasional forays into higher- and lower-level "codes" [Pittman 87] and into the translation between levels. The important observation, however, is that any general set of rules that apply to machine code apply also to other code forms. Only the specific costs and frequencies of operations vary.

9.2 Runtime

The meanings of "runtime" and, equivalently, of "self" in "self-modifying", depend on the spatial and temporal *extent* of the code being considered. They are thus relative definitions rather than absolute, much as the definition of "code" is relative to the engine that executes it.

> *Try this simple exercise. Point a finger at your belly-button. Poke. You poked yourself. Your finger didn't poke itself.*
>
> –Adam Kao

Extent describes how and when interactions occur. Although "extent" seems at first unrelated to "runtime", the extent is important, because it is only sensible to consider runtime interactions among entities that *can* interact. Examples of spatial extent include individual bits, instruction

[2]Pun: even code that implements a numerical or symbolic math programs will rarely differentiate itself.

```
                                                                        (* /*
C > /) 2> /dev/null & echo hello world!; exit
C << 'CC'
C
C This file is a portable:
C – (ANSI/KR) C source, about which lint(1) does not complain/warn;
C – (ISO) Pascal source;
C – (ANSI) Fortran-77 source;
C – (POSIX/V7/BSD/System V) shell script.
C
C*/ foo()) { extern *bar; return ++bar; /*
C*) program {*/} int (* /*
C*) (output); begin writeln('hello world!') end. {
        print *, 'hello world!'
        end
C*/ bar); main() { puts("hello world!"); return !foo(); /*
CC
C*/}
```

Figure 9.1: A valid program in several different languages.

A code fragment that is the "hello, world" program [Kernighan & Ritchie 88] simultaneously in several languages. Note that the "(" in line 1 must be in column 73 or higher.

fields, code blocks, pages, modules, programs, storage devices, file systems, and so on. Examples of temporal extent include clock cycles, the lifetime of a procedure call frame, the execution time of a program, the lifetime of a collection of files, and so on.

Most entities are composites of other entities. Thus, descriptions of "runtime" behavior or "self" for one entity implicitly name a variety of entities, each of which may have varying extents. For example, when an instruction modifies another block in the same procedure, the procedure is self-modifying, but the instruction is not. Likewise, different features of a single entity may have different extents. For example, an *add* instruction's temporal extent may be a single cycle of virtual (process) time, but page faults and arithmetic exceptions may arbitrarily extend the wallclock (real) time.

Careful analysis of extents can reveal some surprises. For example, instruction cache coherency problems appear at first to be a result of program self-modification. Yet, even computers that *never* perform code generation can still require cache coherency! The view of each *program* is that time (temporal extent) begins at invocation and ends at termination and instruction memory (spatial extent) consists of the range of virtual addresses holding the program's code. Thus, the *program* requires instruction cache coherency only when it writes a new value into instruction memory and then executes that location. From the *system's* perspective, however, time begins when the computer is booted and ends when the computer is shut down, and the instruction memory consists of all physical memory that at some time holds executable instructions. Since memory is finite, a general system will eventually reuse memory and thus eventually put instructions into memory that had previously held other instructions. At that point, the *system's* perspective is that it is modifying instruction memory, and it must, therefore, perform instruction cache coherency. Note that using virtually-addressed caches tagged by address space identifier (ASID) doesn't solve the problem, either, because ASIDs are also of finite size and eventually get reused.

Finally, careful consideration of extent is useful in design. For example, instruction fetch is the first stage of instruction execution; logically, the fetch stage completes before other stages are started. However, an executing instruction may fault at any stage and be restarted, thus expanding the temporal extent of "instruction fetch" from instruction issue up through instruction completion. As a consequence, if an instruction location is updated while the instruction is executing, the new instruction may be executed instead of the old, even though "instruction fetch" logically completed before the modification. The inconsistency arises because the restart of faulting instructions expands the temporal extent of the instruction fetch. One way to avoid problems is to disallow *all* updates to a set of instructions until all of those instructions have completed; this is the design used in Chapter 11.

The importance of extent is that it focuses on interactions between entities. This dissertation focuses on extents typical of processes in contemporary operating systems; the reasons are largely that most systems perform code generation only within an application, save the code only for the duration of a single process invocation, and because operating systems typically provide instruction cache coherency only down to the granularity of processes.

9.3 "Generation" or "Modification"?

What constitutes "generation" of code and what constitutes "modification"? A simple definition is that code is "generated" when the behavior of the new code does not depend on the previous value where the code has been written, whereas "modification" is when a single logical sequence of instructions depends in some way on the prior memory values, for example if the new sequence is a combination of old and new instructions.

A variety of details are important. First, it is necessary to define "sequence"; that is, the extent as in Section 9.2. Common units include instructions, basic blocks, and procedures. Generating a new procedure, for example, does not involve any "modification" at the basic block or procedure level, but may be seen as program modification.

Second, when code is added to a running program, it is usually irrelevant from the program's perspective whether the code is "new". That is, dynamic insertion of existing code is largely indistinguishable from dynamic insertion of new code. Thus, for example, dynamic linking is "code generation" whether or not the code existed when the static code was compiled. Likewise, copying instructions from one location to another is "code generation", even though the instructions being copied may have been created at the same time as the rest of the "static" program code.

Third, a *value-preserving write* is one that overwrites memory with its old value. It is an update in the sense that a write has been performed, but the memory values and code behavior are unchanged and no cache coherency is required. It is often safe (conservative) to omit value-preserving writes (see §B.15). In general, however, writes may have side-effects, and thus a conservative definition of

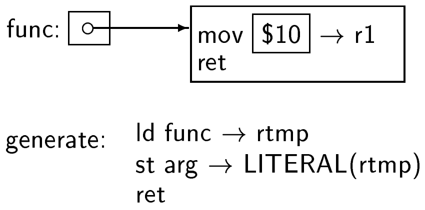


Figure 9.2: Code generation implemented using code modification.

update must include value-preserving writes.

Fourth, "generation" and "modification" may include changes that are in principle observable but which are not observed in practice. For example, all references to the dynamically-linked sine() routine may, in practice, execute the same instructions. However, *if* there had been a spurious reference to sine() before the dynamic linker had been invoked, the reference would have caused an exception. The question may be phrased "if a memory location is updated but only a single value is ever fetched, has the memory really changed?" The deep answer is like answering "if a tree falls in the forest and nobody hears it, does it make a sound?" However, the pragmatic answer is yes, it has changed; in practice, programs have occasional errors that expose such changes [Sun 94]. Thus, any runtime change of value, protection or existence should be conservatively considered as update, even if all references to the memory nominally observe the same value.

Finally, distinguishing between "generation" or creation of new code and "modification" of existing code is tricky, because generation may be implemented as modification. For example, Figure 9.2 shows two code fragments, generate and func. *Logically*, generate is a compiler that deletes the code for func, frees the storage, allocates space for new code, and generates new code. However, the *implementation* takes advantage of the fact that the old and new versions are the same size and have very nearly the same bit patterns; the implementation thus simply modifies part of the original code storage for func. The behavior is as if "generation" were being performed, but the implementation uses "modification".

In summary, "generation" and "modification" are both examples of update. This dissertation generally treats both as "generation", since most of the issues are the same, and since many of the differences are hidden by manipulating the code in structured ways.

9.4 Definitions

Instruction Space An *instruction space* is the range of locations that may be referenced directly via an execution engine's program counter. For example, in a conventional system, storage addressed by instruction virtual addresses is considered part of the instruction space. In contrast, a disk location containing code is not part of the instruction space, even though it may be referenced as a side-effect of a demand-paged virtual memory fault caused by an instruction-space reference.

Instruction Space Modification Anything that writes to an area which has been defined as holding or being able to hold "code". For example, the Ural, the Gier, the IBM 7090, and many other machines performed procedure calls by storing procedure arguments in memory immediately after the "call" instruction (§10.4). Thus, although no *code* was written, the instruction space was still being modified. Instruction space modification also includes code modification that is visible to execution within the instruction space, whether or not the modification was initiated within the process. For example, a debugger that patches a target address space is modifying that space, since after the modification, different code will be executed in the target space. In contrast, moving code from one set of physical pages and then mapping the new pages at their old location is not instruction space modification at the virtual address level — though it *is* modification at the level of the physical pages and will generally require instruction cache coherency — because no change is visible within the instruction space.[3]

Code Generator, Compiler An entity which produces or modifies code. Traditionally, "compiler" refers to an entire system that translates source code to machine code, including lexing, parsing and optimization, and a "code generator" that performs the mapping of the compiler's internal representation onto machine instructions. However, in an RTCG system, the source representation is rarely source text and the compiler representation may already be instructions; for example, in a template compiler. Thus, in the context of RTCG, a runtime "compiler" is the entity which consumes source representation (see "Source Representation", pg. 116) and produces executable machine code. It may be indistinguishable from or even simpler than a traditional code generator.

Runtime Code Generation Anything that creates or modifies instructions in a structured way. "Structured" is vague, but includes the requirement that it conforms to the system's ABI (Applications Binary Interface; typically a machine instruction set) and API (Applications Program Interface; typically a set of operating system services); that race conditions are handled predictably; and that reentrancy is dealt with either by viewing the code as encoding a value — so that sharing code implies sharing the value — or by disallowing sharing of modified code.[4] "Structured" manipulation also implies that code is manipulated using a well-defined interface whose machine- and application-dependencies are clearly stated.

Self-Modifying Code Generating or modifying instructions, whether or not the modification is done in a well-structured way. For example, an 80x86 program that modifies a subsequent instruction and then executes it without an intervening branch is self-modifying code. Although it is self-modifying code, it is *not* runtime code generation: the ABI for the Intel Pentium Pro requires that a branch is executed between the time an instruction is written and the time it is executed.

[3] Timing changes *are* potentially visible but are indistinguishable from other delays and thus cannot be attributed to code modification.

[4] "Sharing" here refers to all sharing, including sharing within a processes as well sharing of code pages between independent processes. Thus, VSO'd code might be shared by threads of a parallel application even if it is not shared among instances of that parallel application.

9.5 The "Type" of Code

The notion of "type" is meant to provide an abstract description of compatibility. "Type" is thus an abstraction of a set of values. One of the key benefits of using types is that they allow both the programmer and the code manipulation tools to reason about entities generally. In particular, they allow general statements about which combinations are desirable and thus error detection when there is incomplete information — in particular, before execution. In an ideal system, types constrain a program's execution precisely and static examination can prove that all valid inputs lead to valid outputs. Most practical type systems, however, must either relax type constraints, which inhibits precise statement of the desired runtime behavior, or must defer some type checking to runtime, thus allowing checking of only specific execution paths rather than checking of program invocation in general.

Code may be typed a variety of ways. For example, one "type" is to allow arbitrary combinations of machine instructions. Alternatively, the type system may be restrictive and allow only very specific code combinations. For example, a code type system for a particular template compiler may specify that a loop prologue always appears first, then a bounded number of loop bodies, and so on.

Code may be typed in a variety of ways that are unconventional for data typing. "Type" may constrain the legal successors to an invocation (e.g., code must always return to the caller), the execution time [Deutsch & Grant 71],[5] what memory that may be read and what may be updated by the code (§10.5.15), and so on. Some classical data operations may be disallowed or constrained when applied to code. For example, incrementing an instruction [Mead 90].

Likewise, "type" applied to code may encode certain information that is not needed when typing data, including the normal and exceptional successors to an instruction, execute permissions, and so on. Finally, the relatively irregular coding of "code" may convolute the typing system. For example, probabilistic protection via sparse addressing relies on protecting linkage segments from being read in order to prevent a malicious segment from determining the ultimate target of an otherwise protected jump [Yarvin & Sah 93]. However, processor delay-slot mechanisms may allow execution of isolated instructions from the linkage segment and thus cleverly promote unreadable code to readable data, thus violating type-based protection.

[5] Real-time systems include timing as one component of type.

Chapter 10

Runtime Code Generation: Issues and Examples

Three things are most perilous:
Connectors that corrode
Unproven Algorithms
And self-modifying code

–From "Threes", by Frank Hayes [Hayes]

The first time, it's a KLUGE!
The second, a trick.
Later, it's a well-established technique!

–Mike Broido

Do not become archivists of facts. Try to penetrate to the secret of their occurrence, persistently search for the laws which govern them.

–Ivan Pavlov

Runtime code generation is not new: it was conceived even before the first stored-program computers were constructed. This chapter provides a brief look at a variety of ways in which RTCG has been used and at a variety of RTCG design and implementation issues and techniques. For brevity, this chapter uses the term "runtime code generation" to include uses of self-modifying code and instruction space modification.

This chapter has several goals. First, it provides a brief historical retrospective. Second, it presents examples to help to explain when and why RTCG is useful. Third, it surveys RTCG. Finally, it provides a broad base of related work for the other parts of this dissertation.[1]

The Chapter is organized as follows. Section 10.1 discusses reasons for using runtime code generation and Section 10.2 describes design and implementation decisions that are common to all systems that use it. Section 10.3 provides an extensive list of systems that have performed some kind of code modification, and Section 10.4 describes instruction space modifications that don't

[1]There are bound to be some inaccuracies. Many RTCG practices have been "handed down" from programmer to programmer. Moreover, since the use of RTCG has been deprecated, the techniques are often undocumented. In some cases, the original authors can be located, but even they have vague memories of code that is now 30–40 years old.

involve changing code. Section 10.5 describes a variety of issues and techniques that appear in the context of RTCG; some, such as instruction cache coherency and code generator performance, are not themselves RTCG, but are nonetheless crucial to the success of RTCG systems. Section 10.6 explores a variety of practical problems that have dogged instruction space modification through the years. Finally, Section 10.7 describes a variety of systems that do not perform machine code instruction space modification but which are nonetheless closely related.

10.1 Reasons for Using RTCG

There are two principle reasons for using runtime code generation: necessity and performance. The first arises because some basic system operations, such as reading code from secondary storage into memory, cannot be performed without modifying instruction memory. The second arises because additional runtime information allows general code to be optimized for the task at hand.

Behind the simple veil of performance, however, there are a variety of other reasons. They generally center around the observation that general code that is too slow is worse than specialized code that is less broadly useful but which runs fast [Lampson 84]. In particular, many of the systems described in this chapter use RTCG in order to maintain performance while improving some other metric, such as flexibility; ease of construction, debugging and maintenance; interactive response; reliability; or portability. That is, RTCG can help by flattening indirection and eliminating redundant operations that are usually implied by improving these other metrics.

Metrics such as ease of construction and reliability are related to performance, but there are several reasons to consider them separately. First, there is no direct correlation between the metrics. For example, code that is twice as robust is not necessarily half as fast. The principle advantage of performance as a metric is that it is much easier to measure than most other metrics. For example, what is a sensible measure for determining if one program implementation is “twice as clean” as another? A second reason for considering them independently is that many metrics can be improved together. For example, improving flexibility often improves code reuse and thus robustness. Finally, studying RTCG’s effect on these metrics individually gives insights about when RTCG will help and how it should be applied.

For each system described below, it is useful to understand what metric is being improved, what specific RTCG mechanism is being used and also what application information changes slowly enough that RTCG is an effective solution.

10.2 Design and Implementation Strategies

The focus of this chapter is on systems that generate or modify machine instructions or which otherwise update a machine code instruction space during program execution. All of these systems deal with several issues related to code generation: the internal representation of the code to be manipulated at runtime; the source representation that drives runtime compilation; and the code generation policy. These issues also appear in standard compilers, but runtime compilers demonstrate a much wider range of choices.

Internal Representation A system that manipulates code must have some internal representation for that code.[2] Important features of the representation are its implementation, its interface to the rest of the system, and the information contained in the representation.

- *Implementation:* The representation may depend on any of: the source language used for creating the representation; the source language for the program which it represents; the application or algorithm which it helps to implement; and the architecture and operating system of the *host* machine where it is manipulated and the *target* machine where the generated code is executed. For example, the tree compilers used in Part I of this dissertation use a representation in which the individual nodes depend only on low-level details of the language that it represents, while the trees built from the nodes depend also on the applications. The template compilers use a representation (templates) that depend also on the target machine.
- *Interface:* The interface reflects the same dependencies, but may have a different set than the implementation. For example, templates are machine-specific, but the templates used in Part I of this dissertation have a machine-independent interface.
- *Content:* Representations may be *high-level* or *low-level*. High-level representations are typically more abstract; low-level representations typically contain less general information but are faster to convert to executable code. High-level representations may be augmented with information to improve code generator performance [Franz 94], and low-level representations may be augmented with high-level information in order to improve or simplify analysis, again improving code generator performance and potentially improving the quality of the generated code [Gosling 95, Lucco et al. 95, Wahbe et al. 94].

Representations include source text, used commonly for edit-compile-debug systems [Hansen 73, Brown 76, Ford & Sawamiphakdi 85, Stemple et al. 92, Stemple et al. 93]; abstract syntax trees or other high-level forms [Fyfe et al. 91, Engler & Proebsting 94, Franz 94, Consel & Noel 96]; bytecodes, three-address codes or some other virtual machine instruction set [Deutsch & Schiffman 84, Chambers 92, Wahbe et al. 93, Pountain 94, Krall & Berger 94, Krall & Berger 95b, Krall & Berger 95a]; machine code [Deutsch & Grant 71, Massalin 90, Fitzhardinge 93, Leone & Lee 94, Auslander et al. 95, Auslander et al. 96, Eager 95]; microcode [Wilkes 69, Tucker & Flynn 71, Reigel et al. 72, Robbi 72, Wilkes 72, Flynn & Neuhauser 75, Ebeling 76, Roskos & Winner 81, Ditzel & McLellan 87, Ditzel et al. 87a, Colwell et al. 88, Pountain 88, Kehl 93, Pow 94, Gwennap 95c, Gwennap 95a]; or even programs for programmable gate logic [Snyder 82, Koopman 89, Brunvand & Sproull 89, Johnson & Fox 91, Gokhale et al. 91, Athanas & Silverman 93, DeHon 94].

Source Representation The *source representation* is the form of the code being provided to the compiler. In compilers for C, FORTRAN and so on, the source representation is text. However,

[2] Arguably, a program that randomly permutes memory does not have a distinct model of code.

a wide variety of other forms are possible. For example, binary translation tools compile machine code for some target machine to run on a different host machine; the source representation is target machine instructions. Likewise, in Part I of this dissertation, the source representation is a set of data values: the size and associativity of a simulated cache; the order of field comparisons for sorting records; and so on.

Code Generation Policy Code generation may be performed any time between the first point at which parts of the source representation become available and the last point at which the runtime-generated code is used.[3] Policies include [Ford & Sawamiphakdi 85, Stonebraker & Rowe 85, Heering et al. 91]:

- *Demand* or *Lazy* — code is produced as needed. Typically, a particular extent (§9.2) is implied, for example "demand generation of procedures".
- *Greedy* — the generated code is viewed as a transformation of a source representation, and all new code is generated immediately after any change to the source representation.
- *Periodic* — code is generated "every so often" in response to an event; the period may be based on time, a user command, entry to a particular procedure, and so on.
- *Background* — code is generated as time is available, for example, while waiting for user input.

Various factors influence the choice of update policy, including the amount of information that is available during code generation; the frequency with which advance compilation is performed "uselessly"; the turnaround time and user pauses implied by putting code generation at different places in the computation; and the complexity of code generation, including synchronization of the code generator with the consumer of the generated code.

Performance and Measurement Different systems optimize different aspects of "performance"; specifically, some optimize latency, while others optimize throughput. Measurement techniques likewise vary. Where asymptotic code quality is most important, the performing measurements are sometimes taken after running the program to generate and optimize code and thus "pre-warm" the code cache [Franz 94, Chambers & Ungar 89, Chambers et al. 89, Hölzle 94]. On the other hand, interactive response may be best measured by including the compile time or even the I/O times; for example, virtual machine code may be enough smaller than native machine code that the reduced I/O costs more than pay back for the cost of compilation and asymptotically slower code [Fraser et al. 84]; thus, some system measurements may start measurement before the code is even loaded into memory [Franz 94, Hölzle & Ungar 94a].

[3] The *last* use of the code because slower general-purpose code may be used while the faster code is being prepared.

10.3 Selected Uses of Code Modification

This section describes a variety of uses of code modification, organized by the size of the region being updated.

10.3.1 Changing Parts Of Instructions

A large number of systems modified parts of individual instructions, sometimes because code modification was the only possible implementation, but more often to improve performance. Early systems were built with a few hundred electronic switches and a few hundreds or thousands of words of memory. Thus, both simplicity and density of representation were important, albeit conflicting, goals. One way of coping was to push some complexity into software. For example, many computers used one part of an instruction word to hold an address; indirect addressing was implemented in software, by computing the indirect address and then storing it into the address part of the word.

Architectural Support for Instruction Patching Some architectures defined instructions for updating instructions. For example, the PDP-1 had dip and dap to modify the instruction and address parts of an instruction [Dig 62, Russell 95b, Greenblatt 95]; the Gier had both "address" and "count" fields in some instructions, and some instructions were defined to both read and update their count operand field [Simonyi 94]. The standard IBM 1401 procedure call patched the return instruction with the return address [Albaugh 95].

Execute Several machines defined an *execute* instruction, which "promoted" a data value to an instruction and then executed it [von Neumann 45, Brooks 60, Dig 62, Firestone 95]. An execute instruction is, in effect, a structured form of self-modifying code. It provides a guaranteed mechanism by which dynamically-formed instructions can be executed across a range of machine implementations. It thus allows some standard uses of instruction modification, while simultaneously avoiding reentrancy and instruction caching problems.[4] There have been many implementations of *execute*: some operated from memory [Brooks 60, Dig 62]; the IBM 360 operated from machine registers [Baker 95]; it and the Burroughs B1700 microarchitecture first or'd the instruction or microinstruction with a bitmask [Baker 95, Firestone 95]. The execute instruction was used for manufacturing new addressing modes (§10.3.1), fast simulation (§10.3.2), and for implementing parameter closures ("thunks") [Brooks 60]. On the Gier, PDP-1 and IBM Stretch *execute* behaved specially on control transfer instructions [Brooks 60, Dig 62, Simonyi 94, Eggers et al. 96], making simulation easier but limiting the generality of thunks [Baker 95].

Instruction Implementation Patching Some machines allow instructions to be changed, for example in order to fix bugs (see "Runtime Microcode Generation", pg. 136). The IBM 1620 implemented the add instruction using a lookup table in writable memory. One application sped up an inner loop by patching the add table so that the add instruction performed a different operation,

[4] At any rate, it at least redefines the complexities as purely data-space problems.

then re-patching the table to return it to a conventional add. The optimized loop was profitable even though it had added overheads from reorganizing the loop to avoid address calculations when the add instruction was redefined and from patching and restoring the table on every loop iteration [Gerson 62].

Opcode Patching The opcode of an instruction may be patched to reduce or eliminate control flow without replicating code. For example, the CDC 6400 had two memory segments, addressed with different instructions. Code that accepted arguments from either segment could have used static code and either dispatched on the segment type for each multi-segment memory reference or, for N parameters, used 2^N code duplication. Instead, the routine prologue patched the memory reference instructions to use the proper opcode type. Conditional branches have frequently been implemented as unconditional branches whose opcode is patched to be "branch always" or "branch never"; example applications include simulators, assemblers and operating system schedulers [Baumann 86, Eastlake III 95, Harrenstien 95]. As with bitblt transfer operations, the inner loops of graphics library operations may vary depending on the mode; these routines have thus been implemented by patching the inner loop [Breed 92, Cathey 92].

Opcode Toggling Some algorithms use alternate operations on alternate iterations. Where the code for the cases is similar — for example, they both use the same number of bytes — the loop can be built using straight-line code that is updated on every iteration by XORing the current instruction encoding with a *toggle mask* (§10.5.18), which is the XOR of the two encodings. For example, $sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$, is $s_{i+1} = s_i \pm \frac{x^{2i+1}}{(2i+1)!}$. Static code implementations could unroll the loop by two or use the parity of the iteration count to drive a conditional branch or result mask [Massalin 87, Granlund & Kenner 92]. Code patching has been used to alternate the inner loop operation between add and subtract [Henry 88].[5]

Operand Patching – Traversal Direction Programs produced by some versions of the Borland C compiler used operand patching to change table walk directions to implement forward table walks on program entry and a backwards table walks on exit [Gutmann 92].

Operand Patching – Manufactured Addressing Modes Some machines lacked particular addressing modes and instead implemented them using code modification. Examples include conditional branches for TX-0 [Gross et al. 95], and indirect addressing on the EDSAC [Wheeler 50, Gill 51], EDVAC [Knuth 70], URAL [Simonyi 94], IBM 650 [White 95] and other machines. On some machines, missing addressing modes were simulated a variety of ways, with code-modification techniques being most efficient. Examples include string copying with a runtime-determined length, using the IBM 360 move instruction [Baker 95] and I/O port operations on the Intel 8080 and early 8086 processors, where the port number was a runtime value [Henry 88].

[5] Code patching, as described here, was not profitable. In this particular case, a better solution was to use loop unrolling or a different algorithm such as $s_{i+1} = \frac{x^{2i+1}}{(2i+1)!} - s_i$, with i restricted to odd values.

Operand Patching – Faster Addressing Modes Sometimes, existing addressing modes were simulated more efficiently using code modification. Such collapsing was used with many machines and in many contexts. For example, the PDP-1 had both direct and indirect addressing, but indirect addressing took an extra cycle; in some circumstances, the indirection address was written directly to the instruction word, saving both a word of memory and the cycle needed for indirection [Russell 95b, Greenblatt 95]. Likewise, variables that were read by just one instruction were often stored as immediates in the instruction [Russell 95b, Greenblatt 95]. The CAL DATA 135 accessed microcode using efm microaddress. Dynamically-bound indirect microroutine calls required 3.2μSec; patching in the microaddress to form a direct call reduced the overhead to 1.0μSec [Ebeling 76]. Measurements of Self and Smalltalk-80 systems have shown that caching method receivers in instruction immediates rather than using an associated data word can provide up to a 30% performance improvement [Deutsch & Schiffman 84, Ungar et al. 84, Samples et al. 86, Ungar & Patterson 87, Hölzle 94]. Synthesis assigned trap numbers dynamically; time-critical callers were then patched to use the new trap number as an immediate [Pu et al. 88, Massalin 92a]. The 6502 microprocessor addressed memory outside of the low 256 bytes either by indirecting through a location in the low 256 bytes or by writing the address as an immediate in an instruction operand [Maurer 84]. The "virtual PC" for Microsoft Basic on some computers used operand modification [Albaugh 95]. Some compilers for the Burroughs 220, CDC 6400 and IBM 7094 implemented call-by-reference parameters by generating function prologues that patched the parameter addresses into memory reference instructions in the routine [Organick 72, Breed 92]. A common microcomputer "hack" was to spill register values to the immediate field of an later instruction which, when executed, would restore the value without an explicit load from memory [Cochran 92, Collier 92a, Gutmann 92].

Operand Patching – Simulating Segmentation Segmented memory has been implemented without hardware support, by using memory *overlays* (§10.3.5). When code was removed from memory, referencing instructions were patched to call the overlay driver; when the code was returned to memory, the instructions were again updated, so that they called the code at its new location [Spacek 72].

10.3.2 Patching Individual Instructions

Some systems replaced one instruction with another, typically a jump. The usual reason for using such code was to speed up indirection, as with "faster addressing modes", above.

Debugger Breakpoints Debuggers often set breakpoints by overwriting or *patching* an instruction to transfer control to the debugger. Debuggers such as dbx [Linton 90] and gdb [Stallman 87] run in a separate address space and use an operating system call that patches a given debuggee instruction with a *debugger trap* instruction.[6] The trap transfers control to the debugger, which evaluates the breakpoint condition and, if false, continues program execution. As shown in Figure 10.1, the debugger replaces the displaced instruction, patches the successor with a trap, and

[6] The debugger trap instruction must generally be as small as the smallest instruction.

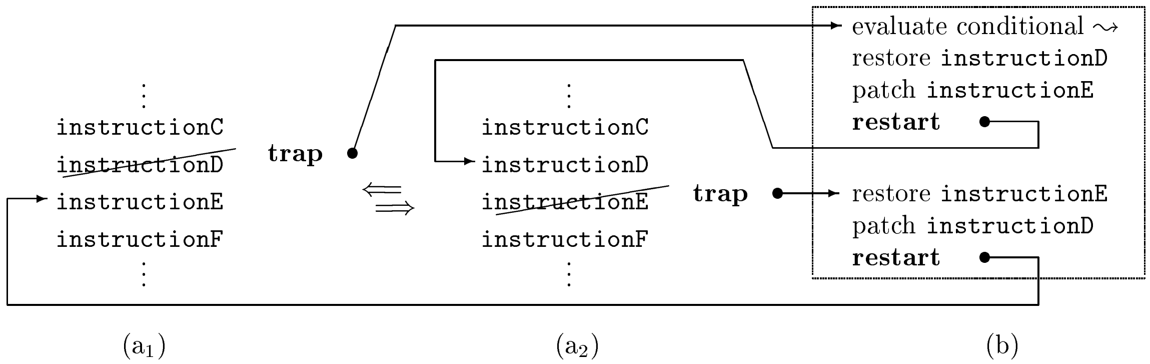


Figure 10.1: The operation of breakpoints based on trap patching.

Here, (a_1) and (a_2) are two versions of the same code. Initially, in (a_1), the debugger patches `instructionD` to trap to the debugger's breakpoint handler, (b). If the conditional is false, execution continues. The debugger executes `instructionD` by placing it back in the debuggee and patching its successor, as in (a_2). The debugger then restarts the debuggee at `instructionD`. The successor's trap causes the debugger to patch the code again to be like (a_1) and then restart the program at `instructionE`. All code patching is done via a machine-independent OS interface, so the debugger may be ported to any operating system that supports the patching interface.

restarts the application. The application executes a single instruction, then restarts the debugger, which re-patches the application. At least one profiler operates the same way [Bishop 87]. As shown in Figure 10.2, some debuggers operate partly in the same address space as the debuggee, and operate by patching instructions with branches instead of traps [Gill 51, Robinson & Owen 73, Aral & Gertner 88, Kessler 90a, Linton 90, Paxson 90b, Chase & Hood 91, Wahbe 92, Bedichek 92, Keppel 93a, Gross et al. 95]. The displaced instruction is typically executed out-of-line in order to avoid continual re-patching of the code [Kessler 90a, Keppel 93a, Greenblatt 95, Gross et al. 95]; thus, the debugger may be viewed as patching itself with the displaced instruction.

Dynamic Linking Dynamic linkers bind unbound routines; some also re–bind routines ("Dynamic Linking", pg. 141). Many architectures have a direct procedure call form that is faster than an indirect procedure call. Therefore, some dynamic linkers have been implemented to patch the caller to use the faster direct-call form. For example, Microsoft Windows compiles calls to virtual device drivers as traps; on first execution, the trap instruction is rewritten as a call to the appropriate code [Gutmann 95].

Fast Conditionals Slow-changing multi–cycle conditional branches have been implemented as faster unconditional "branch always" or "branch never" (`nop`) instructions that were patched when the branch direction changed [Baumann 86, Eastlake III 95, Harrenstien 95]. Likewise, demand recompilation may be driven by a having the dynamically-generated code test a status variable. The test may instead be performed by the code that changes the status variable; if the test evaluates to

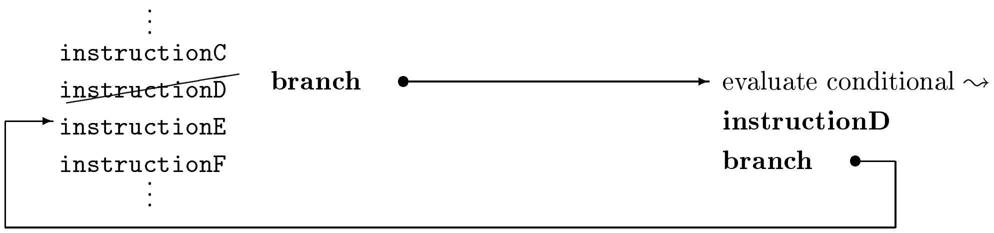


Figure 10.2: The operation of debugger breakpoints based on branch patching.

The debugger overwrites `instructionD` with a branch to the debugger's breakpoint handler code, which is mapped into the debuggee. The debugger evaluates the conditional; if false, execution continues, and the debugger evaluates the displaced instruction, `instructionD`, out-of-line. Some instructions are evaluated out-of-line by patching them into the debugger. Sometimes, out-of-line execution requires extra work; for example, for position-dependent instructions. Finally, the debugger branches to the successor of the displaced instruction.

true, that code also overwrites the first instruction of the dynamically-compiled code so that when the dynamically-generated code is invoked, it branches to the recompilation machinery. The latter implementation has been used in both Synthesis [Massalin 92a] and Shade [Cmelik & Keppel 94] (see §B.3).

Instrumentation Instrumentation tools have been implemented to patch individual instructions and classes of instructions with jumps to "arbitrary" code that typically recorded tracing information, simulated the instruction, and then restarted the instruction's successor [Deutsch & Grant 71, Litman 89].

Language Implementation The COBOL ALTER verb was often implemented by patching the labeled goto with the address of the ALTER target [Russell 95c].

Link Snapping Link snapping transforms an indirect reference into a direct reference by performing the indirection once and then saving the computed result.[7] Various systems have implemented link snapping using code patching. For example, Maclisp call targets could be either interpreted or compiled code; compiled callers initially invoked code that resolved the callee implementation and then updated the caller to invoke the interpreter or the callee as needed [Moon 74]. Procedure initializers that are executed only on the first invocation of a routine have been implemented similarly: the routine prologue branched unconditionally to the initializer and the initializer overwrote the branch with a nop or with the "real" first instruction of the routine [Litman 89]. Likewise, LISP callees have checked parameter types on a caller-by-caller basis, by patching particular callers so that subsequent calls from that site would skip the checking prologue [Raymond 93]. Finally,

[7] Caching is a generalization of link snapping, where much of the indirection is removed by computing and saving the general result and then checking it quickly. Thus, some uses of instruction space modification may be viewed as extensions to link snapping rather than special cases of caching.

link snapping has been used to lazily excise calls to debug routines, by having the debug routine patch it's callers, also recording the snapped sites, so that they could be unsnapped if the callee changed [Moon 74, Gosling 95].

Same-Machine Emulation Several instruction-set simulation and tracing tools, including some debuggers ("Debugger Breakpoints", pg. 120) simulated instructions by patching each target instruction into a simulator loop and executing it there, with pure simulation or instruction fixups for PC-altering and PC-relative instructions [Gill 51, Hsu 89, Kessler 90a, Cmelik 90, Chase & Hood 91, Wahbe 92, Bedichek 92, Gross et al. 95]. In some cases, the resulting code was quite small; for example, the entire EDSAC emulator was twenty-six machine instructions plus five words of data [Gill 51]. Some machines avoided inner loop patching by instead using an "execute" instruction ("Execute", pg. 118).

Threading Several implementation styles, such as threaded code [Bell 73, Eckhouse Jr. & Morris 79, Kogge 82, Miranda 87, Bedichek 89] and combinator reduction [Turner 79, Hughes 82] link together code fragments, so that control passes directly from one to the next instead of following a strict call/return discipline. Such "threading" has been implemented various ways. For example, Figure 10.3(a) shows traditional threaded code, where each code fragment ends with a dispatch operation that increments a thread pointer, reads an address from the thread and jumps to the address. When a pair of direct branches were cheaper than an indirect branch, the "thread" addresses was patched with jump instructions [Thompson 68, Kelly 89, Koopman 89, Koopman & Lee 89, Massalin & Pu 89, Santi 89, Koopman et al. 92, Massalin 92a], as in Figure 10.3(b). Another variation is *chaining*, shown in Figure 10.3(c), which uses direct branching from fragment to fragment. It has been used where execution of one fragment *always* follows another [Cmelik & Keppel 93, Cmelik & Keppel 94, Bedichek 95a]. These techniques have been generalized to arbitrary graphs representing, e.g., finite state machines ("String and Regular Expression Searching", pg. 137).

The various techniques have also been combined. For example, Shade used general threading that was then link snapped to chain the predecessor to its successor [Cmelik & Keppel 93, Cmelik & Keppel 94, Bedichek 95a]. Chained fragments were unlinked (disconnected) lazily: when a particular predecessor/successor relationship ended (see "Link Snapping", pg. 122), the successor was overwritten with code that would lazily unchain the predecessors, which was in turn lazily re–chained to a valid successor using link snapping.

List and graph structures have also been combined with the engine that traverses them, resulting in a form called *executable data structures* (§10.5.8). For example, Figure 10.4 shows a linked list of free memory in which each cell encodes the behavior of the next allocation request.

Unimplemented Instructions Some processors do not implement all of the instructions defined for the architecture family, and executing a "missing" instruction causes a trap. In a variety of systems, the trap handler has been implemented so that it patches the program, replacing the missing instruction with a jump to code that implements the missing instruction. For example, this

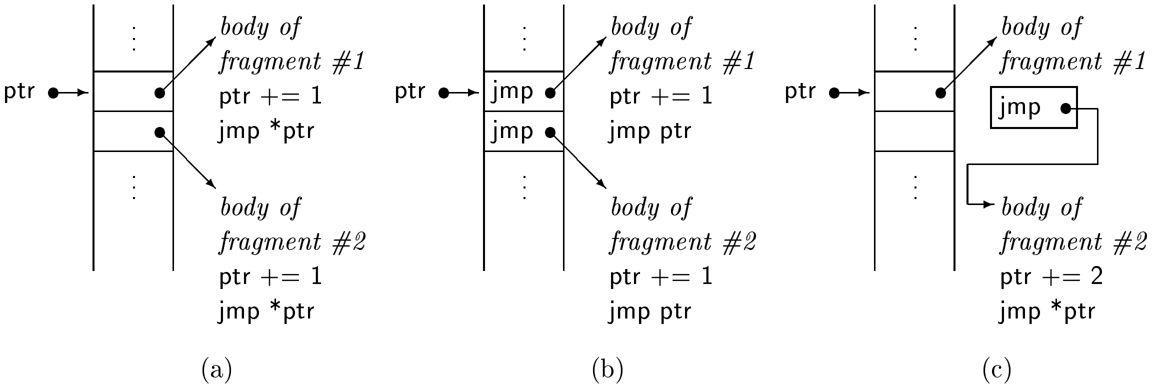


Figure 10.3: Threaded code implementations.

Threaded code is traditionally implemented using indirect jumps, as in (a). It may also use jumps to jumps, (b), or use direct chaining, (c).

technique has been used for floating-point instructions on programs compiled for the IBM PC/RT, Intel 80386 family [Chase et al. 95], Motorola 68000 family [Sloan 90] and SPARC processors [Cmelik 94].

10.3.3 Small Instruction Groups

Bootstrap Loaders Early machines were initialized at power-on by entering instructions on console switches or by having very simple hardware read and execute instructions from an external device. Since bootstrapping machinery was useful only for starting the machine, bootstrap hardware was minimal. However, bootstrap loaders were size-constrained because of the time it took to, for example, enter one by hand using toggle switches. Thus, any hack that reduced the code size, including self-modification, contributed directly to faster boot times. For example, the PDP-6 bootstrap loader size was cut by more than half, in part by using self-modifying code [Greenblatt 95]. In addition, bootstrap loaders have typically been simple programs that loaded more sophisticated bootstrap sequences that, in turn, loaded and started the operating system. Thus, the storage used by the bootstrap loaders was ultimately overwritten by other code and data.

Closures Interlanguage and curried calls may bind some parameters before routine invocation. Effectively, the routine is specialized to one accepting fewer parameters. One implementation technique is to generate code fragments that insert the early-bound arguments into the argument list and then call the real routine. For example, the functional programming system described by Kelly implements the curried call f(x,y) with y=20 by generating (approximately) the code fragment $\mathsf{f}_{\mathsf{y=20}}(\mathsf{x})$ {f(x,20)} [Kelly 89]; more generally, each closure generation can produce either a tuple of "code pointer, data pointer" or a newly-compiled code fragment that embeds the data value [Kelly 89, Feeley & Lapalme 92]. Similarly, C routine callbacks are of the form f(a); a C++ or Modula-3 callee

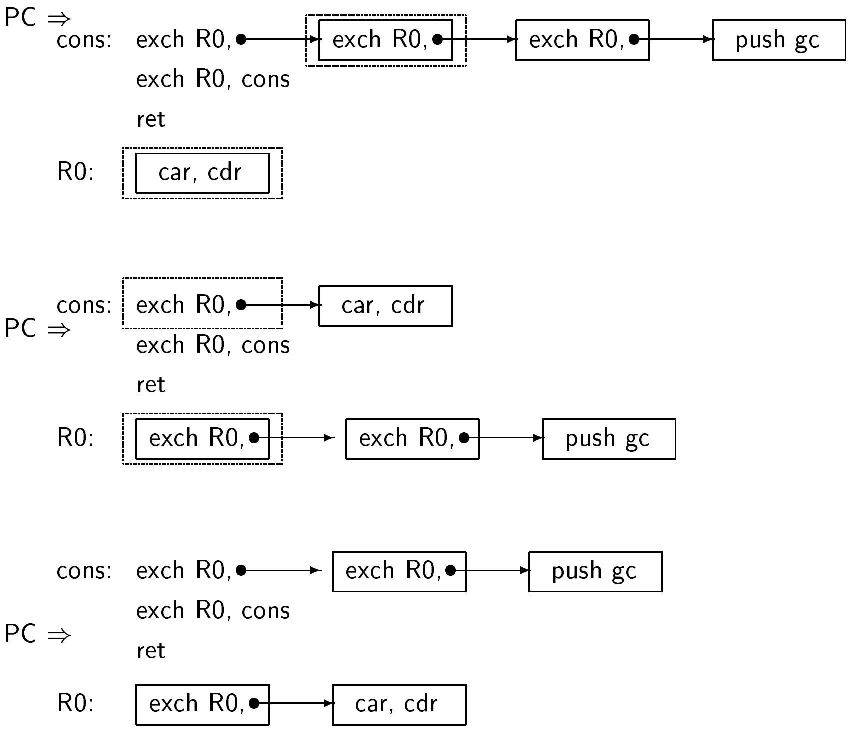


Figure 10.4: A self-modifying CONS cell memory allocator.

Self-modifying code at its best. The routine cons removes a cons cell from the storage free list, fills it with its new value, and conditionally calls the garbage collector. The whole routine is two "exchange register, memory" instructions and a procedure return, plus each element in the free list contains an instruction that replaces one instruction of the allocator, describing how the subsequent allocation request should be serviced. In each code fragment, the symbol "PC ⇒" shows the instruction that is about to be executed and the dashed box shows the storage that will be updated (not shown for the ret instruction). Here, the free list has three elements. On entry to cons, the register R0 holds the value to write in the allocated memory; on exit, R0 holds a value that, when used in a data context, is a pointer to the newly-allocated memory. No other registers are needed in normal execution. Normally, the replacement instruction from the free list updates the current list head. The last element's push gc instruction, however, updates cons so that a subsequent call to cons will push the address of the garbage collector and the return instruction, ret, will invoke the garbage collector.

will expect an extra self argument, which is implemented by having the C code call a stub f(a) { f(self, a) }, where self is bound when the stub is created [Breuel 88, Chase 91, Dig, Chase 95a]. Static code implementations are generally possible only where all routine type signatures can be deduced statically; otherwise, the static mechanisms may introduce new failure modes [Breuel 88].

Context Switching Operating system context switches have used RTCG in several ways. First, some architectures must save a register to a global before loading the address of the current thread's

```
save_thread:
        mov     r0 → tmp        /* spill */
        ld      curr → r0
        movr    <r1..r16> → 4+REGS(r0)
        mov     tmp → r1        /* reload */
        mov     r1 → REGS(r0)
        call    dispatcher      /* dispatch */
restore_thread:
        st      r0 → curr
        movr    REGS(r0) → <r0..r16>
        rei
```

(a)

```
save_thread_1:
        movr    <r0..r16> → thread_1+REGS
        jmp     restore_thread_2        /* dispatch */
restore_thread_2:
        st      thread_2 → curr
        st      thread_2_ivec → ivec /* new switcher */
        movr    thread_2+REGS → <r0..r8>
        rei
                    ⋮
save_thread_N:
        movr    <...> → thread_N+REGS
        jmp     dispatcher              /* dispatch */
```

(b)

Figure 10.5: Thread context switching on an interrupt.

In (a), all registers are in use, so one must be saved to a memory temporary in order to discover the address where the registers should be saved; the instruction movr saves and restores blocks of registers. In (b), the save and restore code is replicated and each instance is specialized to the particular thread, including the location of the thread save area, the number of registers used by the thread, and per-thread dispatching decisions. In (b), the general dispatcher is called only periodically; most threads are chained (§10.3.2) directly. Replicating the context switch code means changing interrupt dispatch on every context switch, as indicated in (b) by the store to ivec on each switch.

save area. Thus, the save area address was embedded in an instruction immediate, which saved both the load of the address and the save and reload of the register that would otherwise be needed in order to hold the save address. Second, using thread-specific context switch code allowed per-thread optimizations based on the thread's register usage; for example, floating-point register saves were omitted for threads when they were using only integer registers, and threads that were executing particularly "lightweight" inner loops could use context switch code that saved and restored just a few registers. Finally, where a group of threads typically executed in sequence, the general-purpose dispatcher was specialized to a thread and then inlined directly in the per-thread context switch code. These thread-specific optimizations were all used in Synthesis, allowing a general-purpose mechanism to still provide low overheads for threads that didn't need full generality [Pu et al. 88, Massalin & Pu 89, Felten et al. 92]; they are shown schematically in Figure 10.5.

Dynamic Linking Some dynamic linkers have implemented dynamic linking by compiling static code to call a short sequence of instructions called a *dynamic link table entry* or *DLTE*. Typically, the DLTE was initialized to jump to the dynamic linker, which would find, map and initialize the call target. The dynamic linker then updated either the DLTE [AT&T 90a, AT&T 90b, Keppel & Russell 93] or data that it referenced [Organick 72, Shaw 74], so that subsequent jumps to the

```
caller:   push   arg0
          call   DLTE07

                 (a)                     (b)                       (c)                       (d)

DLTE07:   set    DLTE07 → r0     set    DLTE07 → r0       set    DLTE07 → r0       set    DLTE07 → r0
+4:       br     dylink          br     dylink            sethi  %hi(foo) → r0     sethi  %hi(foo) → r0
+8:       nop                    jmp    r0+%lo(foo)       jmp    r0+%lo(foo)       nop

dylink:   ... find and install foo ...
          or     "jmp r0+%lo([?])", %lo(foo) → r2          /* Form jump to foo */
          st     r2 → DLTE07+8                             /* Install */
          call   coherency DLTE07+8                        /* Propagate to instruction caches */
          or     "sethi %hi([?]) → r0", %hi(foo) → r2      /* Setup for jump to foo */
          st     r2 → DLTE07+4                             /* Install */
          jmp    r1                                        /* Jump to foo */
```

Figure 10.6: A dynamic link table entry (DLTE) being updated.

Initially, (a), the DLTE invokes the dynamic linker. During update, (b), the entry is still a valid call to the dynamic linker, so that a concurrent call will see a valid instruction sequence and call the dynamic linker. In the final form, (c) the DLTE entry has been updated to call the dynamically-linked routine. The dynamic linker must perform instruction cache coherency, "call coherency *addr*", in order to ensure that the jmp instruction appears in the instruction stream before the sethi. Otherwise, the sequence (c) could be written to memory but the invalid sequence (d) might be executed from the cache. Coherency for writing the sethi is optional. With coherency, sequence (c) replaces (b) sooner in the caching structures; without it, the only penalty is that the slower form (b) is sometimes executed.

DLTE would invoke the dynamically-linked callee. Problems arose when multiple DLTE instructions were updated, because interrupts and multithreading caused DLTE invocation during the update. Thus, any partially-updated sequence needed to be legal [AT&T 90b, Keppel & Russell 93]. Figure 10.6 shows an example of a DLTE being updated during dynamic linking. Unlinking and relinking [Kendall 89, Segal & Frieder 89a, Segal & Frieder 89b, Ho & Olsson 91, Frieder & Segal 91, Segal 92, Segal & Frieder 93, Sirer et al. 96] have similar atomicity/race requirements [Keppel & Russell 93, Sirer et al. 96]; as a result, some systems overwrote the "unlinked" code rather than performing a complete unlink/uninstall [Collier 92a]. Static code alternatives have included indirect procedure calls, which require interprocedural optimization to avoid slowing the common case of statically-linked calls [Gingell et al. 87], and load-time linking [Maeda 90, Kempf & Kessler 92a, Kempf & Kessler 92b, Nelson & Hamilton 93a, Nelson & Hamilton 93b], which only works for some of the cases handled by more general dynamic linking.

Invocation-Time Dynamic Inlining A callee may inline itself at call time, so that subsequent calls avoid the out-of-line call. For example, the CDC 6400 used library routines named X after machine instruction X; each time routine X was called, it patched its caller with instruction X. Like techniques were used with Burroughs 220 and IBM 7090 [Breed 92]. The Shade dispatcher dynamically inlined simple dispatching operations by replacing a call to the dispatcher with a jump to the dispatch target; inlining thus implemented chaining [Cmelik & Keppel 94]. See also "Inlining at Dynamic Compile Time", pg. 133.

Software-Controlled Caching Some systems use a small amount of fast memory and a larger amount of cheaper but slower memory. Performance is improved by dynamically copying a loop to fast memory and executing it there, effectively implementing software-controlled caching. For example the inner loop of the PDP-6 or PDP-10 string move routine was implemented this way [Greenblatt 95, Stallman 95, White 95]. Similarly, the Mfast processor uses a high-bandwidth on-chip memory for executing VLIW code: "[Loading] the VIM (VLIW Instruction Memory) ... is costly, but it can be done in the background while the PE is executing other VIM-resident functions or, for the tricky programmer, from locations that are being modified on the fly." [Epstein 95].

Trap Vectors *Traps* cause a machine to switch to supervisor mode and jump to a location named by an indirect table. The indirect table may be an array of addresses or an array of instruction fragments [Sun 91, SPA 92]. Operating systems that support dynamically-bound traps have bound the traps by updating the indirect table [Hamilton & Kougiouris 93, Kessler 93, Bershad et al. 95].

Signal Trampolines Some operating systems provide an abstraction of hardware interrupts called *signals*. Part of the machinery for signal handling saves the user-visible processor registers before signal delivery and restores them after the signal handler returns. Under early versions of the 4BSD variant of Unix, the registers were restored by arranging the signal handler to return to cleanup code that the kernel had placed on the stack.

10.3.4 Medium Code Blocks

Abstraction Embedding Many programs manipulate concrete data types. An alternative is *abstraction embedding*, in which an application is written to manipulate data whose type and structure are unavailable statically. Instead, the code is manipulated by way of an abstract interface whose implementation is associated with the data [Stemple et al. 92, Winroth 92, Stemple et al. 93, Wahbe et al. 93, Com 94, Piersol 94, Pountain 94, Franz 94, Nelson 95, Johnston 95a, Johnston 95b, Johnston 95c, Nov 95]. Systems that use abstraction embedding have used dynamically-incorporated machine code, because the two pure static-code alternatives have severe problems: using virtual machine code would cause interoperability problems (see §10.5.5) and using machine code that was executed in another process would make data sharing dramatically harder. Additionally, the overhead of interpreting or interprocess control transfers could hurt overall application performance.

Cache Optimizations In the physically-indexed caches of an experimental DEC system, cache "hot spots" were discovered via profiling and were eliminated by moving code to different physical pages [Kehl 93]; the same idea was also suggested elsewhere [Stone & Cocke 91].

Compiling Interpreters An *interpreter* is an engine that executes a program without *explicitly* performing translation of the program. Some systems have implemented interpreters by compiling machine code dynamically. Such *compiling interpreters* have been used for a wide variety of systems, such as translators for high-level languages [Brown 76, Dyke & Bree 77, Guibas & Wyatt 78, Johnston 79, Saal & Weiss 79, Ford & Sawamiphakdi 85, Christopher & Wallace 86, Sloman 88, Vegdahl & Pleban 89, Wakeling 95, Ltd. 96], including special-purpose languages for databases (see "Databases", pg. 131), debugging [Kessler 90a, Ramsey & Hanson 92, Sun 94], graphics manipulation [Holzmann 87, Fitzhardinge 93, Fraser 93], numerical mathematics [Littlefield 90] and "desktop calculators" [Engler et al. 95, Engler et al. 96, Auslander et al. 95, Auslander et al. 96], and string and regular-expression matching (§10.3.4); for fast implementations of virtual machines that execute a portable code representation [Deutsch & Schiffman 84, Hölzle et al. 91, Hölzle & Ungar 94b, Chambers 92, Pountain 94, Hölzle 94, Col 95, Gosling 95, Lewis et al. 95, Jav 96]; and for instruction-set simulation and tracing tools [Deutsch & Schiffman 84, Lang et al. 86, May 87, Nielsen 91, Magnusson 93, Veenstra 93, Cmelik & Keppel 94, ARD 94, Veenstra & Fowler 94, Bedichek 95a, Pittman 95, Rosenblum et al. 95]. There have also been hardware equivalents in which microcode is generated dynamically from machine code [Ditzel & McLellan 87, Ditzel et al. 87a, Colwell et al. 88, Pow 94, Gwennap 95c].

There are a variety of reasons for using dynamic compilation to implement interpreters:

- An implementation based on dynamic compilation may be faster than one based on interpretation [Deutsch & Schiffman 84]: processors have special hardware for fetching, decoding and dispatching; compilation allows operations to be expressed in a form close to that of the machine; and general operations may be readily specialized to the specific context in which they appear.[8] In addition, compiling interpreters have used adaptive reoptimization (§10.5.1) in order to apply sophisticated optimization only to heavily-used portions of the application.

- Dynamic compilation may produce code that is faster than code produced by static compilers [Mitchell 70, Deutsch & Schiffman 84, Stonebraker & Rowe 85, Carey et al. 88, Chambers 92, Hölzle 94, Cmelik & Keppel 94, Hölzle et al. 91, Hölzle & Ungar 94b, Stemple et al. 92, Stemple et al. 93, Krall & Berger 94, Dillenger 95, Gosling 95, Jav 96, Krall & Berger 95b, Krall & Berger 95a]. RTCG improves the performance of interpreters for the same reasons that VSO works: static optimization is limited by specialization-induced code growth and by the accuracy of predicted behavior [Wall 90, Wall 91]. For example, observed runtime types, values, and virtual machine instruction combinations may be used to generate highly-specialized code

[8] Interpreters typically have comparable performance if the cost of each primitive is large. For example, PostScript™ [Ado 85] has many primitives such as "draw filled polygon", each of which takes a long time to execute compared to the cost of interpreter dispatching. In contrast, the interpretation overhead is relatively large for programs that are composed mostly of operations such as "add" and "if-then-else".

without the space explosion that would accompany specializing around all types or instruction triples that might arise (see, for example, "Type-Based Dynamic Optimization", pg. 139; "Value-Based Dynamic Optimization", pg. 139). In addition, some compiling interpreters over-optimize and adapatively reoptimize as needed (§10.5.1).

- Dynamic compilation may be more space-efficient than static compilation but faster than interpretation [Brown 76, Wakeling 95]. For example, in the LML/X-code system, interpreter code sizes were roughly one-quarter the size of static machine code, but interpretation was estimated to be about one quarter the speed. A compiling interpreter was able to deliver three-quarters the performance of statically-compiled code even when both code and data were allocated from a small heap [Wakeling 95].

- Dynamic compilation may be more flexible than static compilation. First, dynamic compilation allows both the code and types to be modified; static compilers have difficulty with both. Second, statically-compiled machine code is often large, nonportable, and difficult to check (§10.5.15). In comparison, "compiling interpreters" can execute compact, portable and checkable code representations; and by supporting dynamic change of representation, can improve debugging, heterogenous migration, and other applications that manipulate or incorporate code forms [Deutsch & Schiffman 84, Comer 87, Hölzle et al. 92, Col 95, Gosling 95, Lewis et al. 95, Jav 96], (§10.5.5). Finally, some systems use dynamic compilation to make the common case fast while still handling "unexpected" situations such as breakpoints and signal, by dynamically compiling code in response to the event [Comer 87, Hölzle et al. 92, Cmelik & Keppel 93, Hölzle 94, Bedichek 95a].

- Dynamic compilation allows dynamic inlining of instrumentation for both profiling and tracing, allowing the instrumentation to vary dynamically, which, in turn, has been used to solve a variety of sampling problems [Cmelik & Keppel 94, Bedichek 95a, Rosenblum et al. 95] (see "Parallelism-Based Dynamic Optimization", pg. 134).

- A dynamic compiler can emulate an interpreter. Thus, in systems where a compiler is a prerequisite, making it available for interactive compilation eliminates the need for the interpreter. A dynamic compiler may thus simplify the implementation, improve performance of sophisticated interactive commands and eliminate inconsistent behavior between the interactive and batch translators [Brown 76, Steel Jr. 82, Goldberg & Robson 83, Vegdahl & Pleban 89].

In addition, switching to compilation may improve error reporting over an ad-hoc interpreter. For example, the XTEC version of TECO was rewritten to compile user commands to machine code. The goal was improved performance, a side-effect was an improved user interface. In particular, forgetting an "I" (insert) before a string caused the interpreter to execute the input instead of inserting it, often destroying the document. The compiler version parsed the entire line before executing any of it, and would thus detect such errors [Crispin 90]. Improved error reporting is possible with interpreters and is not intrinsic to compiling interpreters, but it is nonetheless an *observed* benefit.

Constraint Programming A system of constraints may be created at runtime. In one Prolog compiler, constraints were compiled on introduction [Krall & Berger 94, Krall & Berger 95b, Krall & Berger 95a]; in the ThingLab interactive constraint solver, constraints were compiled on invocation [Borning 87, Freeman-Benson 90]. In both cases, the result was a set of routines that evaluated varying inputs to the given constraint configuration.

Databases A major component of many databases is an engine that executes user queries. A variety of systems have compiled user queries to machine code in order to improve performance [Chamberlin et al. 81a, Chamberlin et al. 81b, Cheng et al. 84, Stonebraker & Rowe 85, Carey et al. 88, Stonebraker et al. 90, Wilkinson et al. 90, Shipley 91, Kerns 92a, Stemple et al. 92, Stemple et al. 93]. Some of the motivations for using RTCG are discussed under "Compiling Interpreters" (pg. 129); in particular, some databases have supported extensible types, and queries themselves are user-provided code. Note that many query optimizations are for data placement and do not involve code optimization [Ceri & Pelagatti 84]; papers on query compilation to machine code have typically been vague about how machine code generation affects performance.

Debugging A variety of debuggers inject debugger code into the instruction space of the debuggee, so that debugger code can be executed without expensive interprocess communication [Aral & Gertner 88, Litman 89, Kessler 90a, Paxson 90b, Chase & Hood 91, Paxson 90a, Bedichek 92, Wahbe 92, Keppel 93a, Gross et al. 95], (§4.1.5). Updating the debuggee directly may improve performance, reduce dependence on operating system support, and reduce the debugger's timing intrusion in a multithreaded system. Some systems have also dynamically recompiled debuggee code in order to simultaneously support optimization and debugging [Comer 87, Hölzle et al. 92].

Device Drivers Some early operating systems allowed dynamic linking of device drivers. For example, one tape drive supported a wide variety of formats by putting format-specific machine code onto the beginning of each tape [Kay 88].[9] Most recent operating system kernels have required device drivers to be linked statically before startup. Apollo's Domain operating system supported device drivers that were added dynamically, but code space is not reclaimed when the device driver was discarded [Bianco 93]. Thus, it allowed incremental development of device drivers but punted the more general storage reclamation needed for a long-running system. The Linux operating system was designed and implemented so that peripherals could be added and removed during operation, and thus device drivers were bound and unbound dynamically and automatically [Hinds 95, Ward 95].

Dynamic Version Updating Sometimes it is better to update a running program than to halt the system, modify the program and then restart. Systems that have used dynamic version updating include interactive programming and debugging environments [Mitchell 70, Deutsch & Schiffman 84,

[9] Alan Kay's formulation of object-oriented programming was inspired from the tape drive code [Kay 88]; thus, it might be said that object-oriented programming is an offshoot of self-modifying code.

Comer 87, Craske 91, Hölzle et al. 92, Hölzle 94, Sun 94], heterogenous process migration [Deutsch & Schiffman 84, Theimer & Hayes 92], and updating of systems where shutdown causes unacceptable service interruptions [Segal & Frieder 89a, Segal & Frieder 89b, Frieder & Segal 91, Segal 92, Segal & Frieder 93]. Complications arise in these systems when an update that has requested an "older" version of a code fragment is still being executed or is suspended along a call chain. Solutions have included deferring update until there are no "live" versions, updating each live copy before it is allowed to continue, or mixing versions; additional complications occur in environments with distributed processing elements or distributed administration [Segal & Frieder 89b, Segal 92].[10]

Extensible Systems An *extensible system* provides a mechanism by which user-supplied code is called in response to particular system actions, thus enabling a user to extend a system by programming it [Deutsch & Grant 71, Gaines 72, Gifford 81, Stallman 86, Call et al. 87, Falcone 87, Notkin & Griswold 87, Stallman 87, Notkin & Griswold 88, Stamos & Gifford 90, Rodriguez Jr. 91, Yokote 92, Cmelik & Keppel 94, Bedichek 95a, Bershad et al. 95, Campbell & Tan 95, van Dorn et al. 95, Lea, Jr. et al. 95, Hsieh et al. 96, Sirer et al. 96]. For example, performance analysis of the operating system of the SDS-40 used user-supplied machine code that was verified and inserted into the running kernel [Deutsch & Grant 71]. From the standpoint of the extensible system, the extension code is often treated as mobile code (see "Mobile Code", Section 10.5.15).

Games A folklore programming challenge is to write a program that, when executed, clears all memory, including the program itself and all registers including the stack pointer and the program counter [Aitchison et al. 92].

Graphics Bitblt is a standard example of medium-grained RTCG; it is explained in detail in Section 4.1.1.

Heterogenous Process Migration Heterogenous process migration is when an executing application is halted and restarted on another machine of a different type. Some implementations have used statically-compiled machine code with one version of the code for each type of migration target [Shub 90, Bank et al. 94]. However, migration was limited to platforms for which code was precompiled. Using virtual machine code offers better portability but may suffer worse performance [Shub 90]; RTCG has been used to provide faster virtual machines (see "Compiling Interpreters", pg. 129). Another difficulty in heterogenous process migration is updating data representations [Bank et al. 94]. One heterogenous migration system dynamically generated machine code that mapped from the state organization on the old machine to the state organization on the new machine [Theimer & Hayes 92].

[10] Program and implementation changes may also be viewed as an extended copying garbage collection problem. For example, program modifications are expressed as movement between version spaces, process migration is movement between implementation spaces and physical location spaces, and so on. The intuition in this view is that representational transformations and dependencies may be managed in a regular way much as pointer values and their referents are tracked and updated in a copying garbage collector.

Inlining at Dynamic Compile Time Several systems have performed full or partial inlining at dynamic compile time. One strategy is to inline all code that meets a fixed criteria. For example, Shade inlined all tracing code except for user-supplied tracing code [Cmelik & Keppel 93, Cmelik & Keppel 94]; a similar strategy was proposed for implementing network protocols [Abbott & Peterson 92] and was used to implement constraint solver planning [Freeman-Benson 90]. A second strategy is to perform inlining using compiler analysis and/or profiles to determine profitable sites. A large number of language translation systems have used this approach [Hansen 73, Jones 88, Hölzle et al. 91, Hölzle & Ungar 94b, Chambers 92, Blair 93, Cutts et al. 94, DeHon & Eslick 94, Hölzle 94, Brown 95a, Brown 95b]. A third strategy is to invoke each call target at dynamic compile time, when the *caller* is being compiled. each callee then returns "code" for the call, which can be an out-of-line call, inline code for the body, or some hybrid [Chow 88] that inlines common paths but implements uncommon paths as out-of line calls (see also §10.5.7 and "Invocation-Time Dynamic Inlining", pg. 127). For example, Synthesis defined the standard quaject interface to use "bind-time" entries that would return code to inline in the caller; that code would transparently implement inlining, an invocation-time call, or some hybrid [Massalin 90].

Interactive Programming Systems Interactive programming systems allow users to provide code that is incorporated into a running system, typically in order to provide fast edit-debug turnaround. A variety of systems have used compiling interpreters ("Compiling Interpreters", pg. 129) and/or batch-style compilers whose output was then dynamically linked ("Dynamic Linking", pg. 141) [Brown 76, Dyke & Bree 77, Guibas & Wyatt 78, Johnston 79, Chamberlin et al. 81a, Deutsch & Schiffman 84, Pollock & Soffa 84, Ungar et al. 84, Christopher & Wallace 86, Holzmann 87, Goodman 88, Goodman et al. 89, Fyfe et al. 91, Johnson & Fox 91, Shipley 91, Chambers 92, Hölzle 94].

Linking Loaders Linking loaders read a set of unlinked (or incompletely-linked) modules and produce an executable core image that is then executed [Wheeler 50]. Linking loaders for early systems were similar to later load-time dynamic linkers [Maeda 90, Kempf & Kessler 92a, Kempf & Kessler 92b, Nelson & Hamilton 93a, Nelson & Hamilton 93b], but there was no clear notion of "application runtime"; linking was completed before the first program instruction was executed, but the transfer from loader to program was accomplished with a simple jump.

Load-And-Go Systems A variety of systems have stored code in source or intermediate form and then dynamically compiled the code automatically, as a part of program execution, or when an explicit "generate compiled version" command was issued. These systems have all used fast compilers, and have garnered some performance benefits from small I/O times due to the compact representation. Some systems have also used adaptive reoptimization (the more you use it, the more it gets optimized, §10.5.1), which has provided both fast startup and fast execution. Load-and-go systems have used compiling interpreters ("Compiling Interpreters", pg. 129) but may also perform all code generation on startup. Examples have included BASIC [Brown 76, Koch & Walker 95], Self [Chambers 92, Hölzle 94] and Smalltalk [Deutsch & Schiffman 84].

Logic Simulation Switch-level models of digital circuits can model both digital and analog behavior components and certain kinds of faults. The HSS logic simulator [Barzilai et al. 87] and a system described by Bryant [Bryant 87] transformed an input circuit to machine code that was then executed. In HSS, the generated code was further patched to simulate particular faults (§10.5.16).

Network Routing Tables It has been alleged that a network router embedded current routing decisions in the microcode for a silicon switch engine, thus reducing the time needed to route a particular packet. When the routing information changed, a coprocessor generated new microcode and installed it in a timely way.

Operating Systems Operating systems are prime candidates for runtime optimization because they must be general to support a wide variety of applications, cannot easily be specialized by a particular user or application, and often run with many slow-changing values that are relatively constant during each processes' lifetime. A wide variety of operating systems have thus used dynamic specialization of interfaces, implementations or both [Gaines 72, Falcone 87, Pu et al. 87, Massalin & Pu 88, Notkin & Griswold 88, Pu et al. 88, Massalin & Pu 89, Stamos & Gifford 90, Massalin & Pu 91, Kiczales 92, Massalin 92a, Consel et al. 93, Pu & Walpole 93, Thekkath & Levy 93, Montz et al. 94, Pu et al. 94, Berg et al. 95, Bershad et al. 95, Auslander et al. 96, Chambers et al. 96, Engler 96, Leone & Lee 96b, Leone & Lee 96a]. The IBM System/38 and OS/400 systems treated all user code as mobile code (§10.5.15), and system security relied on the fact that only the operating system was allowed to generate native machine code [Berg et al. 95].

Parallelism-Based Dynamic Optimization Dynamic change of tracing level has been used to parallelize tracing and analysis of sequential programs. Although dynamic change of tracing can be emulated with static code by performing full tracing and then discarding unneeded information, actually changing the tracing level gives better performance, since execution is slowed by tracing only when useful trace data is actually being collected [Sosič 92, Cmelik & Keppel 93, Cmelik & Keppel 94, Sosič 94, Bedichek 95a, Rosenblum et al. 95]. Dynamic change of tracing level has thus allowed tracing to be divided into segments so that segment i is traced on on processor i. For each processor, segments $0..(i-1)$ are run without tracing in order to "fast forward" to the correct execution state. Improved untraced execution performance translated directly into improved parallel speedups [Cmelik & Keppel 93].

Placement-Based Dynamic Optimization The Concert system proposed dynamic optimization based on the current placement of data; it would have replaced general remote accesses with direct local accesses [Chien et al. 93, Chien 95]. Note, that many databases have optimized around data placement, but the optimizations were typically used to control the order of application of operations; it is unclear whether the generated machine code was optimized based on placement (§10.3.4).

Portable Binaries Normal program binaries will run on a modest number of machines but are not portable across machine architectures; a *portable binary* is one that uses virtual machine code (which has sometimes been a designated native machine code). One strategy is to explicitly install the program and convert the virtual machine code to native machine code at that point [Dahlby et al. 78, Koldinger 88, Sites et al. 93, Berg et al. 95]. Installation-time conversion, however, requires a separate copy of the program for each target in a heterogenous network. It also cannot be applied to code for which there is no explicit installation time (see §10.5.15). A second strategy is to interpret the virtual machine code; compiling interpreters have been used to improve the performance of interpretation and thus help deliver most of the performance of a less portable representation (see "Compiling Interpreters", §10.3.4). Note that portable binaries are generally subject to the UNCOL problem (see "Interoperability", pg. 145).

Program Analysis Instruction-set simulation and program tracing tools that operate statically are confused by some forms of instruction space modification, including some forms of dynamic linking and self-modifying code. Also, while static discrimination of what is code and what is data is sometimes been amenable to static analysis [Fla 94, Hayden 94, Larus & Ball 94] the general problem is undecidable. For these reasons, a variety of tools have used dynamic analysis, based on observed usage; however, deferring analysis to runtime necessarily means deferring instrumentation to runtime, so analysis schemes that drive code generators must use RTCG [May 87, Nielsen 91, Cmelik & Keppel 93, Magnusson 93, Veenstra 93, Cmelik & Keppel 94, Veenstra & Fowler 94, Pittman 95, Rosenblum et al. 95].

Profile-Based Analysis A variety of systems have driven runtime optimizations based, in part, on runtime profiles that have been used to identify "hot spots" or common paths [Hansen 73, Magnusson 93, DeHon & Eslick 94, Hölzle 94, Brown 95a, Brown 95b]. In some systems, the profiling has been implicit; for example, a cache of dynamically-compiled code has been used as a mechanism for maintaining heavily-used code in a highly-optimized state [Naur 65, Brown 76, Deutsch & Schiffman 84, Ditzel & McLellan 87, Ditzel et al. 87a, Cmelik & Keppel 93, Cmelik & Keppel 94, Wakeling 95].

Runtime Hardware Reconfiguration Below the level of runtime generation of microcode ("Runtime Microcode Generation", pg. 136) is runtime reconfiguration of programmable gate arrays, referred to here as *Runtime Hardware Reconfiguration* or *RTHWR*. The goal of RTHWR is to configure the hardware to execute a particular inner loop, then reconfigure the hardware dynamically to execute a different inner loop. The general idea is like VSO, but the perspective of "code" (§9.1) is typically programs for a gate array. Examples of RTHWR have included processor reconfiguration [Koopman 89, Johnson & Fox 91, Sutherland 91, DeHon 94, Kilbane 95], coprocessor reconfiguration [Brunvand & Sproull 89, Gokhale et al. 91, Athanas & Silverman 93, Buell & Pocek 93] and interconnect reconfiguration [Snyder 82]. Although technology differences have made direct comparisons difficult, order-of-magnitude application speedups appear possible [Athanas & Silverman 93].

Many of the issues in RTHWR are like those in runtime microcode generation (below). For example, it is typically difficult to generate gate array "machine code" from a program written in a conventional high-level language; language subsets and supersets appear to be a workable alternative [Athanas & Silverman 93]; Similarly, virtualizing configurable hardware to support multitasking may require reconfiguration on each context switch. Reconfiguration can cost tens of thousands of processor cycles [Johnson 93]; configuration caching tagged by context number has been proposed as a potential solution [DeHon 94]. The effectiveness of RTHWR also depends on the granularity of reconfiguration of the underlying hardware and on the "gate-level instruction set" that is provided by the reconfigurable hardware [Johnson 93].

Runtime Microcode Generation A variety of systems have supported user-programmable microcode. Most, however, have optimized around application code but not runtime data values [Wilkes 69, Tucker & Flynn 71, Reigel et al. 72, Robbi 72, Wilner 72, Nan 74, Flynn & Neuhauser 75, Ebeling 76, Hassitt & Lyon 76, Roskos & Winner 81, Deutsch 83, Koopman 87, Pountain 88, Koopman 89, Baker 91, Johnson & Fox 91, Kehl 93, Drummond 95, Firestone 95, Greenblatt 95, Sand 95a] (but see "Network Router", pg. 134). Some machines have used multiple levels of microcode, with, for example, horizontal microcode executing vertical microcode that in turn executed the "user-visible" instruction set architecture [Reigel et al. 72, Nan 74]. In addition, the microcode equivalent of the "execute" instruction ("Execute", pg. 118) was used for indexing and operand patching [Firestone 95]. The appeal of user-programmable microcode has been that it offered access to a fast VLIW microarchitecture and thus potentially allowed fast inner loops to be written in microcode, while less-important code was interpreted from a compact form [Pittman 87].

A variety of systems have also used system-writable microcode in order to support boot-time patching of buggy microcode. In these systems, most of the microcode may reside in read-only memory, but the first microinstruction of each sequence is in a writable store, in order to allow arbitrary fixes.

Writable microcode has suffered from a variety of problems and disadvantages compared to designs such as read-only microcode and hard-wired logic [Sand 95a]: servicing page faults arising from user-written microcode was complex unless paging was below the microcode layer [Drummond 95]; it was sometimes difficult to ensure that protection and sharing were maintained; virtualizing the limited microcode memory typically complicated scheduling and increased the cost of context switches [Wilkes 72]; cheaper faster memory and caches made it more practical to build faster but less-dense macro instruction sets, which in turn eliminated microcode's advantage from using a fast local memory; read-only microcode memory was often cheaper and faster than writable microcode memory; microcode was rarely portable between processor implementations and was difficult to generate automatically; using more microcode increased the likelihood of errors in that code, and bugs were easier to remove from conventional code.

An alternative is microcode generated automatically from the macro-instructions, then saved for reuse [Dahlby et al. 78, Ditzel & McLellan 87, Ditzel et al. 87a, Colwell et al. 88, Koldinger 88, Berg et al. 95]. Caching is driven by program behavior, which is in turn dependent on program values; the behavior may thus be seen as optimization around program values and not just program code.[11] In contrast, simple caching of microcode [Robbi 72, Sand 95b] is not RTCG, because only the placement, and not the representation, is being changed by data-dependent behavior. Caching eliminated some of the aforementioned problems with writable microcode, but it also put cache lookup in the microinstruction fetch critical path.

The performance gain from user-writable microcode is difficult to estimate, even in comparison to an interpreted instruction set on the same machine. It appears, however, that order-of-magnitude speedups were possible [Ebeling 76].

Static Code Optimization using Observed Behavior At least two code optimization strategies have used generate-and-test. The first technique was based on the observation that two instruction sequences were interchangeable if they produced the same result; there was no need, for example, to use branches to implement an if-then-else if a faster sequence would produce the same result. The *superoptimizer* strategy thus generated arbitrary code sequences and ran them using pseudo–random inputs, saving out those that appeared to compute the same result as the original but which were, for example, smaller [Massalin 87, Granlund & Kenner 92].[12] The second optimization strategy used the observation that modeling the timing behavior of modern processors is at best difficult [Bedichek 95b]; the optimizer thus generated a variety of schedules for the same sequence and tried each one in order to determine which was fastest [Baker 92a, Kerns 92b].

String and Regular Expression Searching Matchers have used finite automata (FA) search engines consisting of code compiled to the particular search. Instead of using a data structure to represent the FA, the FA traversal engine was partially-evaluated with respect to each FA node [Thompson 68, Beeler et al. 72, Aus 90].[13] For example, Figure 10.7, shows regular expression pattern matching using an NFA implemented as an expression-specific code sequence [Thompson 68].

[11] A variety of processors, such as the Hal implementation of the UltraSPARC [Gwennap 95c] and the Intel P6 [Colwell 95, Gwennap 95a] have operated by translating user instructions to microcode using hardware and then issuing the microinstructions conventionally; since the translated form was not reused it is debatable whether this was RTCG; more likely it was simply pipelined instruction decoding. For systems that cache the translated form, the line between "generating microcode" and "tagging instructions in the instruction cache" is blurry. For example, the AMD K5 tagged each cached byte with four bits for detecting instruction boundaries, branch prediction [Halfhill 94a]; the MIPS R10000 tagged each 32-bit instruction with five bits that, for example, preassign the instructions to functional units [Halfhill 94b]; the PowerPC 620 decoded each 32-bit instruction into a 39-bit form in the instruction cache; predecoding eliminated one processor pipeline stage [Thompson & Ryan 94]. Each of these designs converted the representation dynamically and then cached that representation. However, these systems still maintained a one-to-one correspondence between translated and untranslated instructions.

[12] The term "superoptimizer" has more recently come to mean any exhaustive optimization strategy, for example compiling a benchmark with every combination of compiler optimization flags and then selecting the flag combination that produces the fastest executable.

[13] Gosper devised and described a linear-time string searching algorithm that was dynamically compiled to machine code for improved efficiency [Beeler et al. 72]. Half a decade later the algorithm became more widely known as the KMP algorithm.

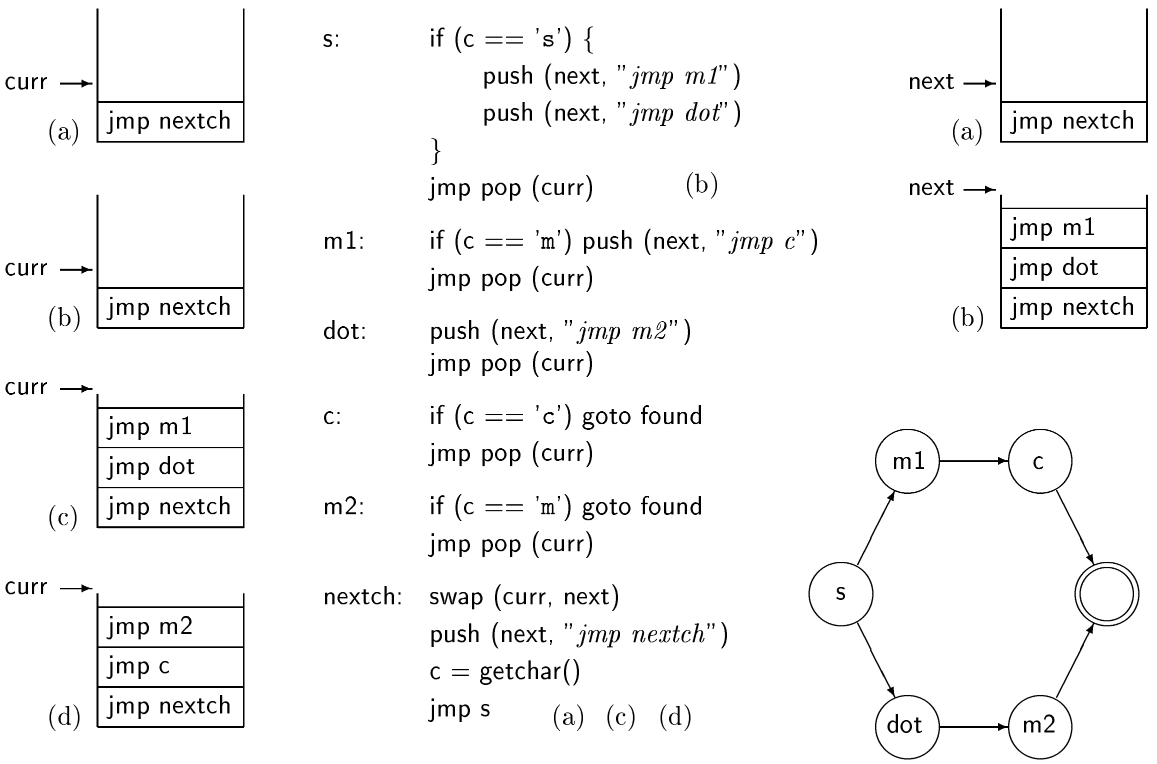


Figure 10.7: A threaded-code NFA interpreter for matching regular expressions.

A threaded-code interpreter that performs regular expression matching using a breadth-first traversal of the NFA in the lower right. The NFA is created from the regular expression "`s(mc|.m)`", where "`.`" denotes a match on any single character. The code blocks s, m1, etc. represent states in the NFA shown in the lower right. The stacks curr and next are the "threads" that, at any given point in pattern matching, represent the search frontiers for the current and next input characters. The pattern matcher starts execution by initializing curr and next and then jumping to nextch. The following describes execution with the input string "`smc`". At (a) after reading the character "s", the current and next frontiers are empty. The state "s" examines the input character and pushes states "m1" and "dot" on the next frontier, implemented by pushing the instructions jmp m1 and jmp dot. When control reaches (b), the current frontier is empty so jmp pop (curr) jumps to the jmp nextch, and it jumps to nextch to swap frontiers and read another character. At (c), after reading "`m`", the current frontier contains the two states previously pushed by s. This time, s doesn't push any states, so jmp pop (curr) jumps to jmp m1, and then to m1. State m1 pushes jmp c and then jumps to jmp dot, which jumps to dot, pushing jmp m2, and then jumping to jmp nextch and then to nextch. At (d), after swapping frontiers and reading "c", the current frontier has two states; state s pushes no new states and state m2 doesn't match; however, state c matches, transferring control to the final state found (not shown).

Type-Based Dynamic Optimization In dynamically-typed languages, the type of each operand is used in deciding which version of an operation to invoke. When the common operations are simple, the cost of such type dispatch can dominate total execution times. Static optimization is difficult because, in general, each operand to each operation may be of arbitrary type; even modest static special casing may thus lead to rapid code size explosion. A variety of programming language implementations have improved performance by dynamically generating code that was optimized around the observed types [Dyke & Bree 77, Guibas & Wyatt 78, Johnston 79, Saal & Weiss 79, Deutsch & Schiffman 84, Ungar et al. 84, Christopher & Wallace 86, Samples et al. 86, Ungar & Patterson 87, Chambers et al. 89, Chambers & Ungar 89, Chambers & Ungar 91, Hölzle et al. 91, Chambers 92, Stemple et al. 92, Ungar et al. 92, Blair 93, Stemple et al. 93, Cutts et al. 94, Hölzle & Ungar 94b, Hölzle 94]. These optimizations are profitable, because processors are rarely optimized around around type dispatch, thus making type-based optimizations relatively advantageous, and because types are typically stable (§B.2), so runtime recompilation rates are typically low.

In practice, some static type specializers have successfully used profiles of previous runs and static type analysis [Calder & Grunwald 94, Dean et al. 95a, Dean et al. 95b, Grove et al. 95]. However, active use of first-class types complicates static analysis [Stemple et al. 92, Stemple et al. 93, Dillenger 95]. Static optimization is also complicated by separate compilation of shared libraries, because structure offsets depend on use [Pennello 93]. Here, RTCG has still shown benefits, because concrete type representations have been available when code is generated [Lewis 93, Franz 94, Gosling 95].

Value-Based Dynamic Optimization In both statically- and dynamically-typed languages, stable values have been used to drive code optimization (Part I).

A wide variety of systems have used fine-grained instruction modification, as described in Sections 10.3.1 and 10.3.2.

One of the earliest appearances of medium-grained, value-based optimization was in Spacewar, the first real-time interactive computer game. Originally, each ship outline was drawn by interpreting a list of offsets relative to the ship's current coordinates. In order to provide time for gravity calculations, outline drawing was rewritten to use dynamically-unrolled ship drawing routines, each specialized to the offset list of the particular ship. The code generated for two ships occupied about 20% of the total memory and the compilation time was about triple the interpretation time. However, updates ran at peak hardware update speeds, more than five times as fast as the interpreter version [Russell 95a].

A variety of other applications have used medium-grained, value-based optimization, including bitblt [Pike et al. 85, Locanthi 87, Miranda 87, Foley et al. 90, May 92, Kaepplein 95]; cache simulation [Przybylski 88, Przybylski et al. 88]; matrix multiply [Leone & Lee 93, Engler & Proebsting 94, Leone & Lee 94, Auslander et al. 95, Engler et al. 95, Leone & Lee 95, Auslander et al. 96, Engler et al. 96, Leone & Lee 96b], scaling [Auslander et al. 95, Engler et al. 95, Auslander et al. 96, Engler et al. 96], and solving a system of linear equations [Leone & Lee 96b]; operating system communication, scheduling and I/O [Pu et al. 87, Massalin & Pu 88, Pu et al. 88, Massalin & Pu

89, Massalin & Pu 91, Massalin 92a, Pu & Walpole 93, Pu et al. 94, Bershad et al. 95, Leone & Lee 95, Leone & Lee 96b]; sorting [Levine 89], symbolic operations such as association lists [Leone & Lee 96b], set membership [Leone & Lee 96b] and genetic algorithms [Leone & Lee 96b]; and so on. A related class of value-based optimization are compiling interpreters for languages with only a single type; examples include embedded interpreters for small languages ("Compiling Interpreters", pg. 129), instruction-set simulation and tracing ("Program Analysis", pg. 135), logic simulation ("Logic Simulation", pg. 134), regular expression [Thompson 68, Aus 90, Leone & Lee 95, Leone & Lee 96b] and string [Beeler et al. 72] searching, and so on.

For bitblt, speedups were on the order of 2X over carefully-optimized C and assembly [Locanthi 87] and as much as 10X improvement over "straightforward" code [Miranda 87]. Matrix multiply showed speedups of 50% to 4X for relatively large (200 by 200 or larger) matrices [Leone & Lee 93, Engler & Proebsting 94, Leone & Lee 94, Auslander et al. 95, Engler et al. 95, Leone & Lee 95, Auslander et al. 96, Engler et al. 96, Leone & Lee 96b] and large matrix scaling showed speedups of 35% to 10X for large (1024 by 1024) matrices [Engler & Proebsting 94, Engler et al. 95, Engler et al. 96]; for the SPARC-based results, substantial improvement came from replacing calls to a multiply library with strength-reduced shifts and adds, and some improvements were observed on straight-line code for the AXP [Auslander et al. 95, Auslander et al. 96]. For operating systems, some speedups were over 50X [Massalin & Pu 89], but direct comparisons are difficult because both the interfaces and implementations varied. Performance comparison numbers are not available for the remaining.

Historically, most value-based optimizations have been implemented manually. Some systems automate the derivation of application-specific code generators but use manual (programmer) identification of optimization sites [Appel 88, Leone & Lee 93, Leone & Lee 94, Yarvin & Sah 94, Auslander et al. 95, Engler et al. 95, Leone & Lee 95, Auslander et al. 96, Chambers et al. 96, Consel & Noel 96, Engler et al. 96, Leone & Lee 96b, Poletto et al. 96].

A few systems have performed value-based optimization automatically, but these systems rarely try to improve performance over statically-compiled statically-typed code. Graph reduction systems, typically used in lazy functional languages, have performed value-based optimization by executing a graph and rewriting the graph as a part of execution [Turner 79, Hughes 82, Hudak & Kranz 84]. Most graph reduction systems are based on interpreted data structures and performance has usually been worse than using statically-generated machine code that does not self-optimize [Appel 88]. There are at least two alternatives that use both machine code and value-based optimization. One proposal was to dynamically compile reduced graphs, though the proposal suggested that doing so automatically in a broad setting was difficult [Appel 88]. Other systems have implemented the graph as machine code, so that performing reduction would automatically produce machine code [Kelly 89, Koopman 89, Koopman & Lee 89, Koopman et al. 92].

Some language implementation systems have cached dynamically-generated value-optimized code for reuse without recompilation overhead [Ditzel & McLellan 87, Ditzel et al. 87a, Cmelik & Keppel 93, Cmelik & Keppel 94]. As in a conventional cache, the contents of the code cache have depended on what was used recently. Since usage patterns generally depend on program values, the compilation and caching process may be seen as value-specific optimization. In particular, code that is used heavily may be "promoted" by reoptimizing it; for example Adaptive FORTRAN fused blocks that were executed closely in time, even if they were not spatially adjacent [Hansen 73].

Virtual Memory via Code Generation One function of virtual memory is to allow code to be loaded at varying physical memory addresses during execution. Code relocation may be performed by patching the code and updating stored program addresses each time the code is moved [McGee 65]. Another form of virtual memory has been implemented by quickly generating code from a compact representation instead of reading it from disk, since regeneration is sometimes faster than reading from secondary storage [Brown 76, Deutsch & Schiffman 84, Pountain 94] (see also "Overlays", pg. 142). Note also that some functions of virtual memory have been implemented in systems which lazily patch static code (e.g., "Unimplemented Instructions", pg. 123) and then discard the modified pages instead of paging them out, by lazily re-patching the original code when it was needed again.

10.3.5 Large Code Blocks

Copy Protection, Reverse Engineering Protection Many microcomputer programs were protected against copying, for example, by rewriting disk drivers so that the programs could use a disk format that was incompatible with the native machine drive format. The details of the copy protection scheme were then hidden by encrypting parts of the program and decrypting them on demand. Encryption was also used to protect against reverse engineering. Some systems not only decrypted code, they also re-encrypted code after execution [Rossi 92].

Dynamic Linking Dynamic linking allows separately-created code modules to be be bound together at runtime; unlinking and relinking are also possible. Dynamic linking is typically large-grained, manipulating only procedures and collections of procedures; it is thus less flexible and less powerful than other forms of RTCG, but it is also typically cheap [Seeley 90] and allows sharing of the dynamically-linked code. Dynamic linking has been used to implement a variety of services including abstraction embedding ("Abstraction Embedding", pg. 128), extension programming ("Extensible Systems", §10.3.4), incremental compilation ("Compiling Interpreters", pg. 129), interposition, platform-specific implementation of operating system services, shared libraries (code sharing across disjoint applications), user customization, and so on. Some dynamic linkers have also used RTCG in their implementation of procedure calls ("Dynamic Linking", pg. 121, "Dynamic Linking", pg. 126). Dynamic linking issues include: statically verifying the existance of code that will be dynamically linked; versioning to ensure that the caller and callee are compatible; optimizing to minimize the runtime cost of dynamic linking and of dynamically-linked calls; producing code that may be relocated while still allowing sharing; determining whether code can safely be unlinked; and ensuring that privileged programs do not invoke user-supplied dynamically-linked libraries [Crowe 87, Gingell et al. 87, Sun 89, Segal & Frieder 89b, Ho 90, Ho & Olsson 90, AT&T 90a, Maeda 90, Sabatella 90, Seeley 90, AT&T 90b, Ho & Olsson 91, Frieder & Segal 91, Dig 92b, Garrett et al. 92, Segal 92, Hamilton & Kougiouris 93, Keppel & Russell 93, Ho et al. 95, SGI 94, Sirer et al. 96].

Overlays OS kernels and application programs have both used *overlay* schemes that divided memory into segments, so that not all segments needed to be in memory at once. The effect was analogous to hardware paging or segmentation but was implemented programatically using software [Spacek 72]. Some schemes were driven by program development tools and were thus invisible to the programmer [Naur 65]; some were also adaptive so that overlay-related I/O would decrease when there was more available memory [Naur 65, Rowe 86]. Most overlay schemes were used to map large programs into small memories. For example, each cross-segment call was patched (§10.3.1) when the segment was moved to secondary storage, so that subsequent calls were redirected to the overlay driver, which would reload the target segment and re-patch the calls to go directly to the callee. More sophisticated schemes have allowed software to "hint" the overlay driver in advance, in order to take advantage of overlapped computation and I/O [Spacek 72]. Some overlay schemes were used to map small virtual addresses to large physical memories The general design was like other overlay schemes, but instead of performing physical I/O, the overlaying performed remapping operations [Harrenstien 95, Harris 95]. An overlay system may be viewed as a sophisticated dynamic linker ("Dynamic Linking", pg. 141).

Self-Decompressing Binaries Self-decompressing programs have been built such that the program executable consists of a small program and a large amount of initialized data. On invocation, the small code fragment would decompress most of the data into executable code that was then executed [Joy 90, Kaepplein 95]. Typically, the compressed data was stored starting at the same location as code in an uncompressed binary. An "obvious" implementation would thus need to move the data before decompression started. However, decompression was implemented so that it ran "backwards" from high addresses to low, thus overwriting compressed data only after it was read by the decompressor and thus avoid the extra copying [Joy 90].

10.4 Other Instruction Space Modifications

Procedure Linkage Various machines codified procedure linkage by patching parameter values or addresses into the code segment immediately following the call instruction, or by patching the return address into a location near the start of the procedure. For example, the URAL, DEC PDP-1, PDP-6 and PDP-10, and CDC 6400 implemented `call foo` by placing the return address at `foo` and jumping to `foo+1` [Russell 95b, MIT ed, Littlefield 90, Simonyi 94, Greenblatt 95]. Linkage return was accomplished with `ret foo`, which read the return address from `foo` and branched to the specified address or with `jmp foo`, which would transfer to `foo`, which contained a jump instruction whose target had been updated by the `call foo`. Parameters to procedures were placed immediately following the `call` instruction [Organick 72, Simonyi 94] and accessed using the address in `foo`. In the URAL, procedure return was implemented with an instruction, `#30`, which added a number to the result of the following instruction; thus returning past two words of instructions was accomplished with #30 2; `ret foo` [Simonyi 94].

10.5 Issues and Techniques

This section explains several techniques that appear in the context of runtime code generation. Some, such as executable data structures, deserve special attention because they appear periodically and because they "straddle the fence" of the code/data duality. Others, such as instruction cache coherency and register reallocation are not themselves examples of RTCG, but are used in implementing RTCG.

10.5.1 Adaptive Reoptimization

A variety of systems, typically compiling interpreters ("Compiling Interpreters", pg. 129), have used *adaptive reoptimization*, in which the optimization level is adjusted to reflect current usage. In many ways, adaptive reoptimization has the flavor of a *competitive* algorithm [Karlin et al. 86]; in practice, however, adaptive code optimization has been based on heuristics. Several adaptive reoptimization policies have been used:

- Code is initially translated at low optimization or is interpreted. The low-optimization code is also profiled, so that heavily-used regions may be recompiled at successively higher levels of optimization [Hansen 73, Blair 93, ARD 94, Hölzle 94, Ltd. 96]. Background reoptimization has also been suggested to reduce observed delays from reoptimization and thus allow expensive optimizations and optimizing trials that may be discarded [DeHon & Eslick 94, Brown 95a, Brown 95b]; the expense of background processing is that the optimized code typically is not available until a later invocation of the program.

- Code is initially "over-specialized" and adaptively reoptimized as the common usage changes. Strategies have included reoptimizing with new assumptions [Dyke & Bree 77, Guibas & Wyatt 78, Johnston 79, Cmelik & Keppel 93, Cutts et al. 94], reducing optimization [Dyke & Bree 77, Johnston 79, Saal & Weiss 79, Cmelik & Keppel 93], and dynamically selecting among optimized versions [May 87, Hölzle et al. 91, Cmelik & Keppel 93, Hölzle & Ungar 94b, Hölzle 94, Cutts et al. 94, Samuelsson 94, Bedichek 95a].

- Optimizers may make time/space tradeoffs, for example, generating denser but slower code if recompilation rates rise due to space constraints [Turner 79].

Adaptive code reoptimization attempts to balance compilation and optimization costs against the added benefits in heavily-used parts of the code. In principle, it is possible to gradually increase optimizations and thus "creep up" on highly-optimized code with relatively little chance of generating poorly-optimized code. In practice, however, gradually increasing the optimization level also means performing compilation many times, thus increasing the cost that must be amortized. Therefore, it appears that it is usually best to make large jumps in the optimization level [Hansen 73].

One general problem with reoptimization are that profiled code is typically slower, and that the optimized code only becomes available after some feedback delay, that is, after some of the execution had proceeded using the slower code.

10.5.2 Breakeven Point

Previous work has explained the concept of a breakeven point [Hansen 73, Deutsch & Schiffman 84, Heering et al. 91]. Those systems, however, used RTCG below the breakeven point. The average performance loss was minimized by for example, using a competitive-style algorithm [Karlin et al. 86] to switch between low-cost and high-performance implementations; but individual runs would still sometimes run with worse performance than if purely static code had been used.

10.5.3 Cache Coherency Primitives

On many platforms, instruction cache coherency is performed by calling some kind of "cache coherency primitive". The primitive should be accessed using some kind of dynamic binding; there are several reasons. First, coherency implementations vary widely across otherwise similar machines [Sun 91, SGI 94, Mashey 95]. Dynamic binding allows each platform to supply the most efficient implementation, without sacrificing generality. Second, there may be many contenders for access to any given part of the instruction space; for example, compiling interpreters ("Compiling Interpreters", pg. 129), debuggers (§10.3.2 and §10.6.2), dynamic linkers, ("Dynamic Linking", pg. 121, §10.3.3 and pg. 141), some mobile code verification systems (§10.5.15), optimizers ("Type-Based Dynamic Optimization", pg. 139, "Value-Based Dynamic Optimization", pg. 139, etc.) and so on. Using a dynamically-bound coherency mechanism allows an appropriate arbiter to be interposed efficiently.

Note that using a well-structured interface for instruction-space modification does not solve the more general problem of naming code that exists only transiently. For example, additional mechanisms are required to support a debugger that needs to be informed when routine `foo` has been generated, so that it can set a breakpoint there.

The costs of coherency should be exposed, so that each application can compute the costs and benefits of RTCG. Ideally, coherency should also be performed via some portable interface (Chapter 11).

10.5.4 Cache Flushing Via Code Placement

A wide variety of systems provide hardware or operating system primitives for cache coherency [Mot 90]. Some systems have bypassed the system primitives and instead flushed incoherent code by executing another code fragment that has been placed to cause a cache conflict and thus replace the incoherent code [Deutsch 90, Massalin 90, Massalin & Pu 92] (Chapter 11).

Cache flushing via code placement manipulates cache state implicitly, so it works only when there is a predictable mapping between the virtual address of an instruction and its location in the cache — for example, the cache is virtually indexed — and when the cache uses predictable replacement policy, so that a finite sequence of instructions can guarantee that a cache location has been flushed. Figure 10.8 shows an example of cache flushing using a *table of code* approach, where a modified instruction at address `addr` is made coherent in a direct-mapped cache by executing another instruction that maps to the same cache line [Deutsch 90, Massalin 90, Massalin & Pu 92] (see also §11.7.2 for tradeoffs).

The instructions in the "table of code" exist solely to implement cache flushing; Synthesis implemented even faster instruction cache flushing, by forcing instruction placement so that when one instruction in a sequence was modified, another instruction — often, one in the sequence that modified it — would map to the same cache location and thus implement flushing (§10.5.20).

A Toshiba implementation of a MIPS–architecture processor *required* cache flushing by code placement, and also required careful code placement in order to achieve the effect of locking code into the instruction cache [Turley 94].

10.5.5 Code Formats for Dynamic Code Incorporation

A wide variety of applications have dynamically incorporated and executed some kind of code. Dynamic code incorporation has been used to implement abstraction embedding ("Abstraction Embedding", pg. 128); *extension languages*, which allow users to extend existing applications by programming them ("Extensible Systems", pg. 132); systems which provided programmers a mechanism for automatically generating and incorporating code [Koopman 89, Krall & Berger 94, Krall & Berger 95b, Krall & Berger 95a], (§10.7); interactive programming systems ("Interactive Programming Systems", pg. 133), and dynamic version update systems ("Dynamic Version Updating", pg. 131).

In systems where instruction-space modification is strictly prohibited, there are two implementation choices. One is to represent the dynamically incorporated code as virtual machine code that is loaded as data and then executed using an interpreter [Johnston 79]. The other choice is to use machine code, but the "dynamically incorporated" code is not incorporated at all; instead, it is accessed via interprocess communication [Notkin & Griswold 87, Notkin & Griswold 88]. In contrast, systems that allow runtime incorporation of native machine code have several advantages:

Flexibility The overheads of interpreting or interprocess communication may slow the entire application; in turn, the flexibility of programs is often limited, because more-flexible designs are too slow [Lampson 84]. In contrast, dynamically-introduced native machine code has performance closer to static machine code, yet has the flexibility of virtual machine code and thus allows for more flexible designs.[14]

Interoperability Using virtual machine code leads to a variety of bizarre interoperability problems. First, applications that include code must contain an interpreter, so it is necessary to decide in advance which applications can include code. Second, it is necessary to decide in advance *which*

[14] Note that dynamic-code implementations may be faster than static code with equivalent functionality, but as flexibility is removed, the performance advantage of dynamic code goes down. For a very inflexible design, static code will often have better performance because the dynamic code implementation sacrifice some optimizations in order to allow for e.g., out-of-line calls to dynamically-linked code.

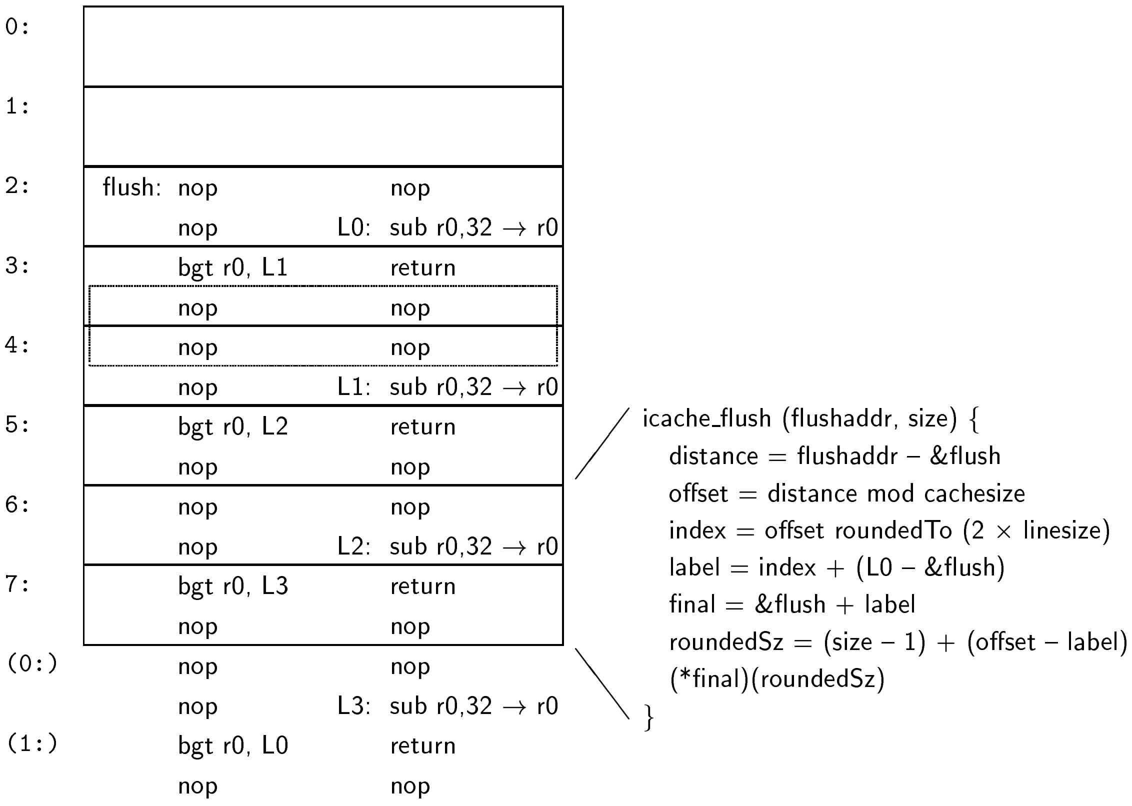


Figure 10.8: A “table of code” for cache flushing.

The “table of code” is superimposed over the cache that it flushes and also over code that invokes it. The cache is 8 lines of 16B each; instructions are shown in pairs for compactness. The table of code is as large as the cache, and each entry flushes two cache lines, so the table must be aligned on an even cache line boundary. The icache_flush routine computes a branch entry by computing a table entry that uses the same cache line as flushaddr, and then jumps to it. Each table entry flushes two lines and jumps to the next table entry, until at least size bytes have been flushed. The rounded size, roundedSz adjusts for the difference between the cache location of flushaddr and the label used to flush it. For example, the dashed box is 32 bytes long; the starting address maps to L0, but subtracting just 32 at L0 would cause a return in block 3, failing to flush the second half of the box. A variety of optimizations are possible: alignment to a cache-size boundary would make invocation cheaper; limiting roundedSz to cachesize would bound the cache flushing cost; the mod and roundedTo operations can be performed with a single bit masking operation; if icache_flush maps to lines 6 and 7, then those table entries should be skipped, e.g., by replacing the preceding sub 32; bgt L2 with sub 64; bgt L3.

virtual machine code interpreters to include. Including too few leads to problems with incompatible virtual machine code formats, because converting or interpreting between formats is often cumbersome or slow; interpreting between formats necessarily uses an interpreter that is itself implemented in some virtual machine code, leading to a double interpretation overhead [Mitchell 70]. Alternatively, the lack of a compatible format means that many extensible applications use a "hardwired" language to achieve acceptable efficiency, but at the expense that users are required to learn different extension languages for different applications [Notkin & Griswold 87, Notkin & Griswold 88]. However, including too many interpreters bloats the application and still doesn't solve the general problem that new virtual machine code formats will appear after an application is built. Third, each virtual machine code format provides better performance for some source languages but worse performance for others, referred to as the *UNCOL* problem [Strong et al. 58, Conway 58, Bratman 61, Halpern 65]. Poor performance, however, encourages a proliferation of virtual machine code code formats. Finally, to allow for symmetrical interoperability, *every* applications must be available in a virtual machine code format. For example, a word processor document may include a spreadsheet, and a spreadsheet may include a word processor document, so both the spreadsheet and the word processor need to be available as virtual machine code.

Dynamic incorporation of machine code eases these interoperability problems. The machine code format defines a common language for communication, and also allows the use of any virtual machine code for which there is either a machine-code translator or a machine-code interpreter [Notkin & Griswold 87, Notkin & Griswold 88]. Applications do not need to be built specifically to include code or to be included in other applications, so all applications can use machine code implementations, and new applications may be used as they become available.[15] In short, using dynamic machine code greatly simplifies a variety of code representation issues by defining a common format and by decoupling source-level interface issues from the implementation-level compatibility issues.

Sharing Interprocess sharing is generally difficult [Chase 95b]; code that is actually incorporated into the running program has more natural access to shared data [Wahbe et al. 93].

10.5.6 Code Generator Performance

Unfortunately, code generator performance is not reported widely, probably because there are few standalone code generators and even fewer that are targeted towards generation of machine code in memory [Fyfe et al. 91, Engler & Proebsting 94, Engler 96]. Instead, most code generators are parts of compilers and, except for specific long-running optimizations, the cost of code generation is typically dwarfed by scanning, parsing, and so on [Fraser & Wendt 88].

A few specific numbers, however are available. Generating locally-optimal VAX assembly code from directed acyclic graphs required a few hundred instructions per generated instruction [Fraser & Henry 91]. The cost for direct machine code generation was similar; for example, 350 instructions per emitted instruction for matrix multiply for a MIPS R3000 [Engler & Proebsting 94]. Interpreting

[15] In order to maintain portability, the machine code itself should be generated dynamically from a portable representation; see "Compiling Interpreters", pg. 129.

three-address code on a PDP-10 took on the order of 15 instructions per interpreted 3-address code; compiling using "dumb" code generation used on the order of 300 instructions per 3-address code, and adding common subexpression elimination, branch optimizations and so on raised the cost to about 650 instructions [Hansen 73].

Template compilers have typically achieved much better performance, for example four to ten instructions per generated instruction for RISC machines [Leone & Lee 94, Leone & Lee 95, Engler 96, Leone & Lee 96b, Leone & Lee 96a]. In Shade, analyzing SPARC V9 target machine code and emitting SPARC V8 host code on an instruction-by-instruction basis (but using local register allocation) typically consumed 600-1,000 host instructions per target instruction and 150-200 instructions per host instruction emitted [Cmelik & Keppel 93, Cmelik & Keppel 94, Cmelik & Keppel 95]. The BLIT bitblt compiler generated code using a template compiler at a cost of approximately 20 instructions per generated instruction [Pike et al. 85]. Massalin's machine code peephole optimizer executed tens of instructions per three-instruction sequence that was considered by the optimizer [Massalin 90]. The Deutsch-Schiffman Smalltalk-80 virtual machine [Deutsch & Schiffman 84] executed approximately 50 instructions per generated instruction [Deutsch 90, Hölzle 94].

The VCODE system was designed to provide a compiler writer's virtual machine that would generate code using templates with one or a few instructions. It typically uses dynamic register allocation, using an implementation similar to that in Figure 3.7. The set of templates corresponded to an abstract machine code rather than the aggregate operations used in application-specific template compilers. It executed tens of instructions per generated instruction [Engler 96] and has been used to implement a code generator for tick-C [Poletto et al. 96] (see "Language-Level View", pg. 170). The New Jersey Machine Code Toolkit (NJMCT) was designed to build a variety of tools, including a code generator generators. A portable code generator built using NJMCT executed thousands of instructions per generated instruction [Fernandez 95].

A few other observations are also in order. First, code generator performance has been improved by saving context information during one compilation run, and then using that to do less work on recompilation [Fyfe et al. 91]. Second, code generators have been "trained" to improve the performance on likely inputs, while maintaining the flexibility of general-purpose code generation [Henry 87, Fraser & Wendt 88]. Finally, 3-address code has been used as an internal representation in order to allow static register preallocation to improve code generator performance. Register allocation was then repeated at runtime if the host machine had fewer registers than the target for which the 3-address code was generated [Col 95].

10.5.7 Compiler Invocation Strategies

There are a variety of strategies for determining when to invoke the runtime compiler and deciding how large a unit to use for compilation.

A variety of RTCG systems have used demand-driven compilation. LC^2, SELF and Synthesis used an implementation similar to shrink-wrapping [Chow 88], in which unexpected paths were initially unimplemented but were implemented on demand if an unexpected path was taken. LC^2 path expectation was driven by actual use. Initially, an if-then-else was implemented by compiling the not-taken path as a call to the compiler. When the path was subsequently taken, the compiler split the existing code and inserted code to implement the newly-taken path [Mitchell 70]. In Self, the expected and unexpected paths were driven by type prediction and by observed use, with "failure" paths for type tests implemented initially as calls to the runtime compiler [Chambers & Ungar 91, Hölzle 94]. In Synthesis, the notion of "expected" was chosen by the programmer [Massalin 90, Massalin 95]. A related scheme is to interpret or to generate low-optimization code that is self-profiling and which invokes the compiler when the profiler trip counts exceed some threshold (§10.5.1).

Synthesis inlined code and data values that changed. Changes were propagated using a variant of the Model View Controller [Goldberg & Robson 83], in which each caller registered a callback with the callee. When the given callee code and data values changed, the callee invoked the callbacks in order to notify the callers. Typically, callbacks did not directly recompile the caller but instead marked the caller in a way that future invocations would cause a lazy update. For example, queue access was generally inlined in the callers, and each inlining operation also registered the caller with the queue object. Different queue implementations were used depending on the number of concurrent readers and writers [Massalin & Pu 91, Pu & Massalin 91]; when a reader or writer attached or detached from a queue, all queue accessors were notified so that the inlined code would be recompiled. Recompilation was actually achieved by patching the caller so that when it was executed, it would jump to the dynamic compiler. Deferring recompilation this way (a) simplified the task of ensuring that code was never recompiled while it was being executed and (b) allowed many changes to result in only a single recompilation [Massalin 90, Massalin 95]. Shade [Cmelik & Keppel 94] used a variation in which invalid translations were modified so that when they were invoked they would lazily unchain their predecessors.

10.5.8 Executable Data Structures

Several systems have integrated code and data using a technique called *executable data structures* [Pu et al. 88, Koopman 89, Koopman & Lee 89, Massalin & Pu 89]. The idea is to partially evaluate an operator on a data structure with respect to each node of the data structure, and then put the resulting specialized code into each node of the data structure. When "data" values change the code is updated to reflect the new values.

Consider an object-oriented system where each object has instance variables accessed with functions, for example "i = x.get()" to read the value of x, and "x.put(i+1)" to increment the value of x. As shown in Figure 10.10(a), a typical implementation associates the code of general-purpose accessor functions with a class object and links each "base" object to the appropriate class object. The accessors are invoked using double indirection off of the base object. As shown in Figure 10.10(b), the function pointers may be cached with the object instead of the class. Then, invoking the accessors needs only a single indirection.

When function pointers are stored on a per-object basis, each generic function may be replaced

```
i = x.get();
x.put (i+1);
```

Figure 10.9: Calls to fetch and set the value of an object.

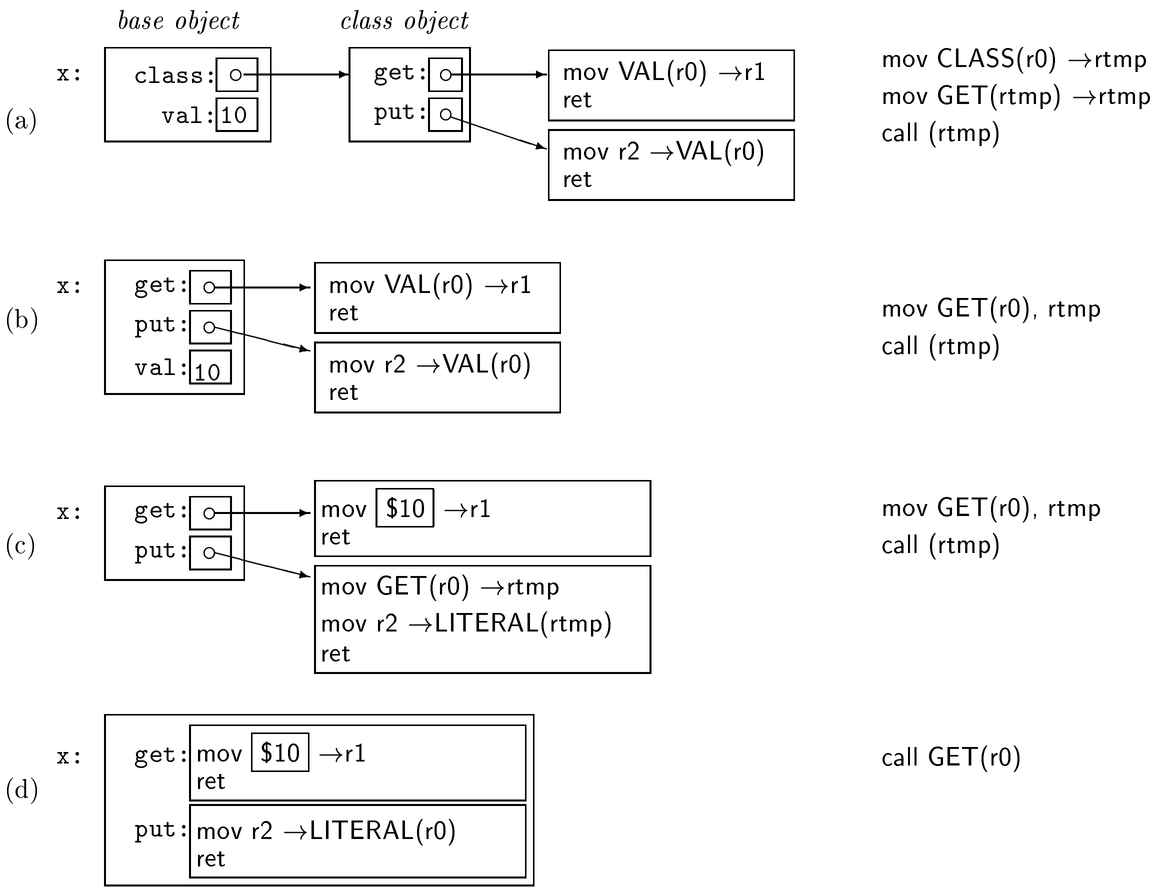


Figure 10.10: Various implementations of methods and method invocation.

Implementations of a simple object and methods to get and put the value of the object; also, code to invoke the get method for each implementation. All examples assume that register r0 contains a pointer to the object, that get sets r1 to hold the value of x, and that r2 holds the new value of x used by put. A conventional implementation is shown in (a), along with code to invoke the get method. In (b), the class object is inlined in the data object, which eliminates one level of indirection from the call to get. In (c), the get routine is specialized with respect to the current data value. The put routine is also rewritten so that when it is invoked it will compile a new version of get. In (d) the specialized code has been inlined directly in the object, removing a level of indirection from the call to get and also from put. Where the get method is inlined in the caller and the put modifies code, as in versions (c) and (d), then put must modify all sites where get is inlined.

with a function that is specialized to the particular object, as shown in Figure 10.10(c). Here, `get` is a routine that returns a constant value, while `put` is a routine that compiles a new version of `get`.[16] Invocation is still via indirection, but the per-object function can be optimized more than the generic version. In addition, the explicit data storage for `val` is redundant and can be eliminated.

Function invocation still requires indirection. That can be eliminated by simply storing the function in the object itself, as shown in Figure 10.10(d). Note that inlining also eliminates the indirection required by the previous version of `put`. Object management is simplified, because there is no need to allocate or free separate code storage, but it must be possible to decide in advance the maximum size of the functions `get` and `put`.

It might at first seem that there is a problem: the specialized version is not reentrant. However, it is not meant to be. Instead, the code represents a particular data value, and that value changes exactly when the data value changes. The code is therefore always "right".

On some systems, the executable data structures implementation can be very fast. In Figure 10.10(a), one invocation of `get` uses 5 instruction references and 3 data references, whereas the executable data structure version in Figure 10.10(d) uses 3 instruction references, no data references, and one fewer register. However, several machine dependencies are encapsulated in this example. The implementation of `put` (a compiler) requires that literal storage is suitably aligned and be large enough to hold all legal values of `val`. Overwriting the `get` code generally requires cache and memory management (§11). It may be necessary to do extra work to ensure that code update is not atomic. Thus, on some machines and for some uses of `get` and `put`, the above implementations will not be profitable and an "interpreter" version, Figure 10.10(a) or 10.10(b), should be used.

A large variety of systems use executable data structures, though a variety of terms are used. Examples include a stack to represent a frontier in a breadth-first search ("String and Regular Expression Searching", pg. 137), linked lists for memory allocation ("Threading", pg. 123), combinator graph reduction [Koopman & Lee 89] and operating system threads [Pu et al. 88, Massalin & Pu 89].

10.5.9 Incrementality

Each compiled code unit depends on some source representation. In a non-incremental system, code is generated for the entire source unit. However, changing any part of the source means that the entire compiled unit becomes invalid and must be recompiled. Alternatively, the source may be broken into pieces, with code generated separately for each piece. Changing one piece then requires code generation only for that piece. However, using smaller code generation units generally hurts code quality. In an *incremental* system, code is generated for the whole source unit, but it is done in such a way that source changes can often update just a part the generated code instead of completely recompiling it [Mitchell 70, Dyke & Bree 77, Johnston 79, Pollock & Soffa 84, Santi 89, Fyfe et al.

[16] Note that "compilation" is accomplished by patching the existing version; conceptually, the `get` routine is being deallocated, new code is being generated and the new code is installed. However, the given version of `get` is, by definition, used only in this one place, so the compilation process is optimized as in Section 9.3, by observing that the old storage is the same size as the new storage and that the bit values in the old storage are the same as the bit values in the new storage, except for one small part.

91, Karinthi & Weiser 87, Heering et al. 91, Krall & Berger 94, Krall & Berger 95b, Krall & Berger 95a]. The complexity in incremental systems is that code is still divided into pieces, but each code fragment depends on the context of the whole unit. When one source fragment changes, it is necessary to propagate the effects to all pieces that depend on the change, and it is also potentially necessary to update the dependency relationships.

10.5.10 Instruction Cache Coherency Implementation

MIPS On MIPS R2000 processors, instruction cache flushing required executing a privileged sequence that swapped the roles of the instruction and data caches and then wrote to the instruction cache — only certain store instructions would do — all with interrupts turned off [Killian 90].

Motorola 88000 Family The standard cache configuration for the 88000 processor family used separate cache chips that supported a wide variety of coherency configurations: flushing was provided at the level of cache lines, pages, or whole caches; individual pages could be marked as noncacheable; and individual pages could be marked so that the instruction cache would snoop writes, though instruction cache snooping caused contention with processor fetches from the instruction cache.

SPARC The standard instruction cache coherency interface used the `flush` instruction, which ensured a doubleword of instruction cache was coherent with memory. On many implementations, the `flush` instruction acted as a no-op or as a relatively quick on-chip operation. Some implementations, however, would trap whenever `flush` was executed [Sun 91]. Thus, coherency costs varied by several orders of magnitude across implementations.

10.5.11 Instruction Space Allocation, Reuse and Freeing

Instruction space may be allocated and freed using any conventional data space allocation models, including static preallocation (§B.6), stack allocation [Breuel 88, Koopman 89, Crispin 90, Chase & Hood 91, Dig], lazy/on-demand allocation with explicit freeing [Glew 87, Abbott & Peterson 92] or with garbage collection [Foderaro 90], reuse of already-allocated areas, copy-on-write, and caching with a variety of reclamation strategies when cache space is exhausted [Brown 76, Deutsch & Schiffman 84, Cmelik & Keppel 93, Cmelik & Keppel 94]. In addition, many of the basic trade-offs are similar to those found in data-space allocation. The unusual overheads for dynamic allocation are that coherency costs are often explicit (§11), that copying position-dependent code means relinking it, and that code generation is one of the costs of servicing a cache miss.

Many systems use the instruction space to hold a transformation of data values. Since the values need to be stored *somewhere*, they might as well be stored in instructions [Thompson 68, Russell 95b]. Likewise, many systems must already convert an external form into an internal form, so the overhead for generating machine instructions instead of some other data form may be relatively small [Thompson 68]. When modified code is paged out, it is not possible (not correct) to put the modified code back in the original file; it is thus either paged out to other storage or it is simply discarded and then "clean" code is paged in from the original file and patched again as needed ("Virtual Memory via Code Generation", 10.3.4).

Caching differs from lazy compilation in that lazily-compiled code cannot, in general, be discarded, whereas caching implies that machine code may be discarded to recover space and then regenerated when it is needed again, Such "forced" reclamation typically implies lazy compilation. A particularly simple cache space reclamation strategy is *throw-away* management, in which space is freed by discarding *all* dynamically-generated code. In contrast, lazily-compiled code may not generally be discarded. Thus, while both schemes can use machine code to provide high performance, caching can also be used to maintain the compactness of the original code form [Brown 76, Deutsch & Schiffman 84, Wakeling 95].

Various systems use code that is nominally sharable, but which is sometimes patched. These systems can use code segments or pages that are copy-on-write [Organick 72, Shaw 74]. Systems that perform code modification on a "global" basis, such as those that "fix" unimplemented instructions by faulting (§10.3.2), can use a trusted agent such as the operating system kernel to perform the patching in a safe way. The modified pages are thus sharable, and processes may also amortize the overhead of the faults [Shaw 74].

Instruction space manipulation may affect real-time processing. For example, when the operating system allocates physical memory it must check if the page has been used for code since the last system-wide instruction cache coherency operation. If so, the kernel initiates a system-wide (interprocessor) cache flush to ensure correctness as the page migrates among processors. Time-critical processes may allocate instruction memory and thus initiate expensive system-wide flushes during the time-critical section. In particular, dynamic linking may cause lazy code allocation, so even applications that complete all linking before entering a time-critical section may still cause code page allocation inside the critical section. To avoid the problem, it is necessary to either statically link the program or explicitly lock the process pages [SGI 94, Mashey 95].

10.5.12 Interdomain Optimizations

For most systems, dynamic optimization takes place within a single protection domain. A few systems have optimized across multiple protection domains, for example by optimizing cross-domain calling conventions and by performing conventional optimizations in one domain based on values supplied by another domain [Massalin 90, Thekkath 91, Massalin 92a, Thekkath & Levy 93] and there are proposals for integrated compiler support [Seltzer et al. 96]. Cross-domain optimizations are complicated by factors such as cross-machine failure modes, heterogeneity ("Portable Binaries", pg. 135), protection and security concerns (§10.5.15) and so on; see Section B.11 for details.

10.5.13 Interface and Implementation Consistency

When code is updated dynamically, the interfaces of the old and new versions may differ, which in turn requires that the callers be updated to call using the new interface convention and/or that calls to the old interface be serviced via some kind of format-converting interprocedure [Segal &

Frieder 89a, Segal & Frieder 89b, Frieder & Segal 91, Segal & Frieder 93]. Correspondingly, several systems profitably use several code fragments with the same interface but different implementation; in particular, the code fragments use different algorithms, rather than being simply different degrees of specialization of one source fragment [Hansen 73, May 87, Massalin & Pu 89, Massalin & Pu 91, Hölzle et al. 91, Hölzle & Ungar 94b]. An advantage of using multiple implementations is that one can be tuned for latency and another for throughput [Hansen 73, Massalin & Pu 91, Hölzle et al. 91, Hölzle & Ungar 94b]. Some systems also require multiple implementations of a code fragment during development so that one is available for reliable use while the new one is being tested [Franz 94] and so on. In all of these cases, a mechanism is required to dynamically choose between the various implementations. Although these issues arise also in static systems, they often require extra care in systems using dynamic code [Appel 94].

10.5.14 Memoizing

When a code generator runs, the resulting code may be entered in a cache tagged by the context in which it was generated, such as parameters to the code generator and the values embedded in the newly-generated code. The code generator may then be *memoized* by having subsequent calls to the code generator check the cache: on a hit, the code generator simply returns the existing code, rather than performing real code generation. This technique has been used in both language translation systems [Henry 87, Hölzle et al. 91, Hölzle & Ungar 94b, Cutts et al. 94] and instruction-set simulators [May 87, Cmelik & Keppel 93, Samuelsson 94].

10.5.15 Mobile Code: Moving Code Between Protection Domains

There are a variety of situations where it is desirable to use *mobile code* that is moved between protection domains, including abstraction embedding ("Abstraction Embedding", pg. 128), databases ("Databases", pg. 131), extensible systems ("Extensible Systems", §10.3.4), network browsers [Col 95, Jav 96] and operating systems ("Operating Systems", pg. 134). Unfortunately, a variety of problems occur when code is moved between physical or logical domains, even though such problems do not usually arise when data is moved across the same boundaries. The problems arise because data is passive, and the interpretation of the data naturally rests with the domain that accepts the data. In contrast, code is active, and transferring control to arbitrary code can result in arbitrary effects. These problems are perhaps most striking in the context of moving code between protection domains because checking the "safety" of incoming code is generally undecidable, but a recipient that executes arbitrary code may be corrupted arbitrarily.

There are four general approaches to moving code across domains safely [Keppel 93b]. One is to use a "limited language" that is less expressive but more easily checked. A second is to push dubious operations back outside of the protection domain so that they cannot damage the recipient. A third is to virtualize any dubious operations and use runtime checks. Virtualizing effectively shifts the domain boundaries so that dubious operations are still executed "outside" of the boundary. However, virtualizing may be more efficient than continually crossing the original domain boundaries [Wahbe et al. 93]. A fourth is to use program verification to reject any code that might be unsafe. Many systems combine several features.

Limited Languages Limited languages are designed so that any expressible program is either automatically safe or may be completely verified before invocation. For example, one operating system implemented only the system call “execute program”. The programming language provided access to a variety of system primitives and provided for a few other kinds of operations such as very limited control flow [Gaines 72]. Likewise, one interprocess communication system used a limited language during session construction in order to describe typical calls that would be made during the session. That language provided information that was used to perform interdomain optimizations such such as register allocation [Thekkath 91, Thekkath & Levy 93]. Another system used mobile code written in a “safe” subset of the system programming language that was then compiled by a trusted compiler [Bershad et al. 95, Hsieh et al. 96]. Language extensions included things such as making interfaces into first-class entities [Hsieh et al. 96].

Outside The Domain Another choice is to accept arbitrary code but execute it outside of the recipient’s protection domain, so that spurious behavior is caught by the existing domain mechanism [Stamos & Gifford 90]. For example, the NeFS distributed file system was implemented so that mobile code was executed on the machine closest to the physical disks, but in a separate process from the one running the file system code [Callaghan 90].[17]

Virtualization Interpreters and pre-execution code-rewrite tools can virtualize dubious operations in order to ensure that erroneous operations can have only limited extent. For example, some interprocess communication systems have allowed requesters to send nearly-arbitrary virtual machine code to the service, and the service then executed the code using a trusted interpreter [Callaghan 90, Falcone 87, Stamos & Gifford 90, Wahbe et al. 93]. Some simulation and tracing tools have performed address translation in order to provide whole-system simulation facilities; in doing so, they also implemented protection. Those tools that performed virtualization earlier and at a coarser grain tended to have better performance [Bedichek 89, Rosenblum et al. 95].

Program Verification Program verifiers [Deutsch 73] can check many constructs and either reject uncertain ones (thus implementing a limited language) or flag them for further scrutiny (e.g., virtualization). For example, the IBM System/360 and System/370 used user-programmable I/O coprocessors. User code was verified before the hardware was allowed to execute it [Baker 95]. One debugging and measurement tool injected user code into protected domains — including the currently-running operating system — and a program verifier ensured both addressing and timing constraints [Deutsch & Grant 71]. Several operating systems would import machine code if it was

[17] In this particular case, the mobile code still had more privileges than a user accessing files via NFS [Sandberg et al. 85], because existing systems did not implement any mechanisms for restricting rights of particular processes [Berman et al. 95].

generated by a trusted compiler performing, for example, type-based analysis in a typesafe language [Stamos & Gifford 90, O'Keefe 94, Bershad et al. 95, Sirer et al. 96]; the machine code was authenticated using either operating system support [O'Keefe 94] or digital signatures [Stamos & Gifford 90, Bershad et al. 95]. A more flexible hybrid strategy virtualizes (interprets) any operations that cannot be validated by the verifier [Deutsch & Grant 71, Wahbe et al. 93, Col 95].

Note that the above schemes do not generally protect against "denial of service" attacks, such as those that simply consume resources and thus prevent other tasks from making forward progress. The difficulty is that it is difficult to determine which mobile code is performing "useful" work and which code is simply wasting resources [Dean et al. 96]. Limiting the execution resources makes it harder to perform useful work, but easier to prevent denial of service attacks [Deutsch & Grant 71].

Nearly all existing implementations of protected mobile code prohibit "approved" code from either generating new code or modifying existing code. These restrictions ensure protection because "approved" code cannot create and execute unapproved code. However, preventing code generation and modification is unnecessarily strict: there are at least two other safe strategies. One strategy is to approve or virtualize both a code generator and the code that it produces, thus ensuring that the generated code is always valid [Fla 94, Hayden 94]. A second strategy is to have acceptance require that all dynamic code generation and modification go via a standard interface that ensures the necessary verification or virtualization is performed (for example, the interface described in Chapter 11).

Systems that export code must be careful about accidentally revealing information about the system that is exporting the code. For example, a system may limit where remote evaluation requests may be sent or may use encryption in order to controlling the remote evaluation environment [Gifford 81].

A variety of such problems and solution techniques also arise when code is moved between name spaces, machine boundaries and implementation instruction sets [Deutsch & Schiffman 84, Shub 90, Theimer & Hayes 92, Bank et al. 94]. See also "Dynamic Version Updating", pg. 131 and Section 10.5.5.

10.5.16 Multi-Level RTCG

Many of the examples discussed in this Chapter have implied that the code generator is itself generated statically. However, a variety of systems have used multi-level RTCG. For example, a bitblt compiler compiler [*sic*] used in Synthesis dynamically generated a bitblt compiler that was optimized around the specifications of the screen expected by the application and the screen provided by the hardware [Massalin 92b]. Likewise, the dynamically-compiled traversal engine for a finite automaton was itself implemented to generate and execute jump instructions that encode automaton state (§10.3.4, §10.3.2). Synthesis contained several examples of coarse-grained dynamic code generation that manipulated code at finer levels, for example by patching jump instructions (e.g., §10.5.20). Multi-level runtime code generation has also been proposed in more general contexts, such as dynamically-generated “little languages” [Draves 95].

Multi-level RTCG may be viewed as a particular instance of incremental specialization [Consel et al. 93] via generation and evaluation of specialized specializers. An alternative is incremental specialization via successive refinement, in which a single code form is updated several times, without being executed between updates. For example, Synthesis generated code using cooperating code generators and then optimized that code using a peephole optimizer (§10.5.17). Alternatively, code may be generated dynamically, executed, and then further patched by static code. For example, the HSS logic simulator generated machine code dynamically to simulate the behavior of a block of gates; faults were simulated by patching the generated code [Barzilai et al. 87].

10.5.17 Peephole Optimization of Machine Code

A runtime compiler may use *peephole optimization* of the generated machine code [McKeeman 65, Massalin 90, Massalin 92a, Fitzhardinge 93, Auslander et al. 95, Auslander et al. 96]. Peephole optimization allows templates to be combined in many ways, while minimizing the number of individual templates required. For example, a peephole optimizer can remove the redundant stores described in Section 2.3.3, without requiring additional templates. A second benefit is that it allows optimization of code produced by multiple cooperating code generators, without requiring the code generators to cooperate on optimization, since peephole optimization can “fix up” the resulting inefficiencies. Peephole optimizers are thus useful in implementing dynamic inlining (“Dynamic Inlining at Dynamic Compile Time”, pg. 133). A third benefit is that a template may encode constants using a general representation that allows for large or general values, and then the peephole optimizer can rewrite instances with small constants or operations that exploit, e.g., power-of-two values, thus achieving the effect of having multiple fixed-format templates. Synthesis used peephole optimization and showed benefit for all three of the above reasons [Massalin 90].

Late optimization provides increased genericity but at the expense of larger code generation costs. For example, the peephole optimizer in Synthesis uses a three-instruction “peephole” and executes tens of instructions per instruction sequence considered [Massalin 90].

10.5.18 Preallocation and Reallocation of Instruction Fields

Template RTCG in Synthesis used statically-preallocated registers, which reduced the cost of runtime code generation. When a template was used in a context where preallocated registers were in-use, the template’s registers were reallocated using symbol table information and a *toggle mask* that transliterated register number encodings from old to new [Massalin 90]. The toggle mask used the identity (old XOR new) XOR old = new. Figure 10.11 shows an example where register r30, encoded as 0b11110 is reallocated to register r5, encoded as 0b00101 by computing the mask 0b11110 XOR 0b00101 = 0b11011 and then aligning the mask to each occurrence of r30 and then XORing it with that word. Toggle masks can also update opcodes, immediates, and other fields in instruction words.

incr:

Assembly	31–26	25–21	20–16	15–0
0: ld F1(c0) → d0:r30	ld	00000	11110	F1
1: add d0:r30,#D01:64 → d1:r30	add	11110	11110	0010 0000
2: st d1:31 → #C00(c1)	st	00001	11110	0000 0000 0000 0000

(a)

```
incr.sym:  sizeof:   12 bytes
           compile:  c0: [0,21]
                     c1: [2,21]
                     C00: [2,0]
           default:  d0: 30 [0,16; 1,21]
                     d1: 30 [1,16; 2,16]
                     D01: [1,0]
```

(b)

```
incr.cpy (dst, n0, n1, N00) {
   dst[0] = incr[0] | (n0 ≪ 21)
   dst[1] = incr[1]
   dst[2] = incr[2] | (n1 ≪ 21) | (N00 ≪ 0)
}

incr.realloc.d0 (dst, n0) {
   dst[0] = dst[0] ⊕ ((n0 ⊕ 30) ≪ 16)
   dst[1] = dst[1] ⊕ ((n0 ⊕ 30) ≪ 21)
}
```

(c)

Figure 10.11: Defaulted register allocation and symbol table information.

A template with both defaulted values and values that are filled in when the template is copied at compile time. In (a), The template incr:, shown in both assembly code and machine code, implements the expression z[C0] = C1 + y→f1. Here, the value of C1 and register allocation for the temporaries t1 and t2 (t1 = y→f1, t2 = C1 + t1) are defaulted, to 64, r30 and r30, respectively. Register allocation for z and y is performed at compile time, and C0 is a dynamic compile-time constant. As shown in (b), the assembler generates symbol table information including the location, representation and default value for each named object. The symbol table information may be used to create a code generator, "incr.cpy", and reallocation routines such as "incr.realloc.d0" as shown in (c). Here, the code generator inserts runtime-allocated registers n0 and n1 for x and y and inserts the runtime value N00 for C0. However, no runtime operations are required for the defaulted entities d0, d1 and D00. If reallocation of, say, d0 is desired, incr.realloc.d0 creates and applies a toggle mask to the two instances of d0. In Synthesis, the code generators and reallocation routines are not usually generated explicitly; instead, their effect is obtained using a symbol table interpreter [Massalin 90].

10.5.19 Representation Coherency

Instruction space modification is useful in part because it allows *dynamic change of representation*, in which information (code or data) is represented in more than one way during its lifetime, being converted between formats as needed for efficient use [Brooks 75, Deutsch & Schiffman 84]. All systems that use dynamic change of representation — whether they use instruction space modification or not — need to deal with two issues. The first is knowing which representation to use and ensuring that it exists when needed. The second is that when a system uses several representations simultaneously, it leads to *representation coherency* issues, in which it is necessary to ensure that modifications to one representation are propagated in a way that ensures all users see a (sufficiently) consistent view. For example, the XTEC version of TECO demand-compiled macros (programs) and saved them for reuse. When the macro source was changed, XTEC would recompile the macros [Crispin 90]. Representation coherency is an example of more general coherency due to duplication; it is useful to single out representation coherency because it helps to pinpoint the source of RTCG-related coherency problems and solutions.

10.5.20 Reuse Rates

Although dynamic code optimization benefits from high reuse rates, there are situations in which very low reuse rates are still profitable. In particular, Synthesis used dynamic optimization in a context where the reuse rate was exactly *one* but optimization was still profitable. RTCG was used to optimize an interrupt handler by replacing a one-of-sixteen indexed jump with an indirect jump through an instruction immediate that was updated on each invocation. The usage was profitable because it reduced the number of registers required and thus save/restore overhead, eliminated a data reference that usually missed, and because instruction cache flushing is performed very cheaply using code placement [Massalin 91, Massalin & Pu 92], (§10.5.4).

10.5.21 Sharing

Most systems using RTCG do not share the dynamically-generated code across protection domains. One reason is that dynamically-generated code may embed domain-specific values and sharing leads to erroneous results [Shaw 74]. Another reason is that the recipient of the code must either trust the code generator that created (or modified) it, or it must treat the code as unsafe (§10.5.15). For example, Oak patches indirect references into direct references based on the context of the thread that executes it; patched code thus cannot be shared among threads [Gosling 95]. Conversely, Oberon code is produced by a trusted compiler and may be shared on a system-wide basis [Franz 94].

10.6 Historical Problems

This section describes a variety of technical troubles that have dogged RTCG implementations, leading to a variety of problems with clarity, debugging, portability, sharing and so on.

10.6.1 Cache Coherency

Traditionally, a small number of applications have had to deal with instruction cache coherency issues. As a result, the operating system and architectural interfaces are relatively under-specified and under-implemented. Here are some examples where problems have cropped up:

- It has been alleged that early versions of some commercial operating systems did not allow user-level code to perform instruction cache coherency. These vendors later added coherency primitives because of commercial applications that used dynamic linking, dynamic generation and garbage collection of code.

- One APL system generated virtual machine code instead of native machine code only because the target machine prohibited efficient instruction space modification [Johnston 79].
- In early versions of DEC Ultrix, the raw kernel trap for instruction cache coherency was available but the C-callable "wrapper" was not [Keppel 91].
- In early versions of DEC OSF/1, executing a writable code page had the effect of (silently) making the page read-only, so that later writes would fault (§3.4).
- Early members of the Intel 80x86 processor family made modest use of self-modifying code and relied on hardware coherency beyond a maximum prefetch of 16 bytes of instructions. In the Pentium design, self-modifying code was supported with special instruction cache consistency hardware. One application broke when run on the Pentium because it modified an instruction that was assumed to have been prefetched, so that the unmodified version would always be executed. On the Pentium, the modification triggered immediate coherency and so the new instruction was always executed [Colwell 95].

10.6.2 Debugging

Debugging programs that use instruction space modification is sometimes difficult: even well-structured systems do not typically represent code modification symbolically. Worse, unstructured modification is downright unpredictable about how and when the code changed, making it difficult to reason about the program.

The results of unintended code modification are often less predictable than the results of unintended data modification — even integral data is in some sense continuous, while instructions are not [Mead 90]. For example, accidentally incrementing a data value will cause an erroneous result, but the answer is usually "close". In contrast, incrementing an `add` instruction may very well turn it into a `jump` that lands precipitously and allows pervasive errors to accumulate before one finally stops the program. Thus, instructions are much less "continuous", much more sensitive to value errors, and the resulting errors are often much harder to debug.

Even intended code modifications may cause debugging problems. One problem is that allowing code modification provides a new mechanism for implementing bugs. For example, the PDP–1 `execute` instruction ("Execute", pg. 118) provided another mechanism for manufacturing a non-interruptable infinite loop [Dig 62, Eggers et al. 96].

because code modification is generally not signaled to the debugger and may thus cause the debugger and debuggee state to disagree [Evans & Darley 65]. Likewise, debugger code manipulation may interfere with program code manipulation (§10.6.9).

Hardware caching structures also complicate matters, because the memory image seen by the programmer is different than the image executed by a processor reading from an incoherent cache.

A particularly notorious example is the Control Data Corporation CDC 6600, which used a non-coherent 16-entry *branch back cache* (BBC). The BBC was a queue of the last 16 instructions executed. Code from the BBC executed faster than code that was fetched from memory. Unconditional jumps flushed the cache, while conditional branches did not. Thus, a standard debugging technique was to augment nonflushing branch instructions with flushing jump instructions until the problem disappeared, then reset the other branches to be nonflushing [Thornton 64, Hughes 94, Littlefield 94]. The CDC 7600 was largely compatible but used a normal cache instead; programs that were optimized to "know" about the BBC then broke when run under the 7600's different caching mechanism [Albaugh 95].

Even some modern systems have obscure interactions with dynamically-generated code. For example, some versions of gdb, the GNU debugger [Stallman 87], have assumed that code does not change. The assumption reduces the cost of interdomain transfers but leads to problems when examining code that has in fact changed. Debuggers have also interacted with code update tools such as dynamic linkers. For example, the dbx debugger [Linton 90] could not register an interest in particular symbols or address ranges with the linker and thus could not set breakpoints in code until after it was loaded and linked [Sun 94]. Worse, invoking the dynamic linker would cause it to execute some application code before the debugger regained control, making it essentially impossible to set breakpoints in some dynamically-linked code [Sun 94].

10.6.3 Emulating Instruction-Space Modification

Supporting self-modifying code complicates processor implementations; it also complicates machine-code analysis and translation tools. For example, early Intel 8086 processors had hardware coherency and applications depended on coherency [Leinecker 93]; as a result, modern members of the family include fairly sophisticated machinery to maintain hardware coherency while in fact providing instruction caches, deep prefetch pipelines, out-of-order issue and completion, and so on [Pollack 91]. As a result, code modification has a bad name among the processor architects and implementors who are required to deal with such details. Note, however, that *all* general-purpose processors must implement some kind of cache coherency (§9.1). Thus, the real problem with code modification is ensuring that all uses follow a common interface (§11) and that the coherency costs — which may vary widely from platform to platform — are available to those considering or using instruction space modification [Chase et al. 95, Mashey 95].

These same code modification issues affect tools that translate program binaries across architectures [Wharton 91, Wharton 92, Cmelik & Keppel 95]. Where an existing system performs instruction-space modification, an analyzer or emulator for that system must successfully operate in the face of instruction-space modification [Bedichek 89, Cmelik & Keppel 93, ARD 94, Cmelik & Keppel 94, Fla 94, Hayden 94, Bedichek 95a, Cmelik & Keppel 95]. In general, determining statically what is code and what is data requires a solution to the halting problem. Although practical solutions exist [Fla 94, Hayden 94], static analysis and translation tools are confused by some uses of dynamic code generation, instruction space modification, dynamic linking, and so on. From an emulation and analysis standpoint, the best situation is when the target system that is being analyzed or emulated uses an interface to explicitly signal code changes [Mot 89a, Mot 90, Keppel 91, Sun 91, Dig 92a, Sites 92]. Otherwise, the system must use extra analysis in order to detect instruction space changes [ARD 94, Cmelik & Keppel 93].

10.6.4 Cache Performance

Dynamically-compiled code is sometimes larger, and sometimes by enough that the space costs make it slower than a statically-compiled implementation. For example, in one instruction-set simulator, servicing instruction cache misses (at 13 cycles per miss) accounted for as much as 75% of the simulator's execution time; a comparable decode-and-dispatch interpreter was theoretically three times slower, but in practice was faster because the cache miss ratio was so much better [Pittman 95].

Dynamically-linked routines may appear at different virtual addresses from run to run and may also cause variations in physical assignment. Thus, for example, on SPARC machines that lack floating-point hardware, dynamically linking the mul routine into the "nasker" SPEC benchmark led to as much as a factor of three performance variation depending on whether it was linked to an address that conflicted with the address of the innermost loop [Lucco 93]. Of course, caching and code placement is significant even for statically-linked code [McFarling 89, Pettis & Hansen 90, McFarling 91]. For example, inserting three nop instructions before each store instruction, in order to pseudo-randomly perturb code placement, may lead to as much as a 40% *decrease* in execution time, even though adding nops substantially increase the number of instructions executed by the program [Lucco 93]. The important observation here, however, is that dynamic linking may lead to substantial performance variations that are never observed during testing. (Conversely, it is also possible that RTCG could improve code placement; see §B.6).

10.6.5 Increasing Portability

With the advent of compilers and the proliferation of architectures, portability became more important. Although some compilers generated simple instruction-modifying sequences [Organick 72, Albaugh 95], most sophisticated uses were generated by hand and it made little sense to expend energy on teaching compilers to perform dynamic code optimizations when large benefits were still to be had from improving the static transformations.

A second issue was that certain constructs were not reliable or were not reliably profitable across a range of machine implementations. Here are some specific examples:

- *Instruction cache coherency costs:* On the RS/6000, the cost of instruction cache coherency eliminated the performance benefits of receiver caching for Smalltalk ("Operand Patching – Faster Addressing Modes", pg. 120), though receiver caching was typically profitable on other platforms and although the cost of instruction cache coherency was still low enough that compiling interpretation was still profitable ("Compiling Interpreters", pg. 129) [Deutsch 90].

- *Memory timing:* On some PDP-10 models, executing from low locations was fastest. On others, it was much slower. Thus software-controlled caching was only profitable on some PDP-10 implementations [Baker 92b, Crispin 95, Eager 95, Greenblatt 95, Stallman 95, White 95], ("Software-Controlled Caching" pg. 128).

- *Relative instruction speeds:* The relative speeds for some instructions varied widely among processor implementations, thus affecting the relative performance of static and RTCG alternatives. For example, one pattern matching engine generated Intel 80x86 machine code dynamically [Aus 90]; however, the generated code used complex instructions that were relatively slower on newer processor implementations, thus eliminating the potential RTCG improvement [Chandross 95].

- *Repetition:* Some processor implementations have implemented block-move instructions by executing the instruction repeatedly, with termination controlled by a register value. However, if the instruction overwrote itself the opcode would become, say, a do-nothing `nop` instruction, thus prematurely ending the block move. Further, on machines with variable-length instructions, the operand to the block-move instruction would then be executed as an instruction [Collier 92b].

10.6.6 Justification

Many early uses of instruction space modification were to save space. As first-level memories became larger, size became relatively unimportant to program performance. Another change was the development of technology that allowed more regular and more sophisticated instruction encodings, thus eliminating some traditional uses of code modification (see "Manufactured Addressing Modes", pg. 119). In turn, some of the remaining uses were not justified by performance; these could be held up as examples of bad programming, while the "bad name" of instruction space modification meant that even the good uses were largely hidden from view. For example, one report states that database queries are much faster if the query is dynamically optimized and compiled [Chamberlin et al. 81a], yet there are no published quantitative analysis of when, how much and why it is faster.

10.6.7 Protection

The problems with mobile code (§10.5.15) are not new. For example, the IBM System/360 and System/370 used I/O coprocessors that were programmable by the user using "channel command words" [Padess 64]. In early machines, the channel command words were an unnoticed security hole, though later versions of the operating system solved the problem by verifying the user code (§10.5.15) before allowing the hardware to execute it [Baker 95]. Unprotected mobile code is used in several contemporary systems but it is also clearly documented as dangerous [Wilkinson et al. 90]. For these sorts of reasons, moving machine code has largely fallen out of favor. An unfortunate consequence, however, is that many of the related lessons are not common knowledge, and so old problems continue to reappear, for example, as viruses encoded as virtual machine code [Ducklin 95, War 95, Weston 95, Graves 96].

10.6.8 Race Conditions During Code Modification

Code modification in a multi-threaded environment is subject to races [Landis et al. 62]. In the Parasight performance debugger, several instructions were modified to plant a breakpoint in code that could be executed during modification. The published implementation patched the first instruction with a "self-looping" instruction, with the goal of guarding subsequent instructions in the basic block [Aral & Gertner 88]. However, threads that were already executing in the sequence could delayed arbitrarily due to, e.g., page faults or preemption. Thus, a thread could sleep across a code region change and, upon waking, execute "new" instructions, thus seeing an inconsistent instruction stream [Kessler 90b].

10.6.9 Sharing

Modifiable code cannot generally be shared between threads of control or between protection domains because its behavior changes over time. That is, modifiable code is not generally *reentrant*. In addition, multithreaded of sharing may lead to code modification races.

Sharing occurs in a variety of situations. For example, debuggers patch debuggee code [Gill 51, Kessler 90a, Cmelik 90, Paxson 90b, Chase & Hood 91, Wahbe 92, Bedichek 92, Gross et al. 95] and share the program text. The sharing leads to a variety of restrictions: debuggers cannot safely plant breakpoints where the target instruction might be modified or used as data [Gross et al. 95], or in a variety of other machine-dependent situations depending on accumulator contents, branch delay slots, and so on [Kessler 90a, Gross et al. 95]. Consistent use of a structured interface to code modification (§11) can eliminate some of these problems by allowing the debugger to monitor all instruction space changes and thus re-plant or update breakpoint instructions as needed.

In multiprogrammed machines, sharing typically reduces memory use because concurrent invocations of a code fragment can share a single copy of the code. Thus, loss of sharing can increase primary memory usage.[18] In additions, unshared code may also need to be duplicated on secondary storage.

Finally, sharing of modifiable code has even more potential for security leaks than mobile code (§10.5.15). Mobile code is a fixed "snapshot" of code. In contrast, shared code may be updated at any time while it is being executed, and thus checking and execution must occur as an atomic operation. For example, debuggers often patch the debuggee using a operating system call. In some early versions of Unix, patches made by one user of a program were seen by all, violating both security and protection.

---

[18] The approach of Part I was to use mostly shared code, and only the key inner loops were implemented as private (unshareable) code. Where the heavily-specialized static code was large and multiprogrammed use of the application was modest, using one small private code fragment per invocation was smaller overall than using a single large shared fragment.

10.6.10 Terminology

Polite names for instruction-space modification are relatively new. For example "Runtime Code Generation" appears to be a relatively recent name, perhaps having appeared as recently as 1988 [Sloman 88].[19] More often, instruction space modification has been referred to, even by its practitioners, as "difficult" [Koopman 89], "hacky" [Locanthi 87] and "primeval" [Kessler 90a]. It has even inspired its own terminology, including:

> **live data** n. **1** *Data that is written to be interpreted and takes over program flow when triggered by some un–obvious operation, such as viewing it. One use of such hacks is to break security. For example, some smart terminals have commands that allow one to download strings to program keys; this can be used to write live data that, when listed to the terminal, infects it with a security-breaking virus that is triggered the next time a hapless user strikes that key. For another, there are some well-known bugs in [the* vi *text editor] that allow certain texts to send arbitrary commands back to the machine when they are simply viewed.* **2** *In C code, data that includes pointers to function hooks (executable code).* **3** *An object, such as a trampoline, that is constructed on the fly by a program and intended to be executed as code.*
>
> –The New Hacker's Dictionary [Raymond 93]
>
> **scrozzle** `/skroz'l/` vt. *Used when a self-modifying code segment runs incorrectly and corrupts the running program or vital data. "The damn compiler scrozzled itself again!"*
>
> –The New Hacker's Dictionary [Raymond 93]
>
> **trampoline** n. *An incredibly hairy technique, found in some [high-level language] and program-overlay implementations (e.g., on the MacIntosh), that involves on-the-fly generation of small executable (and, likely as not, self-modifying) code objects to do indirection between code sections. These pieces of live data are called "trampolines". Trampolines are notoriously difficult to understand in action; in fact, it is said by those who use this term that the trampoline that doesn't bend your brain is not the true trampoline.*
>
> –The New Hacker's Dictionary [Raymond 93]

Note, however, that "trampoline" has evolved so that in modern parlance a trampoline is any code which redirects transparently to the caller; it need not be dynamically-generated or self-modifying.

10.6.11 Tool Design

In some cases, it *ought* to be simple to use RTCG, but actual usage is limited by the programming tools. For example, compilers may be reused as command language interpreters using source-to-source translation [Goodman 88] or by defining the command language to be one for which there is an existing compiler [Massalin 90], also allowing user-selected extension languages ("Extensible Systems", pg. 132). However, reusing existing compilers is often difficult because both their front and back ends are optimized around batch operation. For example, many tools use disk files for their input and output instead of memory [Fyfe et al. 91], and even caching file systems typically promise that data is written to disk before the write operation completes, thus delaying completion of the writes. Likewise, many conventional tools, such as loaders, are not arranged to use only symbol tables built in to program files [Crowe 87]. Reuse is also limited because some low-level tools, such as static and dynamic linkers, vary from system to system [Goodman 88].

[19] The term "breakeven point" had appeared by 1991 [Heering et al. 91]; the concept appears much earlier (§10.5.2).

Many code analysis and modification tools, including debuggers, interoperate poorly with other code modification tools. The root of the problem is that most code modification tools do not provide a clean mechanism by which each tool can inform the others about instruction space changes. For example, software fault isolation (SFI) dramatically improves the utility of dynamic linking, but explicitly disallows code modification, including dynamic linking performed by other agents [Wahbe 92]. SFI would be more widely applicable if it instead insisted that all code modification took place using a standardized interface that guaranteed that all code would be checked (e.g., Chapter 11).. Likewise, debuggers often cannot set breakpoints in dynamically-linked code until that code is actually linked [Chase & Hood 91, Sun 94], because the debugger cannot register its interest in a given symbol or address range, and thus cannot gain control at the right time (see "Dynamic Linking", pg. 141). Debuggers also fail when the debuggee modifies an instruction sequence containing a breakpoint, potentially terminating the debuggee [Sun 94]. In all of these cases, the central problem is that each tool has an independent view of instruction space modification and thus the tools cannot know when or how to react to instruction space modifications.

Even where the desired functionality is available, tool limitations can lead to mental "gear-shifting" overhead that is caused by artificial divisions between compilation and execution environments [Deutsch & Taft 80].

10.6.12 Sociology – Perspective and Mystique

A variety of systems have avoided instruction-space modification either from a unquestioning belief that dynamic machine code generation was necessarily bad, because the tools to support it were lacking, or because instruction space modification was not a standard technique and thus simply never occurred to the authors of the systems.

> *[D]ynamic translation converts a convenient (compact, easily modified, or easily displayed) representation to one that can be quickly interpreted.* (emphasis added)
>
> –[Lampson 84], page 20

> *[A Burroughs] B6700* job ... *consists of a time-invariant algorithm ... In a more advanced view of a job, new code components might be attached to the algorithm during the course of its execution but no component would vary (internally) in time.*
>
> –[Organick 73], page 9

> ldb *owes its retargetability to ... using a machine-independent embedded interpreter ...* ldb *reuses existing compiler functions by having the compiler [dynamically] emit PostScript code that* ldb *later interprets*
>
> –[Ramsey & Hanson 92]

For example, dynamic translation would provide even better performance if the final form was executed directly by the hardware instead of being interpreted. Similarly, in the B6700 system, it was a fine line between adding components and modifying them; for example, the effect of modification could be achieved by invoking the "modified" part via a dynamically-linked procedure call, but possibly at the loss of efficiency because of the out-of-line calls. In an ideal system, the cost of modification would simply have been made explicit. Finally, ldb's retargetability comes at the price of using an extra PostScript^{TM} target for the compiler and at the cost of supporting an extra interpreter. In contrast, ldb could have simply called the compiler, had it generate machine code, and then dynamically linked that into the debugger [Paxson 90b, Fyfe et al. 91].

Another classic "mystical" objection to instruction space modification is that two calls may return different values. That is, "f(a) == f(a)" may evaluate to false. However, the same effect is achieved using per-routine variables whose value persists across call. Although per-routine persistent variables are gradually dropping from favor — exactly because they cause reentrancy problems — it is worth noting that the practice for data lags the practice of code by some thirty years.

Conversely, the attitude of "self-modifying code is bad" has argued against even reasonable uses of RTCG where "f(a) == f(a)" is always true [Mody 92].

Indeed, although self-modification of code has long been frowned upon, self-modification of byte-codes is evidently considered "respectable" because of its data-side representation [Turner 79, Hughes 82, Hudak & Kranz 84, Griswold 89, Gosling 95], despite the fact that modifying virtual machine code introduces exactly the same sharing and reentrancy problems in data space as it does when used in code space.

To summarize, instruction-space modification presents both problems and opportunities. However, instruction-space modification was thrown out "wholesale", prohibiting all cases in order to avoid the hard ones. Unfortunately, one consequence is that some important techniques and lessons — including several that apply to data as well as code — are not widely known. Conversely, instruction-space modification has a bad name, so many systems that could have used it were instead built to avoid it — because of its reputation, and not for technical reasons.

> *This problem could be solved by dynamically creating short sequences of code as needed to act as entry points for the given objects. However, we preferred a solution that avoided the allure of fabricating code at runtime*
>
> –[Hamilton & Kougiouris 93], pg. 3

> *"Several applications ... would benefit greatly from the ability to build and execute programs at run-time. Statically translated programs, though, execute faster."*
>
> –[Horowitz 88], pg. 98.

> *[Runtime code generation] is like handy household uses of Napalm. There may be some, but I don't want to know what they are!*
>
> –Wendy Thrash

10.7 Not Quite N-Code RTCG

A variety of systems perform RTCG using virtual machine code instead of native machine code; a variety of other systems manipulate native machine code very close to runtime. Although the focus of this dissertation is on native code RTCG, these other systems deserve brief mention because they clarify some of the uses and motivations for RTCG and some of the implementation alternatives.

Artificial Life Artificial life simulations use a multitude of programs executing in a shared space; an executing program may attempt to move or reproduce itself or may attempt to change memory that holds instructions for other programs [Ray 92]. As a program executes, other programs and the basic execution machinery also occasionally introduce changes into the program. The instruction set is designed so that programs are relatively tolerant of errors [Mead 90, Ray 92]. Through permutation and self-selection, a variety of phenomena arise, including both parasites and classic compiler optimizations [Ray 92].

Binding Times Close to Runtime Some systems have moved traditional compiler functions closer to run time in order to get the benefits of later binding; for example, linking [Wall 89], code generation [Dahlby et al. 78, Koldinger 88, Andrews & Sand 92, Sites 93, Berg et al. 95]. Conversely, some systems have moved "dynamic" operations to just before runtime in order to reduce the runtime cost [Maeda 90, Kempf & Kessler 92a, Kempf & Kessler 92b, Nelson & Hamilton 93a, Nelson & Hamilton 93b]. Like RTCG, these systems have had to deal with representation issues (§10.2) but have avoided RTCG's instruction-space space coherency issues (Chapter 11).

Choosing Virtual or Native Implementation Various systems exploit the close relation between virtual and real machine code. For example, mixing machine code and virtual machine code may provide most of the size benefits of the latter with most of the performance of the former [Pittman 87, Pittman 95]. Alternatively, a complex "virtual" machine may be implemented so that large or low-usage parts of the interpreter are in machine code while the highest-use parts are implemented in microcode [Hassitt & Lyon 76, Deutsch 83] or hardware [Lewis et al. 86]. In one case, careful implementation meant that typically only 5-15% of the total execution time was spent outside of the microcode [Deutsch 83]

Combinators Some programs may be compiled to a graph representation where vertices are chosen from a small set of operations called *combinators*; programs are executed using *graph reduction* in which regular rules are applied to perform in-place graph rewriting and simplification [Turner 79, Hughes 82, Hudak & Kranz 84]. Since the graph is gradually rewritten, combinator evaluation may be viewed as dynamic self-optimization. One optimization is to produce static machine code that encodes the original graph and executes it without reduction. The machine code performs no self-modification, but the elimination of interpretation overhead means that it is nonetheless typically faster [Appel 88]. A second optimization is to implement the graph using machine code that *is* self-rewriting (see "Value-Based Optimization", pg. 139).

Computational Models Several computational models have incorporated notions of dynamic code.

Turing's 1936 computation model, now known as the *Turing machine*, uses a single store. Clearly, it is arguable whether commands in the store are "instructions" or state. However, there is a set of "universal" Turing machines that may have very simple static control and yet can emulate all other Turing machines and general-purpose runtime-programmable machines [Hopcroft & Ullman 79]. A Turing machine may be programmed by altering the store, so the symbols may be viewed as instructions. Since the store may be modified arbitrarily during a single run, the Turing machine model supports RTCG.

The duality of code and data was stated during early consideration of stored-program computers. Although the focus was on static programs, it was realized that instructions and data were not fundamentally different and thus if they used the same system-level encoding (e.g., 1's and 0's) a single memory could hold either code or data, provided there was a way of distinguishing them [von Neumann 45, Burks et al. 46, Knuth 70].

Kinematic automata define a computational model that is a generalization of cellular automata. Kinematic automata allow elements to change, move and reproduce [Sosič & Johnson 94].

The reflective language model provides for explicit manipulation of code [Smith 84, Wand 88, Craske 91, Cutts et al. 94].

Data Much as it makes sense to speak of native machine code and virtual machine code, it makes sense to speak of native and virtual machine data formats.

Data in the Instruction Space Many compilers embed read-only constants in an instruction segment. Some programs also read other instructions or parts of other instructions as data, if some instruction conveniently encoded the desired data [Raymond 93]. Historically, a variety of problems have occurred because, for example, a caller passed the address of a "constant" and the callee quietly modified the given location. Many of these systems then broke as compilers were developed that placed read-only values in the immutable text segment [Stallman 89, Cannon et al. 90].

Data Specialization Many of the same analyses used for VSO can also be used for performing *data specialization* [Knoblock & Ruf 96a, Knoblock & Ruf 96b]; like VSO, the goal is to perform loop-invariant code motion. Data specialization uses static code generation and is thus closer to traditional loop-invariant code motion. The goal of data specialization is to produce a "loader" phase which performs some computations and stores the results in a compiler-generated cache that is read repeatedly by a streamlined "reader" phase in the inner loop. As with VSO, one goal is to determine when the specialization variables change and then re-run the "loader" phase. An advantage of data specialization is that generating cache values is typically much faster than generating code; a disadvantage is that data specialization does not provide for as much optimization without static code size expansion (see Section 1.1).

Debugging Self-modification introduces the same debugging problems whether it is applied to native or virtual machine code. For example, the database for an early global weather service was rebuilt after crashes using self-modifying job control code. Even though it was high-level code, debugging grew intractable with changes to more than 1-2 subsequent instructions [Blinks 95].

```
#!/util/bin/perl
$s=q@#!/util/bin/perl
$s=q
```

Figure 10.12: A Perl program fragment that, when run, produces itself as its output.

Games A variety of games embed notions of mutable code or expose the duality of code and data.

The game *Core War* has two programmers write code fragments that attempt to stay running while causing the competitor's fragment to stop. Execution of the two fragments is interleaved, so that each has a "fair chance" to affect the other. The code fragments typically either self-replicate and/or attempt to write over parts of the instruction space where the other program might be executing. Core War is commonly played on a special virtual machine [Dewdney 84, Strack 95].

A folklore programming challenge is to write a (non-trivial) program that, when executed, prints itself [Thompson 84, Stallman 93]; an example Perl program is shown in Figure 10.12 [Boyd].

A second folklore programming challenge is to write a program whose meaning depends on the engine that executes it, for example, Figure 9.1 (pg. 109).

A third folklore challenge is to write CISC instruction sequences that appear to be different programs depending on the entry address; for example, a multi–byte instruction can be interpreted as several instructions, depending on the starting address.

Incremental Reexecution *Incremental reexecution* arises when a program is re–executed and either the code or data is very nearly the same as for the previous invocation. Minor code changes may arise, for example, during an edit-compile-debug cycle [Karinthi & Weiser 87, Yellin & Stroom 88].

Language-Level View Many languages provide some mechanism in which the user's programming model is that code is generated programatically at runtime[20] and then executed.

- One version of BASIC formalizes the conversion between data and code [Mitchell 70].
- COBOL has an `ALTER` verb that causes a labeled jump instruction to be updated so that it causes transfer to a new destination.
- Several languages use delayed evaluation operators, including SNOBOL4 [Griswold et al. 71] and tick-C, which provides additional static typing and scoping operators [Engler et al. 95, Engler et al. 96]. One ML variant uses currying to indicate delayed evaluation [Leone & Lee 96b, Leone & Lee 96a].
- The user interface shell `es` supports first-class code in order to order to make the interface more consistent [Haar & Rakizis 93].

[20]Many languages provide language-level facilities for programatically generating code at static compile time; for example, C and LISP macros.

- The eval operator is provided by several languages including LISP [McCarthy 78, Rees & IV 82, Falcone 87] and Perl [Wall & Schwartz 91].

  > *The fourth reason for using eval is, paradoxically, efficiency. There are certain constructs that the Perl compiler is very good at optimizing, provided it can determine what it's supposed to be doing at compile time. In particular, if you're searching for multiple patterns that aren't known until run time, it's much more efficient to build up a loop containing multiple pattern match commands, and then eval that, than it is to try to search using variable interpolations in the pattern, which force recompilation ... each time the pattern is used.*
  >
  > –[Wall & Schwartz 91], page 61

- Many functional programming languages and some other languages support first-class functions and thus allow dynamic composition of operations. It is thus possible for programs to create new routines using all of the language's basic operations [Atkinson & Morrison 85, Mody 92].

- LISP maintains little distinction between the programatic representation of functions and of data [McCarthy 78, Rees & IV 82, Falcone 87]. The similarity is exploited to ease construction of evaluators, compilers, and translators in LISP [Rees & IV 82].

- Prolog's assert and retract builtins provide a mechanism by which rules may be added and removed from the program database [Clocksin & Mellish 84]; traditional machine-code compilers for Prolog do not implement assert and retract because of their potentially global effect on the statically-generated code. However, quick incremental analysis restricts the scope of changes and thus enables dynamic code generation to support the code-altering builtins (§10.5.9).

- If a language is extended by allowing runtime compilation, then operations designed for use with constants will also accept runtime values [Mitchell 70]. Note the correspondence to manufactured addressing modes ("Manufactured Addressing Modes", pg. 119).

Neural Networks Neural networks are "programmed" by providing an input data set and a desired answer for each item in the input data set. Each new training input may be seen as a program update. Larger networks, in particular, learn as a function of the current learning function and thus may be seen to perform self-modification [Mead 90].

Parsing Parsers normally perform reductions and then call semantic action routines. Many of the semantic actions are easily represented as specialized parse rules e.g., ident = expr has a semantic action that checks the type. An alternative is to build a parser that embeds the type tests, for example, by building rules such as int-ident = int-expr, char-ident = char-expr, and so on [Burshteyn 90]. A disadvantage of this approach is that there are a large number of rules, and some types aren't known until runtime. Thus, a solution is to add and delete grammar productions in a controlled way during parsing. For example, int foo creates rules for ident-foo = int-expr [Heering et al. 89, Burshteyn 90, Heering et al. 90].

Partial Evaluation Partial evaluation is a mechanism by which general code is specialized based on the values of particular parameters [Haraldsson 77, Haraldsson 78, Emanuelson 80, Haraldsson 80, Jones 88, Consel & Danvy 89, Jones et al. 89, Berlin & Weise 90, Consel 90, Consel & Danvy 90, Pagan 90, Par 91, Consel et al. 93]. Although partial evaluation is conceptually simple, a variety of issues arise in practice including code growth and termination.

Value-based dynamic optimization ("Value-Based Dynamic Optimization", pg. 139) may be viewed as partial evaluation applied at runtime. Offline specialization also provides speedups. For example, offline specialization of scientific codes around program data and using data-dependent parallelism yielded one to two orders of magnitude speedup [Berlin & Weise 90].

Retargeting Interactive Compilers *Retargeting* is the general procedure of moving code from one execution environment to another. Retargeting of static compilers is often accomplished by first building an intermediate version of the compiler that runs on the old system and generates code for the new system and then using the intermediate version to compile a final version that runs on and generates code for the new system. Retargeting of interactive compiler is more complex because old and new code must somehow run together. For example, an interactive compiler may call itself and then execute the resulting code. During retargeting, however, that may cause the "intermediate version" to try to execute code for the new machine, even though it is still executing on the old machine. Nonetheless, careful decomposition and ordering of the retargeting makes it possible to retarget interactive systems [Appel 94].

Runtime Compilation Static loop parallelization is difficult when element indexing uses indirection, for example array references of the form A[B[i]] where the values in B are set at runtime. One strategy is to scan B at runtime but before loop entry; data-dependence information from B is used to parallelize the loop. The technique summarizes or "compiles" information from B, hence the name *runtime compilation*; however, no code generation is performed [Saltz et al. 90, Leung 92].

10.8 Related Work

> *Everything that ever happens in computer systems is like something that happened at MIT twenty years ago.*
>
> –Jeff Chase

This section briefly compares the work in Part I of this dissertation with the systems described in this chapter. Part I focuses on a few key features: static recognition of sites where VSO might be applied profitably; transformation of static ode constructs into fast application-specific runtime compilers; static typing of both the VSO'd code and the "strawman" used for comparison, in order to show that VSO is indeed performing optimizations that fundamentally are not possible using static compilation; running the code on real hardware to show measured speedups; and, finally, considering the effects of varying runtime optimization, with the surprising result that simple optimizations typically produce code as fast as that produced using a slower optimizing code generator.

10.8.1 General Features

The work in Part I contrasts the speedups of value-based optimizations across a variety of processor architectures. Except for a two-architecture study using sparse matrix multiply [Engler & Proebsting 94], all other studies were confied to a single architecture. Most previous studies of value-based optimization in statically-typed languages were performed on CISC architectures. The studies in this dissertation, as well as a few others [Leone & Lee 94, Engler & Proebsting 94, Auslander et al. 95, Leone & Lee 95, Auslander et al. 96, Consel & Noel 96, Engler 96, Leone & Lee 96b, Leone & Lee 96a] show that benefits also apply to RISC architectures. Part I of this dissertation is the first study of VSO on a processor with superscalar issue.

The work in Part I studies the effects of varying runtime optimization levels when dynamically compiling statically-typed code. Previous studies have studied the effects of varying optimization on statically-typed programs only where the initial code quality is low, for example, in fast-turnaround edit-comple-debug systems [Hansen 73]. Previous studies have also studied the effects of optimization level on implementations of dynamically-typed languages. These show benefit from raising the level of optimization because each type-based optimization is applied many places, thus preventing extensive site-specific specialization of the optimizers [Hölzle et al. 91, Chambers 92, Hölzle & Ungar 94b, Hölzle 94].[21]

A variety of systems perform value-based dynamic optimization. These systems rely on programmer annotations to identify optimization sites and direct automated generation of the application-specific compilers. One system is a proposal, so no performance results are available [Appel 88]. The remaining systems produce either template compilers [Leone & Lee 93, Leone & Lee 94, Auslander et al. 95, Leone & Lee 95, Auslander et al. 96, Consel & Noel 96, Engler 96, Leone & Lee 96b, Leone & Lee 96a] or tree compilers [Engler & Proebsting 94, Engler et al. 95, Engler et al. 96]. Few directly consider breakeven points [Auslander et al. 96] and none consider the effects of varying the degree of runtime optimization. Results include runtime compilation costs under 400 instructions per generated instruction for a tree compiler [Engler & Proebsting 94] and under ten instructions per generated instruction for a template compiler [Leone & Lee 94, Leone & Lee 95, Engler 96, Consel & Noel 96, Leone & Lee 95]; in all cases the runtime-generated code is of sufficient quality that it shows speedups over statically-optimized code.

A variety of systems implemented value-based optimization for particular applications, but without exploring value-based optimization as a general technique that could be applied to multiple applications. Examples of particular applications include the fine-grained modifications described in Sections 10.3.1, 10.3.2 and 10.3.3, as well as larger-grained optimizations used for bitblt [Pike et al. 85, Locanthi 87, Miranda 87, Foley et al. 90, May 92, Kaepplein 95]; cache simulation [Przybylski 88, Przybylski et al. 88]; operating system communication, scheduling and I/O [Pu et al. 87, Massalin & Pu 88, Pu et al. 88, Massalin & Pu 89, Massalin & Pu 91, Massalin 92a, Pu & Walpole 93, Pu et al. 94, Bershad et al. 95, Engler 96]; regular expression and string matching [Beeler et al. 72,

[21] In principle, many dynamic type optimizations could use fast application-specific compilers, thus eliminating much of the benefits of sophisticated runtime analysis. However, to achieve good code quality it would be necessary to use, e.g., one application-specific compiler for each optimization site, leading to an explosion of runtime compilers.

Thompson 68]; and sorting [Levine 89].

10.8.2 Specific Applications

This subsection briefly consideres the applications used for workload used in this dissertation, considering both static-code and RTCG related work.

Bitblt The static code implementation of bitblt used in Part I is based on Locanthi's implementation [Locanthi 87]. The most significant change is that Locanthi's implementation takes advantage of a Motorola 680x0 instruction that zeros a value when it is shifted by the word width. The machines used in Part I instead leave such values undefined, thus requiring extra code — and increasing the number of specialized cases — for situations in which one register is shifted by the word width (see "if (lshift == 0)" in Figure 4.1). The VSO'd version of bitblt was derived from the static-code version.

A previous version of the static-code bitblt [Keppel et al. 93] had a performance bug that overstated VSO's asymptotic benefit.

Cache Simulator The VSO'd version of the cache simulator was inspired by Przybylski's simulator which was built on his observation that some cache simulation parameters varied from run to run but could be effectively "compiled out" [Przybylski 88, Przybylski et al. 88]. In Przybylski's simulator, the simulation parameters were used to programmatically generate C source code definitions that were constants instead of variables. Then, the inner loop was compiled using a standard C compiler. Compiling from source meant that compilation time was sometimes as much as 50% of the overall execution time, but VSO was still typically profitable. The speedups in Przybylski's simulator were higher, in part, because it had extra machinery for simulating some timing-related components, such as write buffers; about 130 parameters were needed to specify a two-level cache system. The simulator in this dissertation is more like Dinero [Hill et al. 93]: it computes miss rates but not miss costs.

Decompression The pardoz decompression algorithm is based on a simple transliteration of an input byte to an output string from a table that is fixed over the dataset. It was designed to provide modest compression ratios and high decompression performance for files that were decompressed frequently. The VSO'd version was derived from an earlier static-code implementation.

Mike Burrows provided a suprising static assembly code inner loop for the DEC Alpha AXP: the inner loop is unrolled by 8, three-iteration software pipelined and dual-issue scheduled [Burrows 94]. Unfortunately, the loop cannot be derived in a straightforward way from C source code, cannot be extended directly to other architectures (it relies on 64-bit words to perform output buffering in a register and uses AXP "byte insert" instructions unavailable on other architectures), and does not handle all of the pardoz cases. However, the performance is theoretically close to the VSO'd code and has the advantage that the inner loop is small enough to fit in the cache, whereas the VSO'd inner loop is larger than the AXP's 8KB instruction cache.

The results for pardoz are not applicable to the familiar Ziv-Lempel algorithm [Ziv & Lempel 78], which has at least two overheads that are not eliminated with VSO. First, Ziv-Lempel typically maps variable-width input codes; the bit-manipulation overhead is potentially a bottleneck to the overall inner loop and is not eliminated with VSO. Even where the width is fixed, it is still not typically an even number of bytes [Bunton & Borriello 90, Bunton & Borriello 92] and manipulation is thus more expensive. Second, the codes index a dictionary that is updated incrementally by each input value. The benefit is that the table can change adaptively, based on local properties of the data, and thus improve compression ratios. Unfortunately, the same incremental table update means that transliterating the table into code is not straightforward with a conventional code generator; even with a carefully-tuned incremental code generator, generation costs might overwhelm the performance benefit.

Debugger The conditional breakpoint details of the debugger in Part I is based almost entirely on the Free Software Foundation's gdb [Stallman 87] debugger, with minor modification to generate code for breakpoints and to download read-only code and data into the debuggee using ptrace. The code generator was implemented by translating gdb breakpoint interpreter code into machine code or into interpreter code for the interpreter linked with the debuggee. Some minor modifications were used to make a return from fast breakpoint code appear as if they had instead come from a conventional breakpoint trap ("Debugger Breakpoints", pg. 120). The conditional breakpoint details are similar to conditional breakpoints in Kessler's Shepard [Kessler 90a]. The static-code version is implemented by downloading read-only data into the debuggee and using an interpreter in a debugging library linked with the debuggee, an idea also described by Paxson [Paxson 90b]. Using static code in the debuggee allows the static code version to avoid domain crossing overheads, whereas previous fast breakpoint implementations compared against static code that was in the debugger.

An alternative is to instrument the program binary statically [Copperman & Thomas 95]; however, static instrumentation must rewrite *all* memory reference operations and thus both slows program execution and also cannot be used to debug some kinds of instruction space modification.

Sorting Mainframe sorts have used RTCG for decades [Levine 89] but the details appear to be guarded, or at least not widely published. Mainframe sort generators perform additional optimizations beyond those described in Part I, such as integrating the predicate into the main sorting loop. An earlier implementation [Keppel et al. 91] of the sorting routine in Part I used the Free Software Foundation's C compiler, gcc [Stallman 93] to generate templates in much the same waye as the extraction phase described by Consel and Noel [Consel & Noel 96]. However, since the code generator was being "tricked" into producing templates, the resulting code was not reliable across all architectures. A subsequent static-code implementation was written to use threaded code, but at the time gcc's support was unreliable, though it has since reached production quality [Bedichek 95b]. The sorting routine used in this dissertation has been used in an automated system on a DEC Alpha, with a breakeven point of 3,000 records and an asymptotic speedup of 1.2 over static C code. VSO was used with an insertion sort in order to generate a new predicate for each item inserted.

Even with fast compilation, however, VSO was not profitable: most comparisons compared only a few characters, reducing speedups, and recompilations were frequent [Leone & Lee 96b].

Chapter 11

Instruction Cache Coherency

"Architectures that allow writes into the instruction space (self-modifying code) ... can cause trouble for pipelining (as well as for cache designs). For example, if an instruction in the pipeline can modify another instruction, we must constantly check if the address being written to by an instruction corresponds to the address of an instruction further on in the pipeline. If so, the pipeline must be flushed or the instruction in the pipeline somehow updated."

–[Hennessy & Patterson 96], page 200.

The villainy you teach me I will execute,
and it shall go hard but I will better the instruction.

–William Shakespere, *The Merchant of Venice*

11.1 Introduction

Whenever the instruction space is modified, it is necessary to perform some kind of coherency in order to ensure correct operation. Although many machines provide coherency mechanisms, the interfaces vary greatly from platform to platform. In addition, the performance of instruction cache coherency may depend on various low-level optimizations. This chapter discusses various machine-level instruction caching issues and describes a portable interface for software-managed instruction cache coherency [Keppel 91].

On older machines, instructions were changed simply by writing to the right memory location. However, processor architectures have been optimized for programs in which the instruction space does not change during program execution; on these machines, software must signal coherency explicitly. Although most systems provide mechanisms for signaling changes to the instruction space, there are a wide variety of details about how and when coherency is performed. For example, some systems perform coherency by flushing part or all of the instruction cache; in some of these, the data cache must be flushed as well. Other systems allow particular pages to be marked so that the instruction cache automatically watches or *snoops* for changes; in these systems, coherency is signaled when the storage is allocated for instructions, *before* the instruction writes actually occur [Bedichek 89].

It is such variety in signaling changes to the instruction space that causes problems. Other machine-dependent operations, such as compiling and linking have been abstracted, encapsulated, and provided via a machine-independent interface, but instruction space modification has not. Thus, for each new platform – that is, each combination of hardware and operating system coherency primitives – every program with a dynamic instruction space must be reexamined and modified to use the coherency primitives for that platform. In essence, the programmer is required to write in the "assembly language" of instruction space coherence operations.

Applications modify the instruction space in a wide variety of ways and for a wide variety of reasons (see Chapter 10). Thus, a portable interface must operate in a wide variety of situations and yet provide efficiency close to that of the nonportable cache coherency operations provided by the underlying machine.

Instruction space modification issues have been dealt with for specific applications [Deutsch 90, Foderaro 90, Massalin 90], but interfaces vary widely between processors and operating systems. Meanwhile, there is significant anecdotal evidence that there are problems both with performing instruction space updates correctly and with achieving good performance for programs that update the instruction space frequently (for instance, by fine-grained incremental compilers) or where large regions are modified (e.g., in garbage collecting) [Johnston 79, Koopman 89, Deutsch 90, Foderaro 90, Kessler 90a, Massalin 90].

This chapter examines architectural features relating to instruction space update and introduces a standard interface for modifying program instruction space. The interface hides machine and operating system specific characteristics, and provides a portable way to move executable code that resides in data space into instruction space. The interface thus provides the programmer with a single, portable abstraction that achieves performance close to the raw machine cost across a variety of platforms.

11.2 Implementation Issues

The two key issues for instruction space modification are protection and coherence. This section describes important protection and coherence features of several common platforms.

11.2.1 Protection

An address space is typically broken into one or more code segments and one or more data segments. Some platforms allow segments to be read, written, and executed arbitrarily, while other platforms restrict their use. For example:

- Many microcomputers allow code segment(s) to be freely modified. Writable code segments are increasingly rare, since protecting code from spurious writes makes debugging easier and writable code cannot be shared among independent processes (§10.6).

- Most minicomputers and workstations prohibit modification to code segments, but do allow data segments to be executed.

- Some systems disallow both writable code segments and executable data segments. An extension that allows mutable code must satisfy any system-level protection and security rules (§10.5.15).

- Some hardware uses separate memories for code and data, so copying is required.

11.2.2 Coherency

Processors cache instructions in various structures, including prefetch buffers. Between the last modification of a code fragment and the first invocation of that fragment, some coherency action must be performed, so that the new instructions and not the old, cached instructions are executed. There are two coherency strategies: *invalidation*, which deletes cached copies, and *updating*, which modifies the cached values so that they are coherent. Invalidation is more common.

For each structure that performs caching, there are various ways to implement explicit coherency:

- If there is no cache or there is hardware coherency (snooping), then no explicit coherence is required, because coherence is provided transparently by the hardware.

- Some instruction caches perform hardware coherency on pages marked specially by the kernel. For the Motorola 88200 caches provide coherency on a page-by-page basis Coherency is provided on a page-by-page basis [Mot 89b, Bedichek 89].

- On some systems, explicit coherency can be done by user-mode code, for example on the VAX [Leonard 87] and the SPARC [Sun 91].[1] Other systems perform coherency at supervisor privilege, so the cost includes both the coherency action itself and the cost of going to a supervisor (kernel) mode. For example, the MIPS R2000 [Kane 87] and Intel i860 [Int 90] both perform flushing in supervisor mode (see also §10.5.10).

- Some systems, especially those with small caches, perform coherency by flushing the entire cache [Sites 92, DEC 92]. Other systems perform coherency at the granularity of cache blocks or cache lines (typically a small number of bytes), pages (typically 1KB to 8KB) or the whole cache (potentially 64KB or more). For example, the Motorola 88000 caches can perform software-controlled flushes of a single line, a page, or of a whole cache [Mot 89b, Bedichek 89].

11.2.3 Multiprocessor Issues

For most shared-memory multiprocessors, hardware ensures that each write to a data segment is held coherent across all processors. However, executing code segment writes and instruction cache updates performed on one processor do not necessarily cause coherency with caching structures on other processors. Therefore, it may be necessary to perform an expensive interprocessor interrupt and update all the other caches or perform an interprocessor translation lookaside buffer update

[1] The SPARC hardware primitive that performs instruction space coherency is an instruction that makes two words coherent [Sun 91]. If there is an off-chip instruction cache, the instruction may cause a reserved operand fault that is handled by kernel software in order to make the off-chip cache coherent. Thus, a naive attempt to make 100 words of memory coherent may cause 50 kernel traps!

("shootdown" [Black et al. 89]) in order to force each processor to trap on the first new reference to a page [SGI 94, Mashey 95]. Note that most systems do make the instruction space coherent when a single process migrates [SGI 94, Mashey 95], but changes to, e.g., instructions in shared memory, are not necessarily reflected on other processors that execute out of that shared memory.

11.3 A Virtual Machine

Because various platforms provide such a variety of coherency primitives, a portable interface must capture machine-dependent information in order to provide efficient implementation. This section describes a virtual machine that is used as the basis for the portable interface. In effect, the virtual machine is the "worst possible" machine and requires the union of all real machine coherency operations. Later sections show various ways that the interface can be mapped to real machines.

11.3.1 Machine Model

The virtual machine features are a set of real machine features, chosen to allow optimizations across a variety of target machines. The virtual machine has separate instruction and data spaces and separate instruction and data caches; the data cache must be written to memory before the instruction cache can be updated. The instruction cache does not snoop, but can be updated or invalidated on varying grain sizes. The cheapest coherency implementation depends on the number of bytes of new code. Cache coherency may be performed only at kernel privilege. The virtual machine makes data executable in two ways: sometimes it copies from data space to instruction space, and sometimes it maps pages from data space to instruction space. In any event, the data-space representation of the code cannot be written while the instruction-space version is being executed.

For the virtual machine, several steps are necessary to make a fragment executable:

- Allocate a data space area for the code, called a *data region*.
- Load machine code into the data region.
- Allocate a "shadow" instruction space area for the executable code, called an *instruction region*.
- Copy from the data region to the instruction region.
- Write back the data caches and update the instruction cache.
- Execute from the instruction region.

When the code is no longer being executed, the regions can be deallocated or reused. If the regions are being reused, then the data region must be made writable before proceeding with writing, copying, cache updates, etc.

11.3.2 Procedural Description

This section describes a procedural version of the interface (called *fly*, short for "on-the-fly"). The interface allows dynamic allocation of functions. The procedural interface is:

```
fly_alloc (size) return handle
fly_daddr (handle) return d_region
fly_iaddr (handle) return i_region
fly_copy (handle, offset, size)
fly_use (handle, offset, size)
fly_modify (handle)
fly_free (handle)
fly_hint (handle, hint)
```

The data and instruction regions are allocated together with fly_alloc, which takes the number of memory units to allocate and returns a *handle* used with other *fly* operations.[2] The address of the data region is returned by fly_daddr, which takes a handle as an argument. The data region is aligned so that it can be read and written as an array of characters, integers, structures, etc. (The alignment requirements are the same as those for the C library malloc dynamic memory allocator.) The instruction region address is similarly returned by fly_iaddr.

All changes to a data region must be explicitly propagated to the shadow instruction region using fly_copy, which takes as arguments a handle, an offset at which the changes start, and the length of the changes. If a region is reused and only a small part of it is changed, then only the changes must be copied to the instruction region. Failure to propagate changes causes implementation-defined results; for example, the instruction region may be executed with or without the changes, or there may be a runtime error.

Once the instruction region has been updated using fly_copy, it is then made executable using fly_use. An offset and length describe the part of the region that might be executed, or which has been modified since the last call to fly_use: for example, if 10KB is allocated but then only 1KB is used; or if 10KB is being used but only 1KB has been updated since the last call to fly_use. After fly_use has been called, the region may not be read or written until fly_modify (described below) has been called.

Storage allocated with fly_alloc is freed using fly_free. Alternatively, fly_modify makes the data region writable, so that it can be manipulated as before. Executing the instruction region after fly_modify (or fly_free) has been called has implementation-defined effect.

The hint argument to fly_hint gives hints about how the region might be used and allows the runtime system to make optimizations. An argument of FLY_MOVEOK says that the runtime system is free to allocate a new instruction region and blindly copy the code, if copying is cheaper than updating the old instruction region. A hint is needed here, because each call to fly_modify might use a new instruction region, and so the address of the instruction region must be rediscovered using

[2]The interface is actually "fly_alloc (size, d_region, i_region) and "fly_free (handle, bool dregion, bool iregion)", which allows the user to ask for particular addresses for the data or instruction regions. The remainder of the chapter uses "fly_alloc (size)" and "fly_free (handle)" as shorthand for calls when both regions are NULL, which means that the *fly* system is free to choose addresses.

fly_iaddr after each call to fly_modify.[3]

11.4 Generality Of The Interface

This section discusses three issues related to the generality of the interface: garbage collection, typing, and self-modification.

Systems that do garbage collection of memory have no explicit freeing, so the programmer should not be held responsible for calling fly_free. Also, a garbage collector can do implicit updates of the instruction space (for instance, code objects can be moved by a copying garbage collector), so some knowledge of the instruction space must be built into the garbage collector. Otherwise, the basic issues are the same for garbage collection as they are for explicit allocation and freeing.

The interface treats code as an untyped array of bytes. For strongly-typed languages, this model is insufficient and a stronger notion of typing is needed. For example, PASCAL has no mechanism for turning arbitrary arrays into functions, so the notion of type must be extended to include code. Likewise, for PASCAL there is no way to define a routine such as fly_iaddr that returns pointers to routines of various types.

The most obvious prohibition of this interface is that a region cannot be simultaneously changed and executed. Instead, the region must be "stopped" by making sure that no program counter points into the region. Thus, the execution of the instruction space representation of a region cannot have the side effect of changing the data space representation. The effect of self-modification must be simulated by jumping out of the region to code that performs the modification and then returning to the updated region. On multiprocessors, when one thread of control modifies a region, it must ensure that there are no other threads of control executing in the region.

11.5 Examples

Figure 11.1 shows example code that demonstrates allocation, copying, and use. In the example, a single function is read from file. The file is file_size bytes long and holds one function starting func_offset bytes into the file. The function is dynamically linked into the running program and a pointer to the function is returned.

Figure 11.2 shows an example that demonstrates changing and freeing regions. A function taking no arguments and returning an integer is run N times. Each time the function is run, an integer embedded in the function is set to the iteration number. The embedded integer is embed units from the beginning of the function and is suitably aligned so that writing an integer to that location will not cause a misaligned reference fault. The function starts at the location pointed to by the variable executable and is exe_size units long. The function is written in position-independent code, so no relinking is needed.

[3]The original interface [Keppel 91] chose new instruction regions after each call to fly_use and thus FLY_MOVEOK worked only with position-independent code. The current interface works with position-dependent code but may require that position dependencies are updated after each call to fly_modify. The interface change is backwards-compatible.

```
    funcptr
load_function (objfile file, int file_size, int func_offset)
{
    flyhandle handle
    d_region *daddr    /* Address of D-space region. */
    i_region *iaddr    /* Address of I-space region. */

    /* Allocate space and read object code. */
    handle = fly_alloc (file_size)
    daddr = fly_daddr (handle)
    read (daddr, file_size, file)

    /* Relink the code at 'daddr' to run at address 'iaddr'. */
    iaddr = fly_iaddr (code)
    dylink (daddr, iaddr)

    /* Move code from 'daddr' to 'iaddr'; update the */
    /* instruction space and get a pointer to the function. */
    fly_copy (handle, func_offset, file_size-func_offset)
    fly_use (handle, func_offset, file_size-func_offset)
    return (&iaddr[func_offset])
}
```

Figure 11.1: Allocating and linking a function.

The operator & returns the memory address of its operand.

11.6 Implementation Strategies

The two basic implementation issues are protection and coherence.

A given implementation might also provide error detection. For example, on a machine with hardware coherence and pages that are both writable and executable, a "safe" implementation can nonetheless have fly_alloc allocate separate instruction and data regions and have fly_copy copy from the data region to the instruction region. Other operations can check to see that the two versions are coherent – that all changes to the data region were in fact propagated to the instruction region and that both regions remained unchanged during execution. In contrast, an optimized implementation allocates only one region and fly_copy and fly_use do nothing. The optimized version performs correctly, but does less checking to ensure that it is used correctly.

The remainder of this section discusses implementation tradeoffs for an optimized implementation on various classes of machines. This section ignores error detection and reporting, assume that there is already a general-purpose memory allocator, and that, if necessary, a system call page_protect can be used to change the default (read, write, execute) page protections. This section implements the interface in terms of various coherence primitives, which are described as they are used. Cost analysis assumes the compiler does function inlining, so that trivial functions (such as empty functions or functions that are simply a call to another function) incur no overhead.

The various implementations are described as differences relative to the implementation for a

```
    void
rerun (void *executable, int exe_size, int embed)
{
    flyhandle handle
    d_region *addr
    i_region *iaddr
    func f              /* Final function pointer. */
    int i, result

    /* Allocate space for position-independent code. */
    handle = fly_alloc (exe_size)
    fly_hint (handle, FLY_MOVEOK)
    daddr = fly_daddr (handle)

    /* Copy whole function first to data space, then instruction space. */
    copy (executable, &daddr[0], exe_size)
    fly_copy (handle, 0, exe_size)

    for i=0 .. N {
        /* Copy 'i' to the data space version then to the instruction space version. */
        daddr[embed] = i
        fly_copy (handle, embed, sizeof(i))

        /* Make region executable. Since FLY_MOVEOK, 'iaddr' may change. */
        fly_use (handle, 0, exe_size)
        iaddr = fly_iaddr (handle)

        /* Execute the updated function. */
        f = (func)&iaddr[0]
        result = f( )
        consume (result)

        /* Done executing, make writable again. */
        fly_modify (handle)
    }

    /* Free both the data and instruction regions. */
    fly_free (handle)
}
```

Figure 11.2: Incrementally updating and executing a function.
The notation (type)y means to convert the value of y to type type before using it.

simple machine.

Simple If data is executable and if there is hardware coherence or no caching, then the implementation is simple. The fly_alloc routine is just a call to the general-purpose allocator and the return handle is the address returned by the allocator; fly_free is call to the deallocator. The fly_iaddr and fly_daddr routines simply return their argument (the handle), and fly_copy, fly_use, fly_modify, and fly_hint are all empty; they ignore their arguments and do nothing.

11.6.1 Allocation

The implementation is optimized in various ways to match the code and data allocation strategies of the target platform. It may be optimized around the memory architecture. The optimizations serve either to reduce the cost of updating a region or to reduce the interference with other regions.

Harvard A "Harvard" architecture is one where the data and instruction regions cannot use the same memory. On such machines, the call to fly_alloc must allocate both a data region and a code region, and the call to fly_copy must perform an actual copy. The call to fly_free frees both regions. Note that there may also be coherency issues, which are discussed below.

Virtual Harvard A "Virtual Harvard" architecture is one in which the default configuration appears to be a Harvard architecture – code is not writable and data is not executable – but in which data pages can be made executable by, for example, a call to page_protect. If a page can be simultaneously writable and executable, the the interface should be implemented using the strategies described below. If, however, a page cannot be both writable and executable, then calls to fly_use and fly_modify toggle a region between writable and executable. Since one region may be writable when an adjacent region is executable, regions returned by fly_alloc cannot share pages.

Page-Based Allocation On some platforms, different regions should or must be allocated to use separate pages. For example, if a page cannot be simultaneously writable and executable, then allocations must use disjoint pages. Where the instruction cache snoops, there may be coherency overhead for writes to pages that are (partly) cached in the instruction cache. Sharing overhead can be avoided by allocating pages using an allocator that minimizes the chance that any given page is used simultaneously for both code and data.[4]

Line-Based Allocation If instruction cache coherency is maintained on a per-line basis, then its overhead can be eliminated by ensuring that there is a line-sized gap between items that are allocated for data and those that are allocated for code. Alternatively, an allocator like that for page-based allocation can be used.

11.6.2 Coherency

There are two major coherency issues: the privilege needed for coherency operations and the granularity of coherency. On some systems, coherency operations require kernel support, while other machines provide user-mode coherency operations (§11.2.2. The granularity of coherency covers a wide range, from a few bytes in a cache line to many kilobytes in a large cache. Thus, the coherency cost is typically composed of a size-independent part and a size-dependent part. For example, coherency might have a size-independent overhead of a system call plus a certain cost for

[4]Typically, the implementor of the fly interface will be responsible for implementing the alternate allocator.

each line invalidated from the cache. This section discusses platform-specific coherency primitives and suggests uniprocessor implementations of the portable interface.

Global Coherence A platform-specific primitive that makes the whole instruction space coherent can be used when flushing the instruction cache is cheap (in both the up-front cost and the side-effects). Unless the instruction cache reads from the data cache, data cache writebacks may also be needed.

Small-Grain Coherence Coherency should be done on a smaller grain if flushing the whole cache is expensive (e.g., the instruction cache must be flushed entry-by-entry). Each changed region or fragment is made coherent, but other regions may be left incoherent.

Vectored Coherence If the coherency primitive takes a vector of addresses and sizes, then each call to fly_copy can put the modified address and size into a vector and defer the coherency call until fly_use is called. When updating requires kernel privilege (or in general, when the size-independent part of the updating cost is high), this optimization can reduce the number of system calls if several parts of a region will be modified for each call to fly_use. Since the vector can fill up with repeated calls to fly_copy, coherency is also performed whenever the vector gets full.

Region Merging Even if the coherency primitive does not take a vector of addresses, two adjacent regions that are "close" or overlap can be coalesced into one region when needlessly updating a few extra locations is cheaper than performing multiple system calls. The value of "close" is the breakeven point at which it is cheaper to needlessly update some number of locations than it is to perform another system call. For example, each call to fly_copy can add to a vector of addresses and sizes as above. When the vector is filled or when fly_use is called, regions can be merged. Where the size-independent cost is high, simply updating the whole region (or as much of it as might be executed) may be best.

Threshold Global Flushing Invalidations that "cover" a large fraction of the cache should invalidate the *whole* instruction cache. Two optimizations are possible at this point. First, no more coherence operations are needed on the region until after it is next invoked. Second, no more coherence operations are required on *any* region until some region is made executable with fly_use. Thus, the implementation can set both a flag in the region and a global flag. Further calls to fly_copy check the global flag. If the global flag is set, the region flag is set and no cache updating is performed. If the global flag is clear but the region flag is set, then no cache updating is performed. The global flag is cleared by any call to fly_use; the region flag is cleared by a call to fly_use on that region.

Lazy Coherence It may be cheapest to test a condition and omit coherence when it is not needed. For instance, if the cache is flushed on every context swap, a coherence operation is redundant if there was a context swap between the last call to fly_copy and the first use of the region. Thus, it may be profitable to have fly_use perform the coherence based on what is actually in the cache at the time fly_use is called, rather than on what was in the cache when fly_copy was called. Threshold global flushing is a variation on lazy coherence that is relatively platform-independent.

Copying Coherence When coherence is expensive it may be cheaper to copy code to an uncached region than to ensure coherence of a cached region. In general, this is only practical when the region being invalidated is small. Furthermore, it is only correct when code can be moved (either the region is filled with position-independent code, or position dependencies are updated for each copy; see §11.3.2) The implementation can recognize candidates for copying coherence when the initial allocation is small and the FLY_MOVEOK hint is given.

Disabled Caching If caching of particular pages can be turned off then execution will be slower but the cost of explicit coherence can be saved (but note that sometimes data caching of those pages must be turned off as well, causing modifications to go more slowly). When a region is changed very frequently, explicit coherence may be a large part of the total cost of execution. Suppose that executing an instruction from cache has cost C, and that uncached execution costs U. If I instructions are executed N times, the uncached cost is:

$$uncached \quad = \quad U \times I \times N$$

When instructions are executed from the cache, there is a coherence cost Coh (which may depend on I), the first execution has uncached cost, and each further execution has cached cost.

$$cached \quad = \quad Coh \; + \; U \times I \; + \; C \times I \times (N-1)$$

The number of executions N at which the cost is the same either way is the *breakeven point* and is roughly:

$$N = \frac{Coh}{I \times (U - C)} + 1 \tag{11.1}$$

Executing fewer times than the breakeven is cheaper with disabled caching, while executing more times than the breakeven is cheaper with caching and coherence overhead.

11.7 Implementations

The *fly* interface has been implemented on the Sequent Symmetry, which is an Intel 80386-based shared-memory multiprocessor [Seq 87, Int 86]; the Alpha AXP-based DEC 3000 [Dutton et al. 92, Sites 92]; the MIPS-based DECstation 3100 and DECstation 5000/200 [DEC 90a, DEC 90b, Kane 87]; several SPARC-based [Sun 91] Sun workstations; and the DEC VAX architecture [Leonard 87] including the 2000, 3500, and 8550 models.

11.7.1 80386-based Sequent Symmetry

On the i386, a non-snooping instruction prefetch buffer holds no more than 16 instructions and is flushed on branches.[5] When the instruction space is modified, the prefetch buffer must be flushed if it holds any of the modified instructions. Thus, coherency is performed either by (a) executing 16 instructions or (b) executing a jump instruction. The semantics of the *fly* interface require a control transfer between code modification and code use, so no explicit coherence is required. The simple implementation even works for multiprocessor applications, because the model requires that *all* processors have to have stopped executing a region before the region can be modified. The only way to execute in a region is to jump there, thus all processors will flush their prefetch before executing "new" code. The remaining coherency is handled by the Sequent hardware [Seq 87].

Many 80x86-based systems depend on hardware instruction coherency, so the *fly* implementation should work on many other 80x86-based platforms as well.

11.7.2 AXP Under OSF/1

The DEC AXP family provides cache coherency using the IMB PALcall [Sites 92]. On the DEC 3000/400, IMB has the effect of flushing the 8KB first-level instruction cache. Since the first-level data cache is write-through, and since the instruction cache is refilled from the unified second-level cache, no additional flushing or writebacks are required [Sites 92, DEC 92].

By default, data pages are not executable. Therefore, when fly_alloc allocates space, it uses mprotect to make the necessary pages executable. Pages with newly-allocated code space may also contain data; that data thus also becomes executable, and the pages remain executable even after being returned through fly_free.

Instead of using IMB, an alternative is to flush incoherent code by executing other code that is constructed to map to the same cache location, thus forcing replacement of the incoherent code [Deutsch 90, Massalin 90] (§10.5.4). Such "flushing by replacement" works with virtually-indexed caches that are either direct-mapped, as with the DEC 3000, or associative using a predictable replacement policy. For *fly*, cache flushing is implemented by creating a cache-sized "table of code" with one table entry per cache line.

The cost for IMB is constant, about 1μS, as shown in Table 11.1. The cost for flushing by replacement increases with the number of cache blocks that must be replaced. Fetching each line in the "table of code" causes a cache miss that takes about 90ns to service from the second-level cache and 225ns from main memory (see Figure 4.2). Thus, flushing by replacement should never be profitable for regions that must replace more than $1\mu\text{S}/90\text{ns} = 11$ lines, which is about 350 bytes of code. If the table is not cached, it must be fetched from main memory, which makes flushing by replacement profitable only for regions smaller than about 128 bytes.

Flushing by replacement is thus implemented as a hybrid scheme that uses global threshold flushing, with full flushing performed by IMB. Table 11.1 shows that the cost for small regions is about half the cost for IMB. However, the cost deciding which implementation to use roughly doubles

[5] The behavior of the prefetch buffer was determined by experimentation, not documentation.

Table 11.1: Coherency costs for IMB and a hybrid IMB/code table implementation.

	IMB		Hybrid		Ratio
n	Cost	Amortized	Cost	Amortized	
1	1.08μS	1,080ns	0.48μS	480ns	0.44
10	1.08μS	108ns	0.48μS	48ns	0.44
100	1.03μS	10ns	0.68μS	5ns	0.66
1,000	0.96μS	1ns	4.24μS	4ns	4.40
10,000	0.95μS	100ps	2.00μS	200ps	2.10
100,000	1.00μS	10ps	2.07μS	20ps	2.07
1,000,000	1.00μS	1ps	2.07μS	2ps	2.07

Times are reported as microseconds to invalidate n bytes and nanoseconds or picoseconds per byte flushed. *Ratio* is *Hybrid* divided by *IMB* and is less than 1.0 where the hybrid implementation is faster.

the cost for large regions. Table 11.1 shows performance with the threshold set at 1024 bytes, which makes flushing some region sizes nearly four times as expensive as using IMB.

There are at least four additional considerations in comparing the IMB and hybrid schemes. First, when a small region is incoherent but the whole cache is flushed, valid instructions may be flushed, leading to overheads when they are later re-fetched.[6] For the DEC 3000, the added overhead may be 90ns per line times up to 255 lines, or nearly 23μS. Selective flushing can reduce the re-fetch costs. In practice, then, the hybrid scheme's practical threshold may be higher than its worst-case threshold.

Second, the cost of making an instruction coherent should be balanced against the cost of generating an instruction. For example, simply copying a byte on the DEC 3000 costs at least 1ns.[7] Thus, although hybrid flushing of a 10KB region is twice the cost of IMB, the doubling adds at most 10% to to the overall cost. In contrast, the cost of patching a four-byte instruction may be less than 10ns, so the hybrid scheme may halve the *total* cost of some small-region operations.

Third, the hybrid implementation is larger. First, the IMB uses a single instruction and may be inlined and thus be smaller than the procedure call needed to invoke the table of code. In addition, the "table of code" must generally be as large as the cache, 8KB on the DEC 3000. The hybrid scheme may thus bloat the application and hurt TLB and paging performance.

Finally, the table of code must be aligned on a 64-byte boundary. The current programming tools support alignment to 8-byte boundaries but not to 64-byte boundaries.

11.7.3 MIPS-based DECstations

The DECstation 3100 and 5000/200 run RISC Ultrix and are implemented using the MIPS R2000 (16MHz) and R3000 (25MHz) processors. Both machines have 64KB write-through data caches

[6]The situation is analogous to false sharing with long lines.
[7]Page mapping tricks may yield lower copying costs.

Table 11.2: Seconds to execute 10,000,000 empty procedures.

Host	Cached	Uncached	Dilation
DECstation 3100	4.2	26.9	6.4
DECstation 5000	2.8	34.9	12.5

and 64KB non-snooping instruction caches [DEC 90a, DEC 90b, DEC 92]. The instruction cache is updated by invalidation, can be partly or completely invalidated, and can be invalidated only in supervisor mode. Thus, a part of optimization is to focus on avoiding unnecessary system calls.

Pages do not need to be cached, which leads to two implementation alternatives: (a) put modifiable instructions on non-cacheable pages or (b) cache the writable instruction pages and then perform explicit cache flushing when instructions change. If the instructions in a region are changed frequently, then putting the instructions in non-cached pages eliminates the cache flushing overhead. However, as Table 11.2 shows, executing from main memory is slower than executing from cache, so the execution time is *dilated*.

Table 11.3, indicates that a basic system call under Ultrix 4.0 takes 16-58ms depending on the underlying hardware. The cacheflush system call time is otherwise roughly proportional to the size of the region being flushed, about 0.6-1.5 nanoseconds per byte flushed, independent of the cache size.[8] For example, the cost per byte for large regions, where the trap overhead is small, is:

$$cost = \frac{(570\mu s \; - \; 21\mu s)}{1,000,000 - 10,000} \approx 0.55ns$$

The peak cached instruction speed of the 3100 is roughly 60 nanoseconds per instruction. To modify a single 4-byte instruction on the 3100, the breakeven point, from Equation 11.1 is at about

$$\frac{58,000ns \; + \; 1 \times (4 \times 1.4ns)}{1 \times (6.4 \times 60ns \; - \; 60ns)} \; + \; 1 \approx 180 \; executions$$

So the instruction should be cached only if it will be executed more than about 180 times for each change. For 100 instructions invalidated at once, however, the breakeven for the region is about 2.8 executions.

Similar extrapolation indicates that the 5000/200 costs about 16 microseconds per cacheflush system call and about 0.55 nanoseconds per byte flushed. At about 40 nanoseconds per instruction, the breakeven for a single instruction is about 36 executions, while the breakeven for 100 instructions is about 1.4 executions.

The preceding computations assume that an entire region is changed all at once. If only parts of a region are changed, then flushing the whole region will flush coherent instructions and the cost of re-fetching those coherent instructions must be included in the coherency cost. If the cache flush primitive is called only on the changed parts of a region, then no coherent instructions will be flushed, but the cost of coherency will include multiple system calls.

[8] Table 11.3 shows that the cache flush system call has been tuned between versions of the operating system. Our analysis uses measurements from the newer system. The performance of the old version is included for comparison.

Table 11.3: Microseconds to invalidate n bytes, and nanoseconds per byte flushed.

n	DS 3100 OS 3.1	DS 3100 OS 4.0	DS 5000 OS 4.0
1	53	58	16
10	53	60	16
100	56	58	16
1,000	77	58	16
10,000	316	70	21
100,000	2,900	200	71
1,000,000	28,000	1,500	570
time/byte	27 ns/B	1.44 ns/B	0.55 ns/B

Usually, it is slightly cheaper to perform coherency on small regions using a single system call, while small changes to large regions are dramatically cheaper when only the incoherent instructions are flushed, even at the expense of multiple system calls. Therefore, *fly* is implemented to perform flushing when fly_copy is called. A third alternative would be to do region merging. For the experiments of this dissertation, the difference in performance of the implementations has not been significant.

11.7.4 SPARC

The SPARC-based Sun workstation described in Chapter 4.3 used unified data and instruction, so the implementation was simple.

The *fly* interface was also implemented so that coherence was performed by fly_use, which executed the necessary flush instructions [Sun 91]. The SPARC Applications Binary Interface (ABI) defines the flush instruction to work across all implementations, so the split-cache *fly* implementation should work for non-Sun SPARC-based platforms as well.

11.7.5 VAX

The VAX family of computers cache instructions in non-snooping hardware structures (caches and/or instruction prefetch queues). The structures are flushed by executing an REI (return from interrupt) instruction from user mode [Leonard 87]. Flushing is otherwise difficult because the caching structures are of implementation-defined size and organization and are not necessarily flushed either by jumps or by executing a fixed number of instructions. Thus, fly_use is implemented using REI, inlined where assembly code inlining is supported [Stallman 93]. Table 11.4 shows the cost per invalidation for 10,000,000 invalidations on several members of the VAX family.

The REI instruction flushes both coherent and incoherent instructions, leading to re-fetch overhead. The larger the cache, the larger the potential degradation. Some implementations can flush the cache more cheaply than an REI by executing a particular sequence of "normal" instructions. Experimentally, the VAX 2000 cache can be flushed by executing 8 nop instructions; the VAX 8550

Table 11.4: Cost per invalidation

Host	Microseconds	Ratio to no-op
VAX 2000	10.2	47
VAX 3500	3.1	33
VAX 8550	1.0	22

can execute thousands of instructions without implicitly updating the cache. Thus, faster hand-coded implementations are possible, but faster *fly* implementations are possible, too. In both cases, using alternative implementations would require precise machine definitions.

11.8 Restrictions

The *fly* interface is intended to be general-purpose in the sense that it can be used to make an arbitrary sequence of instructions executable, and in that it is expected to work on an arbitrary machine. As important as what it *is* designed to do is what it *is not* designed to do. Several potentially useful features have been left out to make the implementation as simple and efficient as possible.

- The portion of a program that is designated by the system as read-only (that is, the statically-compiled program text) cannot be changed using this interface. In order to extend the system to do so, it would (for many systems) be necessary to mark the modifiable pages as copy-on-write, and to change the backing store used for paging the changed portion of the code.
- The interface is not designed to allow static allocation of code space. Static memory allocation is difficult because of varying platform-dependent coherence and protection requirements.
- There is no way to grow an allocated region and there is no mechanism for freeing the data region without freeing the code region. These features would be easy to program, but were left out to keep the interface simple.
- A code fragment must not be discarded or reused while it is being executed. However, the interface provides no mechanism for deciding whether a region is in use, because doing so is sometimes unnecessary, and is otherwise hard. If the region obeys LIFO calling, then the caller "knows" when the code is no longer being used and no other mechanism is needed. If call is non-LIFO, then figuring out whether a region is in use is hard to do either cheaply or generally. Consider, for instance, a signal handler, which can be invoked spontaneously. Even if the handler prologue and epilogue keep reference counts, the counts will be wrong for the first and last few instructions of the handler [Kessler 90].
- The interface does not allow individual regions to be self-modifying, because coherence becomes very complicated. Indeed, some systems take explicit measures to allow self-modification or to

allow simultaneous execution and update, yet are still subject to race conditions [Kessler 90].

Chapter 12

Conclusions

“We are surrounded by insurmountable opportunity.”

–Pogo

This dissertation studies runtime code generation (RTCG), which is the general area of introducing new code into a running program. Runtime code generation is used various ways: to improve performance, portability, flexibility and so on.

Part I of this dissertation focuses on *runtime-compiled value-specific optimization* (*VSO*), which is using RTCG to improve the performance of statically-typed programs by compiling around the runtime values of key variables. Most compiler optimizations have focused on moving work from runtime to static compile time; it is thus perhaps counter-intuitive that moving operations from static compile time to runtime will improve performance. However, the important observation is that extensive static specialization is limited because it causes unacceptable code growth. In contrast, a runtime optimizer uses runtime information to generate highly specialized code on demand. VSO can thus avoiding the static compiler's space explosion, which enables it to produce better code than that of the static compiler.

Part I of this dissertation shows that it is profitable to optimize around the current values of some variables, with application speedups of up to 2.5 over statically-optimized code. However, where static optimizations are limited by space, dynamic optimizations are limited by time. Thus, one key to successful optimization is minimizing the compile time. Part I of this dissertation investigates how value-based optimization may be performed at runtime using an *application-specific* runtime compiler. Specializing the compiler around the needs of the application eliminates unused generality and thus makes the compiler faster, in turn improving the profitability of the runtime-compiled optimizations.

Part II studies the more general area of using RTCG to improve general metrics including performance, flexibility, ease of debugging and maintenance and portability. Part II studies general questions such as “what is code?” in order to provide a basis for discussing code modification. It then investigates a large number of RTCG issues and techniques and investigates instruction cache coherency, which is common to most systems performing RTCG of native machine code.

Specific contributions of Part I include:

- *Feasibility:* This dissertation presents a feasibility study of VSO using several hardware platforms based on contemporary RISC processors. The results show that VSO leads to overall speedups, measured as high as 2.5 over highly-tuned static code, thus demonstrating that VSO exposes optimizations that fundamentally cannot be performed statically (Chapter 5).

- *Code Generation Cost:* This dissertation experimentally validates the model in Figure 1.1 and studies the significance of code generator speed. The results show that some applications can take advantage of today's general-purpose retargetable code generators, but that other applications need faster code generators to be profitable across a wide range of inputs (Chapter 5).

- *Runtime Optimization:* This dissertation experimentally studies the effects of optimization level on overall performance. A perhaps surprising result is that, for VSO, increasing optimization levels rarely produce better code – that is, fast and simple code generation can produce fast-running code. The insight here is that VSO tends to replace variables with constants, and thus replace indefinite information with definite information. As a result, simple runtime constant-propagation optimizations are very effective. (Chapter 5). There are two important consequences. First, it defuses the complex problem of deciding which optimizations to use at runtime. Second, it suggests that very fast template compilers will also produce competitive code.

- *Architecture:* This dissertation analyzes the low-level behavior of VSO'd programs and shows that subtle architectural interactions rarely affect VSO's benefit. In most cases, the bottlenecks that limit VSO speedup are fixed overheads that are present in both the static and VSO'd versions, such as limited memory bandwidth and I/O costs. With VSO, the absolute costs of these bottlenecks are often unchanged, but are amortized over shorter runs and therefore are relatively more significant. A trace-driven study of program execution shows that each application makes different use of machine resources and thus that there is no single feature that makes an ideal "VSO architecture" (Chapter 6).

This dissertation also examines profitability analysis that would be performed at static compile time in order to assess whether to even consider a VSO implementation and, if so, under what runtime conditions it would be profitable. Despite limited static information and approximate performance models, the results show that static analysis can, in at least some cases, predict breakeven points that are close to those seen in practice (Chapter 7).

Specific contributions of Part II include:

- *Pedagogy:* The dissertation provides clarified definitions for basic notions such as "runtime", "code" and "modification" and thus helps provide a standardized terminology for explaining RTCG system behavior precisely (Chapter 9). The dissertation also explores a wide variety of code modification issues including: RTCG design and implementation issues and techniques; typical traps and pitfalls of RTCG systems; and references to a broad base of related work (Chapter 10). The study in turn serves as a framework for identifying, understanding and categorizing uses of RTCG.

- *History:* The dissertation gathers, in one place, a long description of interesting systems that have performed, used, or were related to RTCG (Chapter 10).

- *Instruction Caching:* Investigation and analysis of techniques for implementing instruction cache coherency efficiently across various hardware platforms, while using a portable interface (Chapter 11).

To summarize: runtime code generation has a long history. Although it presents several problems, it also presents opportunities for improving metrics such as code reuse, flexibility, performance, portability and reliability. This dissertation makes several pedagogical and technical contributions to the field. Part I of this dissertation provides a low-level explanation of *why* instruction-space modification presents these opportunities, and of how it is used to optimize around the values of variables; Part II of this dissertation provides a broad survey of RTCG, including definitions, issues and techniques.

Bibliography

[Abbott & Peterson 92] M. B. Abbott and L. L. Peterson. A Language-Based Approach to Protocol Implementation. Technical Report 92–2, University of Arizona, Department of Computer Science, 1992.

[Ado 85] *PostScript^{TM} Language Reference Manual*. Addison-Wesley Publishing Company, 1985. Adobe Systems, Incorporated.

[Aitchison et al. 92] M. Aitchison, I. Collier, and D. M. Engholm. Postings to USENET "alt.lang.asm" newsgroup, 23–30 January 1992. Discussion and trial implementations of a folklore programming challenge.

[Albaugh 95] M. Albaugh. Personal communication, September 1995.

[Amdahl 67] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *AFIPS Conference Proceedings; Proceedings of the Spring Joint Computer Conference*, (30):483–485, April 1967.

[Andrews & Sand 92] K. Andrews and D. Sand. Migrating a CISC Computer Family onto RISC via Object Code Translation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 213–222, October 1992.

[Appel 88] A. W. Appel. Re-opening Closures. Available as of August 1995 from "http://www.cs.princeton.edu/~appel/papers/reo.ps", January 1988.

[Appel 94] A. W. Appel. Axiomatic Bootstrapping: A Guide for Compiler Hackers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1699–1718, November 1994.

[Aral & Gertner 88] Z. Aral and I. Gertner. Non-Intrusive and Interactive Profiling in Parasight. In *Proceedings of Parallel Programming Experience with Applications in Languages and Systems (PPEALS)*, pages 21–30, 19-21 July 1988.

[ARD 94] ARDI. *Posting to USENET newsgroup* comp.emulators.misc*, describing the internals of the Executor simulator*, 3 November 1994.

[Athanas & Silverman 93] P. M. Athanas and H. F. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis. *IEEE Computer*, pages 11–18, March 1993.

[Atkinson & Morrison 85] M. P. Atkinson and R. Morrison. Procedures as Persistant Data Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):539–559, October 1985.

[AT&T 90a] AT&T. *System V Application Binary Interface*. Prentice-Hall, 1990.

[AT&T 90b] AT&T. *System V Application Binary Interface – SPARC Processor Supplement*. Prentice-Hall, 1990. See pages 5-1 to 5-11.

[Aus 90] The Austin Code Works Catalog. 11100 Leafwood Lane, Austin, Texas 78750 USA 512-258-0785, 15 November 1990.

[Auslander et al. 95] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, Effective Dynamic Compilation. Technical report, University of Washington, Department of Computer Science and Engineering, December 1995.

[Auslander et al. 96] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, Effective Dynamic Compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[Baker 91] H. G. Baker. Personal communication, July 1991.

[Baker 92a] H. G. Baker. Precise Instruction Scheduling wihtout a Precise Machine Model. *Computer Architecture News*, 19(6):4–8, December 1992.

[Baker 92b] H. G. Baker. Personal communication, October 1992.

[Baker 95] H. G. Baker. Personal communication, July 1995.

[Bank et al. 94] D. G. V. Bank, C. M. Shub, and R. W. Sebesta. A Unified Model of Pointwise Equivalence of Procedural Compuations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1842–1847, November 1994.

[Barzilai et al. 87] Z. Barzilai, J. L. Carter, B. K. Rosen, and J. D. Rutledge. HSS—A High-Speed Simulator. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):601–617, July 1987.

[Baumann 86] R. A. Baumann. Z80MU. *Byte*, pages 203–216, October 1986.

[Bedichek 89] R. Bedichek. Some Efficient Architecture Simulation Techniques. In *Proceedings of the 1989 USENIX Conference*, pages 53–63, 26 October 1989.

[Bedichek 92] R. Bedichek. A Multi-Space Operating System Debugger (unpublished), December 1992. Describes a user/kernel debugger for a Motorola 88000-based multicomputer; watchpoints are implemented in much the style of Shepard [Kessler 90a], with breakpoint expressions compiled to 88000 machine code and downloaded to the multicomputer and branches patched into the user or kernel code. Speedups were large, in part, because the "trap to debugger" time included communication over a comparatively slow inter-machine link.

[Bedichek 95a] R. Bedichek. Personal communication, 8 August 1995. Describes a user- and system-mode simulator that dynamically cross-compiles and instruments SPARCle target machine code to run on a SPARC host.

[Bedichek 95b] R. C. Bedichek. Talisman: Fast and Accurate Multicomputer Simulation. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 14–24, May 1995.

[Beeler et al. 72] M. Beeler, R. W. Gosper, and R. Schroeppel. *HAKMEM*. Artificial Intelligence Memo Number 239. Masschusetts Institute of Technology, Artificial Intelligence Laboratory, 29 February 1972. Available as of July 1995 from "ftp://ftp.netcom.com/pub/hb/hbaker/hakmem/hakmem.html".

[Bell & Newell 71] C. G. Bell and A. Newell, editors. *Computer Structures: Readings and Examples.* McGraw-Hill, Incorporated, 1971.

[Bell 73] J. R. Bell. Threaded Code. *Communications of the Association for Computing Machinery (CACM)*, 16(2):370–372, June 1973.

[Berg et al. 95] W. Berg, M. Cline, and M. Girou. Lessons Learned from the OS/400 OO Project. *Communications of the Association for Computing Machinery (CACM)*, 38(10):54–64, October 1995.

[Berlin & Weise 90] A. Berlin and D. Weise. Compiling Scientific Code Using Partial Evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[Berman et al. 95] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-Specific File Protection for the UNIX Operating System. In *Proceedings of the 1995 USENIX Conference*, pages 165–175, 16–20 January 1995.

[Bershad et al. 95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 1995 Symposium on Operating System Principles (SOSP)*, pages 267–284, November 1995.

[Bianco 93] D. Bianco. Personal communication, 1993.

[Bishop 87] M. Bishop. Profling Under Unix by Patching. *Software—Practice and Experience*, 17(10), October 1987.

[Black et al. 89] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill, and R. V. Baron. Translation Lookaside Buffer Consistency: A Software Approach. *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 113–122, 3-6 April 1989.

[Blair 93] M. R. Blair. Improving Scheme Program Performance through Heuristic Redundant Predicate Elimination using PAC Learning Techniques. Technical Report (unpublished), Massachusetts Institute of Technology, 9 December 1993. Available as of January 1996 via "http://www-swiss.ai.mit.edu/~ziggy/descartes.html".

[Blinks 95] T. Blinks. Personal communication, 17 December 1995. Describes an early 70's global weather prediction system; database reconstruction after a crash (a near-daily event) was controlled by a task whose job control code was self-modifying. Even though it was high-level code, only 1–2 line changes were used; attempts at larger-grained modification rapidly grew too complex to be debugged.

[Borning 87] A. Borning. Personal communication, November 1987.

[Boyd] D. Boyd. Stupid Perl Tricks. Available as of July 1995 from "http://www.cs.buffalo.edu/~boyd/perl.html".

[Boyer & Moore 77] R. S. Boyer and J. S. Moore. A Fast String Searching Algorithm. *Communications of the Association for Computing Machinery (CACM)*, 20(10):762–772, October 1977.

[Bratman 61] H. Bratman. An Alternate Form of the "UNCOL Diagram". *Communications of the Association for Computing Machinery (CACM)*, 4(3):142, March 1961.

[Breed 92] L. Breed. Re: Nifty assembler tricks (posting to USENET alt.lang.asm newsgroup), 25 January 1992. Implemented systems that gave access to machine features not available in the HLL (an Algol variant). Implemented on Burroughs 220 and IBM 7090. Procedure calls were patched (inlined) dynamically by callee which then restarted caller. The compilers also generated self-modifying code for address arithmetic and passing arguments by reference.

[Breuel 88] T. M. Breuel. Lexical Closures for C++. In *USENIX Proceedings on C++*, pages 293–304, October 1988.

[Brooks 60] F. P. Brooks, Jr. The Execute Operations – A Fourth Mode of Instruction Sequencing. *Communications of the Association for Computing Machinery (CACM)*, 3(3):168–170, March 1960.

[Brooks 75] F. P. Brooks, Jr. *The Mythical Man Month.* Addison-Wesley Publishing Company, 1975.

[Brooks 87] F. P. Brooks, Jr. No Silver Bullet – Essence and Accidents of Software Engineering. *IEEE Computer*, pages 10–19, April 1987.

[Brown 76] P. J. Brown. Throw-away Compiling. *Software—Practice and Experience*, 6(3):423–434, July-September 1976.

[Brown 95a] C. Brown. Smart Compilers Puncture Code Bloat. *Electronic Engineering Times*, (869):38, 42, 9 October 1995.

[Brown 95b] J. H. Brown. Feedback-Directed Specialization of C. Master's thesis, Massachussetts Inistitute of Technology, June 1995.

[Brunvand & Sproull 89] E. Brunvand and R. F. Sproull. Translating Concurrent Communicating Programs into Delay-Insensitive Circuits. Technical Report CMU-CS-89-126, Carnegie Mellon University, 30 April 1989.

[Bryant 87] R. E. Bryant. Boolean Analysis of MOS Circuits. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):634–649, July 1987.

[Buell & Pocek 93] D. A. Buell and K. L. Pocek, editors. *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Los Alamitos, California, USA, April 1993. IEEE Computer Society Press.

[Bunton & Borriello 90] S. Bunton and G. Borriello. Practical Dictionary Management for Hardware Data Compression. Technical Report 90-01-14, University of Washington, Department of Computer Science and Engineering, 1990.

[Bunton & Borriello 92] S. Bunton and G. Borriello. Practical Dictionary Management for Hardware Data Compression. *Communications of the Association for Computing Machinery (CACM)*, 35(1):95–104, January 1992.

[Burks et al. 46] A. W. Burks, H. H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Technical Report to the U. S. Army Ordinance Department, 1946. Reprinted in [Bell & Newell 71], page 92.

[Burrows 94] M. Burrows. Personal communication, July 1994.

[Burshteyn 90] B. Burshteyn. On The Modification of the Formal Grammar At Parse Time. *ACM SIGPLAN Notices*, 25(5):117–123, May 1990.

[Calder & Grunwald 94] B. Calder and D. Grunwald. Call Prediction in Object-Oriented Languages. In *Proeceedings for the Symposum on Principles of Programming Languages (POPL)*, January 1994.

[Calder et al. 95] B. Calder, D. Grunwald, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Corpus-Based Static Branch Prediction. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 79–92, La Jolla, California, 18–21 June 1995.

[Call et al. 87] L. A. Call, D. L. Cohrs, and B. P. Miller. CLAM – An Open System for Graphical User Interfaces. In *OOPSLA '87 Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 277–286, Orlando, Florida, USA, 4-8 October 1987.

[Callaghan 90] B. Callaghan. Network Extensible File System Protocol Specification. Technical report, Sun Microsystems, 12 February 1990.

[Campbell & Tan 95] R. H. Campbell and S.-M. Tan. Microchoices: An Object-Oriented Multimedia Operating System. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 90–94, 4–5 May 1995.

[Cannon et al. 90] L. W. Cannon, R. A. Elliot, L. W. Kirchhoff, J. H. Miller, J. M. Milner, R. W. Mitze, E. P. Schan, N. O. Whittington, H. Spencer, D. Keppel, and M. S. Brader. *Recommended C Style and Coding Standards.* Specialized System Consultants, Seattle, Washington, 1990. Available as of January, 1996 from "ftp://ftp.cs.washington.edu/pub/cstyle.tar.Z". Originally published as *Indian Hill C Style and Coding Standards*, rewritten by the last three authors; a Japanese translation is also available.

[Carey et al. 88] M. J. Carey, D. J. DeWitt, G. Graef, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. Technical Report #808, University of Wisconsin-Madison Computer Sciences Department, November 1988.

[Carroll 60] L. Carroll. *The Annotated Alice (with introduction and notes by Martin Gardner).* Meridian, 1960.

[Case 91] B. Case. Superscalar Techniques: SuperSPARC vs. 88110. *Microprocessor Report*, pages 1, 6–11, 4 December 1991.

[Cathey 92] J. Cathey. Posting to alt.lang.asm, 27 Jan 1992. Describes graphics library application of self-modifying code.

[Ceri & Pelagatti 84] S. Ceri and G. Pelagatti. *Distributed Databases Principles and Systems.* McGraw-Hill, 1984.

[Chamberlin et al. 81a] D. D. Chamberlin, M. M. Astrahan, W. F. King, R. Alorie, J. W. Mehl, T. G. Price, M. Schkolnick, P. G. Selinger, D. R. Slutz, B. W. Wade, and R. A. Yost. Support for Repetitive Transactions and Ad Hoc Queries in System R. *ACM Transactions on Database Systems*, 6(1):70–94, March 1981.

[Chamberlin et al. 81b] D. D. Chamberlin, A. M. Gilbert, and R. A. Yost. A History of System R and SQL/lData System. In C. Zaniolo and C. DeLobel, editors, *Proceedings of the 7th International Conference on Very Large Databases*, pages 456–464, Cannes, France, September 1981.

[Chambers & Ungar 89] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI)*, pages 146–160, June 1989.

[Chambers & Ungar 91] C. Chambers and D. Ungar. Making Pure Object-Oriented Languages Practical. *OOPSLA '91 Proceedings; SIGPLAN Notices*, 26(11):1–15, November 1991.

[Chambers 92] C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD dissertation, Stanford University, Department of Computer Science, March 1992.

[Chambers et al. 89] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Proceedings*, pages 49–70, 1–6 October 1989.

[Chambers et al. 96] C. Chambers, S. J. Eggers, J. Auslander, M. Philipose, M. Mock, and P. Pardyak. Automatic Dynamic Compilation Support for Event Dispatching in Extensible Systems. In *The Inaugrual Workshop on Compiler Support for Systems Software (WCSSS)*, pages 118–126, 1996.

[Chandross 95] J. Chandross. Personal communication, July 1995.

[Chase & Hood 91] B. Chase and R. Hood. Debugging with Lightweight Instrumentation. In *Workshop on Supercomputer Debugging*, 1991.

[Chase 91] D. Chase. Personal communication, February 1991. Describes dynamically-compiled "stubs" used in the Olivetti Modula-3 compiler to implement lexically-scoped routines, used as C callbacks.

[Chase 95a] D. Chase. USENET comp.compilers posting 95-03-030, 3 March 1995. Describes dynamically-compiled "stubs" used in the Olivetti Modula-3 compiler to implement lexically-scoped routines, used as C callbacks. See "ftp://iecc.com/pub/articles".

[Chase 95b] J. S. Chase. An Operating System Structure for Wide-Address Architectures. Technical Report Ph.D Thesis, University of Washington, Department of Computer Science and Engineering, August 1995.

[Chase et al. 95] D. Chase, C.-M. Chen, I. Dall, W. Daniels, G. Gleason, T. Jehl, D. W. Jones, D. Keppel, T. Mathisen, and G. Mariani. Discussion on USENET comp.arch newsgroup, August-September 1995. A discussion of applications and costs of self-modifying code.

[Cheng et al. 84] J. M. Cheng, C. R. Loosley, A. Shibamiya, and P. S. Worthington. IBM Database 2 performance: Design, implementation, and tuning. *IBM Systems Journal*, 23(2):189–210, 1984.

[Chien 95] A. A. Chien. Personal communication, 22 August 1995.

[Chien et al. 93] A. A. Chien, V. Karamcheti, and J. Plevyak. The Concert System – Compiler and Runtime Support for Efficient, Fine-Grained Concrurent Object-Oriented Programs. Technical Report DCS TR R-93-1815, University of Illinois at Urbana-Champaign, 11 June 1993.

[Chow 88] F. Chow. Minimizing Register Usage Penalty at Procedure Calls. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pages 85–94, 22-24 June 1988.

[Christopher & Wallace 86] T. W. Christopher and R. W. Wallace. Compiling Optimized Array Operations at Run-Time. In *APL 86 Conferance Proceedings*, pages 136–141, Manchester, England, July 1986.

[Clocksin & Mellish 84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, second edition, 1984.

[Cmelik & Keppel 93] R. F. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. Technical Report SMLI 93-12; UWCSE 93-06-06, Sun Microsystems Laboratories, Inc. and University of Washington Department of Computer Science and Engineering, 1993.

[Cmelik & Keppel 94] R. F. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[Cmelik & Keppel 95] R. F. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In T. M. Conte and C. E. Gimarc, editors, *Fast Simulation of Computer Architectures*, chapter 2, pages 5–46. Kluwer Academic Publishers, 1995.

[Cmelik 90] R. F. Cmelik. Personal communication, 1990. Describes internals of Shadow [Hsu 89].

[Cmelik 93] R. F. Cmelik. *The Shade User's Manual.* Sun Microsystems Laboratories, Incorporated, February 1993.

[Cmelik 94] R. F. Cmelik. Personal communication, 1994.

[Cochran 92] K. A. Cochran. USENET alt.lang.asm posting, 29 Jan 1992. Describes self-modifying Zilog Z80 assembly code.

[Cockcroft 93] A. Cockcroft. *Sun Performance Tuning Overview.* Sun Microsystems, June 1993. Available as of May 1995 via anonymous ftp from "ftp://ftp.uu.net/systems/sun/papers/perfpapers" as "sun_performance_tuning_overview.ps.Z".

[Col 95] Omniware: A Universial Substrate for Mobile Code. Technical Report available as of August 1995 from "http://www.colusa.com/html/omni_www.html", Colusa Software, 1995.

[Collier 92a] I. Collier. Posting to USENET newsgroup "alt.lang.asm", 28 Jan 1992. Discusses microcomputer applications of self-modifying code.

[Collier 92b] I. Collier. Posting to USENET newsgroup "alt.lang.asm", 29 Jan 1992. Discussion of how a "string clear" instruction quietly performs the wrong operation if it self-modifies (clears itself).

[Colwell 95] R. Colwell. *The P6 Microprocessor – Distinguished Lecturer Series.* University of Washington, Department of Computer Science and Engineering, 9 November 1995. Co-designer of the Intel P6 (Pentium Pro) microprocessor.

[Colwell et al. 88] R. Colwell, R. Nix, J. O'Donnell, D. Papworth, and P. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. *IEEE Transactions on Computers*, 38(8):967–979, August 1988.

[Com 94] OpenDoc^{tm}: Making Software Work Together. Available via http://www.cilabs.org as of June 1995, August 1994.

[Comer 87] R. S. Comer. Debugging with Dependence Analysis, September 1987. Rice University. Unpublished.

[Consel & Danvy 89] C. Consel and O. Danvy. Partial Evaluation of Pattern Matching in Strings. *Information Processing Letters*, 30:79–86, 30 January 1989.

[Consel & Danvy 90] C. Consel and O. Danvy. From Interpreting to Compiling Binding Times. In N. D. Jones, editor, *ESOP '90, 3rd European Symposium on Programming, (also published as "Lecture Notes in Computer Science 432", Springer-Verlag)*, pages 88–105, Copenhagen, Denmark, May 1990.

[Consel & Noel 96] C. Consel and F. Noel. A General Approach for Run-Time Specialization and its Application to C. In *Proeceedings of the Symposum on Principles of Programming Languages (POPL)*, January 1996.

[Consel 90] C. Consel. Binding Time Analysis for Higher Order Untyped Functional Languages. *LFP '90*, 1990.

[Consel et al. 93] C. Consel, C. Pu, and J. Walpole. Incremental Partial Evaluation: The Key to High Performance, Modularity and Portability in Operating Systems. In *Proceedings of the ACM Conference on Partial Evaluation and Program Manipulation (PEPM)*, pages 44–46, Copenhagen, Denmark, May 1993.

[Conway 58] M. E. Conway. A Proposal For An UNCOL. *Communications of the Association for Computing Machinery (CACM)*, 1(10):5–8, October 1958.

[Copperman & Thomas 95] M. Copperman and J. Thomas. Poor Man's Watchpoints. *ACM SIGPLAN Notices*, 30(1):37–44, January 1995.

[Craske 91] N. Craske. SNOOPS: An Object-Oriented Language Enhancement Supporting Dynamic Program Reconfiguration. *ACM SIGPLAN Notices*, 26(10):53–62, October 1991.

[Crispin 90] M. Crispin. Personal communication, 4 June 1990.

[Crispin 95] M. Crispin. Personal communication, 24 July 1995.

[Crowe 87] M. K. Crowe. Dynamic Compilation in the Unix Environment. *Software—Practice and Experience*, 17(7):455–467, July 1987.

[Cutts et al. 94] Q. I. Cutts, R. C. H. Connor, G. N. C. Kirby, and R. Morrison. An Execution-Driven Approach to Code Optimisation. In *Proceedings of the 17th Australasian Computer Science Conference*, pages 83–92, Christchurch, New Zealand, 1994.

[Dahlby et al. 78] S. H. Dahlby, G. G. Henry, D. N. Reynolds, and P. T. Taylor. The IBM System/38: A High-Level Machine. *IBM System/38: Technical Developments*, pages 47–50, 1978. Reprinted in [Siewiorek et al. 82].

[Dean et al. 95a] J. Dean, C. Chambers, and D. Grove. Selective Specialization for Object-Oriented Languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 93–102, La Jolla, California, USA, June 1995.

[Dean et al. 95b] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 1995 European Conference on Object-Oriented Programming (ECOOP '95)*, pages 77–101, 1995.

[Dean et al. 96] D. Dean, E. W. Felten, and D. S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Okland, California, USA, 6–8 May 1996. Available as of May 1996 from http://www.cs.princeton.edu/sip/pub/secure96.html.

[DEC 90a] DECstation 3100 Desktop Workstation Functional Specification, August 1990.

[DEC 90b] DECstation 5000/200 KN02 System Module Functional Specification, August 1990.

[DEC 92] DECstation and DECsystem 5000 Model 240 Overview, Version 2.0. Technical Report EC-N0194-51, Digital Equipment Corporation, Palo Alto, California, February 1992.

[DeHon & Eslick 94] A. DeHon and I. Eslick. Computational Quasistatics. Technical Report Transit Note #103, Massachusetts Institute of Technology, 4 May 1994.

[DeHon 94] A. DeHon. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. Technical Report Transit Note #100, Massachusetts Institute of Technology, 18 January 1994.

[Deutsch & Grant 71] P. Deutsch and C. A. Grant. A Flexible Measurement Tool for Software Systems. In *International Federation for Information Processing Congress 71 (IFIP), booklet TA-3*, pages 7–12, Ljubljana, August 1971. North-Holland Publishing Co.

[Deutsch & Schiffman 84] P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages (POPL-11)*, pages 297–302, January 1984.

[Deutsch & Taft 80] L. P. Deutsch and E. A. Taft. Requirements for an Experimental Programming Environment. Technical Report CSL-80-10, Xerox Palo Alto Research Center, 1980.

[Deutsch 73] P. Deutsch. *The Interactive Program Verifier*. PhD dissertation, Xerox Palo Alto Research Center, 1973.

[Deutsch 83] L. P. Deutsch. *The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture*. Addison-Wesley, 1983.

[Deutsch 90] P. Deutsch. Personal communication, September 1990. An overview of ParcPlace Systems' Smalltalk-80 implementation, including code storage allocation, code generation, retargeting, and machine-dependent performance tradeoffs.

[Dewdney 84] A. K. Dewdney. "Computer Recreations", Scientific American, May 1984.

[Dig] Digital Equipment Corporation. *OSF/1 for Alpha AXP Calling Standard*.

[Dig 62] *PDP-1 Manual*. Digital Equipment Corporation, Maynard, Massachusetts, USA, 1962.

[Dig 92a] DEC OSF/1 Assembly Language Programmer's Guide. Technical Report (field test draft), Digital Equipment Corporation, October 1992.

[Dig 92b] DEC OSF/1 Programmer's Guide. Technical Report (field test draft), Digital Equipment Corporation, October 1992.

[Dig 94] Digital Leads the Pack with 21164. *Microprocessor Report*, 8(12):1, 6–10, 12 September 1994.

[Dillenger 95] R. S. Dillenger. USENET comp.compilers posting #95-08-110, 20 August 1995. Description of "Super Pascal" (circa 1987), in which types were first-class and the compiler could be called at runtime. See "ftp://iecc.com/pub/articles".

[Ditzel & McLellan 87] D. R. Ditzel and H. R. McLellan. Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero. In *Proceedings of the 14th Annual International Symposium on Computer Architecture (ISCA-14)*, pages 2–9, June 1987.

[Ditzel et al. 87a] D. R. Ditzel, H. R. McLellan, and A. D. Berenbaum. The Hardware Architecture of the CRISP Microprocessor. In *Proceedings of the 14th Annual International Symposium on Computer Architecture (ISCA-14)*, pages 309–319, June 1987.

[Ditzel et al. 87b] D. R. Ditzel, H. R. McLellan, and A. D. Berenbaum. Design Tradeoffs to Support the C Programming Language in the CRISP Microprocessor. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pages 158–163, October 1987.

[Draves 95] S. Draves. Lightweight Languages for Interactive Graphics. Technical Report CMU-CS-95-148, Carnegie Mellon University, 1 June 1995.

[Drummond 95] B. Drummond, September 1995. Details of the Rekursiv O-O processor [Pountain 88], and how object faults were handled below the microcode.

[Ducklin 95] P. Ducklin. Information on Winword Virus, 24 August 1995.

[Dutton et al. 92] T. A. Dutton, D. Eiref, H. R. Kurth, J. J. Reisert, and R. L. Stewart. The Design of the DEC 3000 AXP Systems, Two High-performance Workstations. *Digital Technical Journal*, 4(4), 1992.

[Dyke & Bree 77] E. J. V. Dyke and K. A. V. Bree. A Dynamic Incremental Compiler for an Interpretive Language. *Hewlett-Packard Journal*, pages 17–24, July 1977.

[Eager 95] R. D. Eager. Posting to 'alt.lang.teco', July 1995.

[Eastlake III 72] D. Eastlake III. 11SIM Reference Manual (revised). Technical Report MIT AI Lab Memo #240A, Massachusetts Institute of Technology, February 1972.

[Eastlake III 95] D. Eastlake III. Personal communication, 7 August 1995. Describes self-modifying implementation details of 11SIM [Eastlake III 72] and ITS [Eastlake III et al. 68].

[Eastlake III et al. 68] D. Eastlake III, R. Greenblatt, J. Holloway, T. Knight, and S. Nelson. ITS 1.5 Reference Manual. Technical report, Massachusetts Institute of Technology, June 1968.

[Ebeling 76] C. Ebeling. A User-oriented Microprogramming Support System. Master's thesis, Department of Computers Science, Southern Illinois University, December 1976.

[Eckhouse Jr. & Morris 79] R. H. Eckhouse Jr. and L. R. Morris. *Minicomputer Systems Organization, Programming, and Applications (PDP-11). 2nd Ed.* Prentice-Hall, Inc., 1979.

[Eggers et al. 90] S. J. Eggers, D. R. Keppel, E. J. Koldinger, and H. M. Levy. Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 37–47, May 1990.

[Eggers et al. 96] T. Eggers, D. Gross, A. Kotok, P. R. Samson, and R. A. Wagner, February 1996. Explanation of the operation of the "execute" instruction on the Digital Equipment Corporation PDP-1 [Dig 62].

[Emanuelson 80] P. Emanuelson. *Performance Enhancement in a Well-Structured Pattern Matcher Through Partial Evaluation*. PhD dissertation, Linköping University, Software Systems Research Center, S-581 83 Linköping, Sweden, 1980. Linköping Studies in Science and Technology Dissertations Number 55.

[Engler & Proebsting 94] D. Engler and T. Proebsting. DCG: An Efficient, Retargetable Dynamic Code Generation System. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 263–272, San Jose, California, USA, 5-7 October 1994.

[Engler 96] D. R. Engler. VCODE: A Retargetable Extensible, Very Fast Dynamic Code Generation System. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[Engler et al. 95] D. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A Language for Efficient, Machine-independent Dynamic Code Generation. Technical Report 526, Massachusetts Institute of Technology, Laboratory for Computer Science, 1995.

[Engler et al. 96] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A Language for High-Level, Efficient, and Machine-independent Dynamic Code Generation. In *Proeceedings of the Symposum on Principles of Programming Languages (POPL)*, 1996.

[Epstein 95] D. Epstein. IBM Extends DSP Performance with Mfast. *Microprocessor Report*, 9(16):1, 6–8, 4 December 1995.

[Evans & Darley 65] T. G. Evans and D. L. Darley. DEBUG–An Extension to Current Online Debugging Techniques. *Communications of the Association for Computing Machinery (CACM)*, 8(5):321–326, May 1965.

[Falcone 87] J. R. Falcone. A Programmable Interface Language for Heterogenous Distributed Systems. *ACM Transactions on Compuer Systems*, 5(4):330–351, November 1987.

[Feeley & Lapalme 92] M. Feeley and G. Lapalme. Closure Generation Based On Viewing Lambda as Epsilon Plus Compile. *Computer Languages*, 17(4):251–267, October 1992.

[Felten et al. 92] E. Felten, H. Massalin, D. McNamee, and R. Vaswani. Not One Instruction Too Many (Tee-shirt), 1992. A complete source listing of the context switch code for Synthesis [Massalin & Pu 89].

[Fernandez 95] M. F. Fernandez. Simple and Effective Link-Time Optimization of Modula-3 Programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 103–115, 1995.

[Firestone 95] B. Firestone. Personal communication, August 1995. Describes B1700 microcode programming techniques.

[Fitzhardinge 93] J. Fitzhardinge. USENET comp.compilers posting 93-04-102, 28 April 1993. Reimplementation of PICO [Holzmann 87] that used (runtime) strength reduction, constant folding, loop-invariant code motion and peephole optimizations. See "ftp://iecc.com/pub/articles".

[Fla 94] FlashPort Technology Overview, 1994.

[Flynn & Neuhauser 75] M. J. Flynn and C. Neuhauser. EMMY – An Emulation System for User Microprogramming. In *National Computer Conference*, pages 85–89, 1975.

[Foderaro 90] J. K. Foderaro. Personal communication, 1990. Description of code storage allocation and garbage collection in Franz LISP.

[Foley et al. 90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics — Principles and Practice*, pages 988–992. Addison-Wesley, 1990.

[Ford & Sawamiphakdi 85] R. Ford and D. Sawamiphakdi. A Greedy Approach to Incremental Code Generation. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages (POPL-12)*, pages 165–178, January 1985.

[Franz 94] M. Franz. Technological Steps toward a Software Component Industry. In J. Gutknecht, editor, *Programming Languages and System Architectures; Springer Lecture Notes in Computer Science No. 782*, pages 259–281. 1994.

[Fraser & Henry 91] C. W. Fraser and R. R. Henry. Hard-Coding Bottom-Up Code Generation Tables to Save Time and Space. *Software—Practice and Experience*, 21(1):1–12, January 1991.

[Fraser & Wendt 88] C. W. Fraser and A. L. Wendt. Automatic Generation of Fast Optimizing Code Generators. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pages 79–84, June 1988.

[Fraser 93] C. W. Fraser. Personal communication, 24 June 1993. Describes inefficiencies in the portable interpreter originally used in PICO [Holzmann 87].

[Fraser et al. 84] C. W. Fraser, E. W. Myers, and A. L. Wendt. Analyzing and Compressing Assembly Code. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction; SIGPLAN Notices*, 19(6):117–121, June 1984.

[Fraser et al. 92] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG–Fast Optimal Instruction Selection and Tree Parsing. *ACM SIGPLAN Notices*, 27(4):68–76, April 1992.

[Fre 95] GNU CC source code, 1995. Available from the Free Software Foundation, as of January 1996 via "ftp://prep.ai.mit.edu/pub/gnu/". The latest version will be named starting with "gcc-". See "obstack.{c,h}" and "pself*.c".

[Freeman-Benson 90] B. N. Freeman-Benson. Personal communication, November 1990. Describes implementation details of ThingLab-87, ThingLab-89 and ThingLab-90.

[Frieder & Segal 91] O. Frieder and M. E. Segal. On Dynamically Updating a Computer Program: From Concept to Prototype. *Journal of Systems and Software*, 14(2):111–128, February 1991.

[Fyfe et al. 91] A. Fyfe, I. Soleimanipour, and V. Tatkar. Compiling from Saved State: Fast Incremental Compilation with Traditional Unix Compilers. In *Proceedings of the 1991 USENIX Conference*, pages 161–171, Dallas, Texas, Winter 1991.

[Gaines 72] R. S. Gaines. An Operating System Based on the Concept of a Supervisory Computer. *Communications of the Association for Computing Machinery (CACM)*, 15(3):150–156, March 1972.

[Garrett et al. 92] W. E. Garrett, R. Bianchini, L. Kontothanassis, R. A. McCallum, J. Thomas, R. Wisniewski, and M. L. Scott. Dynamic Sharing and Backward Compatability on 64-bit Machines. Technical Report 418, University of Rochester, April 1992.

[Gerson 62] G. Gerson. On Modifying the 1620 ADD Table. *IBM Systems Journal*, 1:82–83, September 1962.

[Gifford 81] D. K. Gifford. *Information Storage in a Decentralized Computer System.* PhD dissertation, Stanford University, Department of Electrical Engineering, June 1981. Also available as Xerox PARC Technical Report CSL-81-8, Xerox Palo Alto Research Center, Palo Alto California, USA.

[Gill 51] S. Gill. The diagnosis of mistakes in programmes on the EDSAC. *Proceedings of the Royal Society Series A Mathematical and Physical Sciences*, 206(1087):538–554, 22 May 1951.

[Gingell et al. 87] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks. Shared Libraries in SunOS. In *Proceedings of the Summer 1987 USENIX Conference*, pages 131–145, 1987.

[Glew 87] A. Glew. Personal communication, December 1987.

[Gokhale et al. 91] M. Gokhale, W. Holmes, A. Kosper, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building And Using a Highly-Parallel Programmable Logic Array. *IEEE Computer*, 24(1):81–89, January 1991.

[Goldberg & Robson 83] A. Goldberg and D. Robson. *Smalltalk-80: The Language And Its Implementation.* Addison-Wesley, 1983.

[Goodman 88] A. W. Goodman. Personal communication, 1988. Describes design and implementation considerations for an image database and query mechanism [Goodman et al. 89].

[Goodman et al. 89] A. M. Goodman, R. M. Haralick, and L. G. Shapiro. Design of an Integrated Programming Language and Data Management System for Knowledge-Based Computer Vision. Technical Report 89-08-03, University of Washington, Department of Computer Science and Engineering, 28 July 1989.

[Gosling 95] J. Gosling. Oak Intermediate Bytecodes. In *Proceedings of IR '95*, pages 111–118, San Francisco, California, USA, January 1995.

[Granlund & Kenner 92] T. Granlund and R. Kenner. Eliminating Branches using a Superoptimizer and the GNU C Compiler. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI)*, pages 341–352, June 1992.

[Graves 96] R. Graves. Microsoft Continues To Mislead Public About Windows Security Bugs, 10 January 1996. Available as of January 1996 via "ftp://unix.sri.com/risks".

[Greenblatt 95] R. Greenblatt. Personal communication, 21 July 1995. Describes system code and debuggers for various members of the DEC PDP family, microcode for various LISP machines.

[Griswold 89] W. G. Griswold. Personal Communication to Robert R. Henry, 1989.

[Griswold et al. 71] R. E. Griswold, J. F. Page, and I. P. Polonsky. *The SNOBOL4 Programming Language (2nd Ed.)*. Bell Telephone Laboratories, Incorporated, 1971.

[Gross et al. 95] D. Gross, S. R. Russell, P. R. Samson, and R. A. Wagner. Personal communication, August 1995. Describes programming conventions for TX-0 and PDP-1; also debuggers such as FLIT, Microflit and the original DDT.

[Grove et al. 95] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-Guided Receiver Class Prediction. *Proceedings of the 1995 Symposium on Object-Oriented Programming Systems and Languages (OOPSLA '95)*, pages 108–123, October 1995.

[Guibas & Wyatt 78] L. J. Guibas and D. K. Wyatt. Compilation and Delayed Evaluation in APL. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages (POPL-5)*, pages 1–8, 1978.

[Gupta & Chi 90] R. Gupta and C.-H. Chi. Improving Instruction Cache Behavior by Reducing Cache Pollution. In *Proceedings of Supercomputing'90; IEEE Computer Society Press*, pages 82–91, New York, New York, USA, 12–16 November 1990.

[Gutmann 92] P. C. Gutmann. Posting to "alt.lang.asm", 26 Jan 1992. Wrote code that spilled and reloaded registers by writing to "downstream" instruction immediates. Hand-examined Borland C compiler code and determined code patching was used to implement table walk direction changing.

[Gutmann 95] P. C. Gutmann. Personal communication, July 1995. More information on Borland C compiler (see [Gutmann 92]) and on the implementation of Microsoft Windows VxD's, which use self-modifying code.

[Gwennap 95a] L. Gwennap. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, pages 9–15, 16 February, 1995.

[Gwennap 95b] L. Gwennap. New Algorithm Improves Branch Prediction. *Microprocessor Report*, pages 17–21, 27 March, 1995.

[Gwennap 95c] L. Gwennap. Hal Reveals Multichip SPARC Processor. *Microprocessor Report*, pages 1, 6–11, 6 March 1995.

[Haar & Rakizis 93] P. Haar and B. Rakizis. Es: A Shell with Higher-Order Functions. In *Proceedings of the Winter 1993 USENIX Conference*, pages 53–62, January 1993.

[Haertel et al. 90] M. Haertel, J. A. Woods, A. D. Olson, R. M. Stallman, D. A. Gwyn, S. Anderson, H. Spencer, and P. Placeway. *GNU* grep. The Free Software Foundation, 1990. Available as of January 1996 via "ftp://prep.ai.mit/edu/pub/gnu", grep distribution.

[Halfhill 94a] T. R. Halfhill. AMD vs. Superman. *Byte*, pages 95–96, 98, 100, 102, 104, November 1994.

[Halfhill 94b] T. R. Halfhill. T5: Brute Force. *Byte*, pages 123–124, 126, 128, November 1994.

[Halpern 65] M. I. Halpern. Machine Independence – Its Technology and Economics. *Communications of the Association for Computing Machinery (CACM)*, (8), 1965.

[Hamilton & Kougiouris 93] G. Hamilton and P. Kougiouris. The Spring Nucleus: A Microkernel for Objects. Technical Report 93-14, Sun Microsystems Laboratories, Incorporated, 1993.

[Hansen 73] G. J. Hansen. *Adaptive Systems for the Dynamic Run-Time Optimization of Programs.* PhD dissertation, Carnegie-Mellon University, March 1973.

[Hanson 90] D. R. Hanson. Fast Allocation and Deallocation of Memory Based On Object Lifetimes. *Software—Practice and Experience*, 20(1), January 1990.

[Haraldsson 77] A. Haraldsson. *A Program Manipulation System Based on Partial Evaluation.* PhD dissertation, Department of Mathematics, Linköping University, S-581 83 Linköping, Sweden, 1977. Linköping Studies in Science and Technology Dissertations No. 14.

[Haraldsson 78] A. Haraldsson. A Partial Evaluator, and Its Use for Compiling Iterative Statements in Lisp. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages (POPL-5)*, pages 195–202, 1978.

[Haraldsson 80] A. Haraldsson. Experiences from a Program Manipulation System. Technical Report LiTH-MAT-R-80-24, Linköping University, Linköping, Sweden, July 1980.

[Harrenstien 95] K. Harrenstien. Personal communication, July 1995. Describes PDP-10 assembler and operating system programming conventions using self-modifying code.

[Harris 95] G. Harris. Personal communication, 18 September 1995. PDP-11 address mapping was used internally by operating systems.

[Hassitt & Lyon 76] A. Hassitt and L. E. Lyon. An APL Emulator on System/370. *IBM Systems Journal*, (4):358–379, 1976.

[Hayden 94] D. Hayden. Personal communication, August 1994. Describes how one application included a small dynamic compiler that generated target code on-the-fly and then jumped to it; the application was translated using FlashPort [Fla 94] by making it generate and execute host code.

[Hayes] F. Hayes. Threes Rev. 1.1 (Dog and Pony). Vince Emery Presents The Funniest Computer Songs. Tape #VE-05, Vince Emery Productins, P.O. Box 460279 San Francisco, CA 94146. Words available as of July 1995 from "http://www.parc.xerox.com/digitrad/song%3DDOGPONY".

[Heering et al. 89] J. Heering, P. Klint, and J. Rekers. Incremental Generation of Parsers. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI)*, pages 179–191, June 1989.

[Heering et al. 90] J. Heering, P. Klint, and J. Rekers. Incremental Generation of Parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990.

[Heering et al. 91] J. Heering, P. Klint, and J. Rekers. Lazy and Incremental Program Generation (revised version). Technical Report CS-R9124, Centre for Mathematics and Computer Science, Computer Science/Department of Software Technology, Amsterdam, The Netherlands, April 1991.

[Hennessy & Patterson 96] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Plubishers, Inc., San Mateo, California, USA, second edition, 1996.

[Henry 87] R. R. Henry. Code Generation by Table Lookup. Technical Report 87-07-07, University of Washington, Department of Computer Science and Engineering, 1987.

[Henry 88] R. R. Henry. Personal communication, 1988. Describes a "sin" routine implemented in DEC VAX assembly language using self-modifying code.

[Hill et al. 93] M. D. Hill, J. R. Larus, A. R. Leback, M. Talluri, and D. A. Wood. Wisconsin Architectural Research Tool Set. *Computer Architecture News*, 21(4), August 1993.

[Hinds 95] D. Hinds. *Linux PCMCIA HOWTO*, June 1995. Available as of August 1995 via "http://sunsite.unc.edu/mdw/HOWTO/PCMCIA-HOWTO.html".

[Ho & Olsson 90] W. W. Ho and R. A. Olsson. An Approach to Genuine Dynamic Linking. Technical Report CSE-90-25, University of California at Davis, August 1990.

[Ho & Olsson 91] W. W. Ho and R. A. Olsson. An Approach to Genuine Dynamic Linking. *Software—Practice and Experience*, 21(4):375–390, April 1991.

[Ho 90] W. W. Ho. *Dld — A Dynamic Link/Unlink Editor*, 1990. Included with dld software distribution, available as of January 1996 from "ftp://prep.ai.mit.edu/pub/gnu/dld-3.2.3.tar.gz".

[Ho et al. 95] W. W. Ho, W. Chang, and L. H. Leung. Optimizing the Performance of Dynamically-Linked Programs. In *Proceedings of the 1995 USENIX Conference*, pages 225–233, New Orleans, Louisiana, USA, 16-20 January 1995.

[Hölzle & Ungar 94a] U. Hölzle and D. Ungar. A Third-Generation Self Implementation: Reconciling Responsiveness with Performance. In *OOPSLA '94 Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 229–243, Portland, Oregon, October 1994.

[Hölzle & Ungar 94b] U. Hölzle and D. Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 326–336, Orlando, Florida, June 1994.

[Hölzle 94] U. Hölzle. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming.* PhD dissertation, Stanford University, Computer Science Department, 1994.

[Hölzle et al. 91] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In P. America, editor, *Proceedings of the 1991 European Conference on Object-Oriented Programming (ECOOP '91)*, pages 21–38, Geneva, Switzerland, July 1991.

[Hölzle et al. 92] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code With Dynamic De-Optimization. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI)*, pages 32–43, 17–19 June 1992.

[Holzmann 87] G. J. Holzmann. PICO - A Picture Editor. *AT&T Technical Journal*, 66(2):2–13, March 1987.

[Hopcroft & Ullman 79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Publishing Company, 1979.

[Horowitz 88] M. L. Horowitz. *Automatically Achieving Elasticity in the Implementation of Programming Languages.* PhD dissertation, Carnegie Mellon University Computer Science Department, January 1988. Technical report #88-104.

[Hsieh et al. 96] W. C. Hsieh, M. E. Fiuczynski, C. Garret, S. Savage, D. Becker, and B. Bershad. Lanugage Support for Extensible Operating Systems. *The Inaugrual Workshop on Compiler Support for Systems Software (WCSSS)*, 23-24 February 1996.

[Hsu 89] P. Hsu. *Introduction to Shadow.* Sun Microsystems, Incorporated, Mountain View, California, USA, July 1989.

[Hudak & Kranz 84] P. Hudak and D. Kranz. A Combinator-based Compiler for a Functional Language. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages (POPL-11)*, page 122. Yale University, January 1984.

[Hughes 82] R. J. M. Hughes. Super-Combinators. *Proceedings of the ACM Symposium on Lisp and Functional Programming (LFP, LISP)*, 1982.

[Hughes 94] P. Hughes. Personal communication, November 1994.

[Hume 88] A. Hume. A Tale of Two Greps. *Software—Practice and Experience*, 18(11):1063–1072, November 1988.

[Ingalls 81] D. H. H. Ingalls. The Smalltalk Graphics Kernel. *Byte*, 6(8):168–194, August 1981.

[Int 86] *80386 Programmer's Reference Manual*. Intel Corporation, Santa Clara, California, USA, 1986.

[Int 90] *i860 64-bit Microprocessor Programmer's Reference Manual*. Intel Corporation, Mount Prospect, Illinois, USA, 1990.

[James 90] D. James. Multiplexed Busses: The Endian Wars Continue. *IEEE Micro*, pages 9–22, June 1990.

[Jav 96] Java Home Page. Sun Microsystems, 1996. Available as of June 1996 at "http://java.sun.com".

[Johnson & Fox 91] C. Johnson and D. L. Fox. The Silicon Palimpsest — A Programming Model for Electrically Reconfigurable Processors. In *Proceedings of the Second and Third Annual SigFORTH Workshops*, San Antonio, Texas, March 1991.

[Johnson 93] C. Johnson. Personal communication, March 1993. Discussion of what makes individual FPGA parts more or less suited to RTHWR (§10.3.4), including: granularity of modifications; the kinds of logic gates provided by the part; and the reprogramming time.

[Johnston 79] R. L. Johnston. The Dynamic Incremental Compiler of APL\3000. *APL Quote Quad*, 9(4):82–87, June 1979.

[Johnston 95a] D. Johnston. File Filters API, April 1995. Available as of September 1995 via "http://www.ep.se/cool/default.htm".

[Johnston 95b] D. Johnston. Cool Edit, June 1995. Available as of September 1995 via "http://www.ep.se/cool/default.htm".

[Johnston 95c] D. Johnston. *Cool Edit 1.50 User's Manual*. Syntrillium Software Corporation, Phoenix, Arizona, May 1995. Available as of September 1995 via "http://www.ep.se/cool/default.htm".

[Jones 88] N. D. Jones. Challenging Problems in Partial Evaluation and Mixed Computation. *New Generation Computing*, 6:291–302, 1988.

[Jones et al. 89] N. D. Jones, P. Sestoft, and H. Sondergaard. Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. *Lisp and Symbolic Computation*, 2(9-50):10, 1989.

[Joy 90] B. Joy. Personal communication, August 1990.

[Kaepplein 95] M. Kaepplein. Posting to alt.lang.teco, 28 July 1995. Describes graphics routines using RTCG, including bitblt for 68000-based BBN Blt terminals; VAX-based X-windows for QVSS; other image manipulation operations (rotation, scale, skew) for Interleaf Desktop Publishing software. On RISCs, the instruction cache coherency costs outweighed the benefits.

[Kane 87] G. Kane. *MIPS R2000 RISC Architecture*. Prentice Hall, 1987.

[Karinthi & Weiser 87] R. R. Karinthi and M. Weiser. Incremental Re-execution of Programs. *ACM SIGPLAN Notices*, 22(7):38–44, July 1987.

[Karlin et al. 86] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive Snoopy Caching. *Proceedings of the 27th Symposium on Foundations of Computer Science*, pages 244–254, 27–29 October 1986.

[Kay 88] A. Kay. *Distinguished lecturer series presentation.* University of Washington, Department of Computer Science and Engineering, February 1988.

[Kehl 93] T. Kehl. Personal communication, June 1993.

[Kelly 89] P. H. J. Kelly. *Functional Programming for Loosely-coupled Multiprocessors.* Pitman, London/MIT Press, Cambridge Massachusetts, 1989.

[Kempf & Kessler 92a] J. Kempf and P. B. Kessler. Cross-Address Space Dynamic Linking. Technical Report SMLI 92-2, Sun Microsystems Laboratories, Inc., 1992.

[Kempf & Kessler 92b] J. Kempf and P. B. Kessler. Cross-Address Space Dynamic Linking. *IEEE Proceedings of the International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, Sept 1992.

[Kendall 89] S. Kendall. Personal communication, 3 December 1989. Descrbes Saber-C organization including including lazy compilation to virtual machine code and dynamic linking and unlinking of native machine code.

[Keppel & Russell 93] D. Keppel and S. Russell. Faster Dynamic Linking for SPARC V8 and System V.4. Technical Report UWCSE 93-12-08, University of Washington, Department of Computer Science and Engineering, 1993.

[Keppel 91] D. Keppel. A Portable Interface for On-The-Fly Instruction Space Modification. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 86–95, April 1991. As of January 1996, an implementation of the fly interface is available via "ftp://ftp.cs.washington.edu/pub/pardo", with files of the form "`fly-`m`.`n`.tar.gz`".

[Keppel 93a] D. Keppel. Fast Data Breakpoints. Technical Report UWCSE 93-04-06, University of Washington, Department of Computer Science and Engineering, April 1993.

[Keppel 93b] D. Keppel. Managing Abstraction-Induced Complexity. Technical Report 93-06-02, University of Washington, Department of Computer Science and Engineering, June 1993.

[Keppel et al. 91] D. Keppel, S. J. Eggers, and R. R. Henry. A Case for Runtime Code Generation. Technical Report UWCSE 91-11-04, University of Washington, Department of Computer Science and Engineering, November 1991.

[Keppel et al. 93] D. Keppel, S. J. Eggers, and R. R. Henry. Evaluating Runtime-Compiled Value-Specific Optimizations. Technical Report UWCSE 93-11-02, University of Washington, Department of Computer Science and Engineering, November 1993.

[Kernighan & Ritchie 88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, (Second Edition) 1988.

[Kerns 92a] D. Kerns. Personal communication, 8 Oct 1992. Describes dynamic compilation in Oracle databases.

[Kerns 92b] D. Kerns. Personal communication, September 1992. Describes automated compiler tuning in Sun Microsystems compilers, using techniques similar to Baker's [Baker 92a].

[Kessler 90a] P. Kessler. Fast Breakpoints: Design and Implementation. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (PLDI)*, pages 78–84, June 1990.

[Kessler 90b] P. Kessler. Personal communication, 1990.

[Kessler 93] P. Kessler. Personal communication, 1993.

[Kiczales 92] G. Kiczales. Towards a New Model of Abstraction in Software Engineering. In A. Yonezawa and B. C. Smith, editors, *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pages 1–11, Tokyo, Japan, November 1992.

[Kilbane 95] S. Kilbane. USENET comp.compilers posting 95-08-060, 4 August 1995. Describies online reprogramming of embedded systems using programmable logic controllers. See "ftp://iecc.com/pub/articles".

[Killian 90] E. Killian. Personal communication, 1990. Describes instruction cache flushing on some MIPS-based processors. Coherency is implemented by performing stores to the instruction cache with interrupts disabled.

[Knoblock & Ruf 96a] T. B. Knoblock and E. Ruf. Data Specialization. Technical Report MSR-TR-96-04, Microsoft Research, One Microsoft Way, Redmond Washington, 98052, USA, 5 February 1996.

[Knoblock & Ruf 96b] T. B. Knoblock and E. Ruf. Data Specialization. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, USA, May 1996.

[Knuth 70] D. E. Knuth. Von Neumann's First Computer Program. *Computing Surveys*, 2(4):247–260, December 1970. A historical retrospective on von Neumann's sorting program for the EDVAC. The program was never executed by hardware, but may have been the first program to use self-modifying code; predecessor computers such as the Zuse Z-3 used, e.g., non-modifiable punched tape.

[Knuth 71] D. E. Knuth. An Empirical Study of FORTRAN Programs. *Software—Practice and Experience*, 1(2):105–133, April–June 1971.

[Koch & Walker 95] P. Koch and C. Walker. Personal communication, September 1995.

[Kogge 82] P. M. Kogge. An Architectural Trail to Threaded Code Systems. *IEEE Computer*, pages 22–33, March 1982.

[Koldinger 88] E. J. Koldinger. Personal communication, 1988.

[Koopman & Lee 89] P. J. Koopman, Jr. and P. Lee. A Fresh Look at Combinator Graph Reduction (Or, Having a TIGRE by the Tail). In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI)*, pages 110–119, July 1989.

[Koopman 87] P. Koopman, Jr. The WISC Concept. *Byte*, pages 187–188, 190, 192, 194, April 1987.

[Koopman 89] P. Koopman, Jr. *Stack Computers: The New Wave*. Ellis-Horwood (John Wiley and Sons), 1989.

[Koopman et al. 92] P. J. Koopman, P. Lee, and D. P. Siewiorek. Cache Performance of Combinator Graph Reduction. *ACM Transactions on Programming Languages and Systems*, 14(2):265–297, April 1992.

[Krall & Berger 94] A. Krall and T. Berger. A Progress Report on Incremental Global Compilation of Prolog. In K. D. Bosschere, B. Demoen, and P. Tarau, editors, *International Logic Programming Symposium, Workshop on Implementation Techniques for Logic Programming Languages*, pages 59–67, Ithaca, 1994. ALP.

[Krall & Berger 95a] A. Krall and T. Berger. Incremental Global Compilation of Prolog with the Vienna Abstract Machine. In L. Sterling, editor, *Proceedings of the Twelfth International Conference on Logic Programming*, pages 333–347, Tokyo, 1995. MIT Press.

[Krall & Berger 95b] A. Krall and T. Berger. The VAM_{ai} – an Abstract Machine for Incremental Global Dataflow Analysis of Prolog. In M. G. de la Banda, G. Janssens, and P. Stuckey, editors, *Proceedings of the ICLP'95 Post-Conference Workshop on Abstract Interpretation of Logic Languages*, pages 80–91, Tokyo, 1995. Science University of Tokyo.

[KSR 94] KSR/Series Principles of Operations, 1994.

[Lampson 84] B. W. Lampson. Hints for Computer System Design. *IEEE Software*, 1(1):11–28, January 1984.

[Landis et al. 62] N. Landis, A. Manos, and L. R. Turner. Initial Experience With an Operating Multiprogramming System. *Communications of the Association for Computing Machinery (CACM)*, 5(5):282–286, May 1962.

[Lang et al. 86] T. G. Lang, J. T. O'Quin II, and R. O. Simpson. Threaded Code Interpreter for Object Code. Technical Report IBM Technology Disclosure Bulletin 4238-4241, International Business Machines, March 1986.

[Larus & Ball 94] J. R. Larus and T. Ball. Rewriting Executable Files to Measure Program Behavior. *Software—Practice and Experience*, 24(2):197–218, February 1994.

[Lea, Jr. et al. 95] R. Lea, Jr., Y. Yokote, and J. Itoh. Adaptive Operating System Design Using Reflection. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 95–100, 4-5 May 1995.

[Leinecker 93] R. C. Leinecker. Processor Detection Schemes. *Doctor Dobb's Journal*, pages 46–49, 126–127, June 1993.

[Leonard 87] T. E. Leonard. *VAX Architecture Reference Manual.* Digital Press, Bedford, Massachusetts, USA, 1987.

[Leone & Lee 93] M. Leone and P. Lee. Deferred Compilation: The Automation of Run-Time Code Generation. Technical Report CMU-CS-225, Carnegie-Mellon University, 1993.

[Leone & Lee 94] M. Leone and P. Lee. Lightweight Run-Time Code Generation. *Proceeedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, 25 June 1994.

[Leone & Lee 95] M. Leone and P. Lee. Optimizing ML with Run-Time Code Generation. Technical Report CMU-CS-95-205, School of Computer Science, Carnegie Mellon University, December 1995.

[Leone & Lee 96a] M. Leone and P. Lee. A Declarative Approach to Run-Time Code Generation. In *The Inaugrual Workshop on Compiler Support for Systems Software (WCSSS)*, pages 8–17, 23–24 February 1996.

[Leone & Lee 96b] M. Leone and P. Lee. Optimizing ML with Run-Time Code Generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[Leung 92] S. Leung. Personal communication, 29 July 1992.

[Levine 89] J. Levine. Personal communication, 1989. Describes how medium-level code generation is used in mainframe sort programs, which compile the sort criteria into a machine code inner loop.

[Lewis 93] B. Lewis. Personal communication, 25 June 1993. Describes why binary distribution of C++ libraries is problematic, because offsets depend on objects that are not part of the distribution; describes how RTCG solves these problems by performing code generation after the class hierarchy becomes known.

[Lewis et al. 86] D. M. Lewis, D. R. Galloway, R. J. Francis, and B. W. Thomson. Swamp: A Fast Processor for Smalltalk-80. *OOPSLA '86 Proceedings; SIGPLAN Notices*, 21(11):131–139, September 1986.

[Lewis et al. 95] B. T. Lewis, L. P. Deutsch, and T. C. Goldstein. Clarity MCode: A Retargetable Intermediate Representation for Compilation. In *Proceedings of IR '95*, pages 119–128, San Francisco, California, USA, January 1995.

[Linton 90] M. Linton. The Evolution of Dbx. In *Proceedings of the Summer 1990 USENIX Conference*, pages 211–220, Anaheim, California, June 1990.

[Litman 89] A. Litman. Personal communication. Technical report, 1989. Describes dynamic code modification in a predecessor of Shepard [Kessler 90a].

[Littlefield 90] R. Littlefield. Personal communication, 1990. Describes CDC6600 programming techniques.

[Littlefield 94] R. Littlefield. Personal communication, September 1994.

[Locanthi 87] B. N. Locanthi. Fast BitBlt With asm() and CPP. *European Unix Users Group Conference Proceedings (EUUG)*, September 1987.

[Ltd. 96] S. P. Ltd. Guava Softway's High-Performance Environment for Running Java^{TM} Programs, 1996. Sydney, Australia. Available as of May 1996 from http://www.softway.com.au/softway/products/guava.

[Lucco 93] S. Lucco. Personal communication, 15 April 1993.

[Lucco et al. 95] S. Lucco, R. Wahbe, and S. L. Graham. Adaptable Binary Programs. In *Proceedings of the 1995 USENIX Conference*, pages 315–325, New Orleans, Louisiana, USA, 16-20 January 1995.

[Maeda 90] K. Maeda. An Operating System Model for Dynamic Linking with a Single Level Store. Technical Report AIAI-PR-49, Artificial Intelligence Applications Institute, University of Edinburgh, December 1990.

[Magnusson 93] P. Magnusson. Partial Translation. Technical Report T93:5, Swedish Institute of Computer Science, October 1993.

[Mashey 95] J. R. Mashey. USENET comp.arch posting, September 1995. Describes variations in instruction cache coherency implementation, including examples of MIPS/SGI uniprocessor and multiprocessor machines.

[Massalin & Pu 88] H. Massalin and C. Pu. Fine-Grain Scheduling. Technical Report CUCS-381-88, Columbia University, 1988.

[Massalin & Pu 89] H. Massalin and C. Pu. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[Massalin & Pu 91] H. Massalin and C. Pu. A Lock-Free Multiprocessor OS Kernel. Technical Report CUCS-005-91, Columbia University, 1991.

[Massalin & Pu 92] H. Massalin and C. Pu. Reimplementaing the Synthesis Kernel on the Sony NeWS Workstation. In *Proceedings of the Workshop on Microkernels and Other Kernel Architectures*, pages 177–186, Seattle, Washington, April 1992. Usenix Association.

[Massalin 87] H. Massalin. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 122–127, 5-8 October 1987.

[Massalin 90] H. Massalin. Personal communication, 1990. Describes internals of Synthesis [Massalin 92a].

[Massalin 91] H. Massalin. Personal communication, 1991. A description of the Synthesis [Massalin 92a] assembler notation, which includes symbolic support for describing holes in code fragments. A description of how queueing is implemented for low-latency interrupt service routines; the code is updated every time that it is used, but is still profitable due to careful tuning of instructioncache updating and the optimizations enabled by using code modification.

[Massalin 92a] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD dissertation, Columbia University, New York, New York, USA, 1992. Available as technical report CUCS-039-92; see "file://ftp.cs.columbia.edu/reports/reports-1992/cucs-039-92.ps.Z".

[Massalin 92b] H. Massalin. Personal communication, 4 August 1992. Describes Syntheis [Massalin 92a] implementation. The bitblt implementation uses a compiler, as in the BLIT [Pike et al. 85, Locanthi 87]; however the Synthesis bitblt compiler is itself generated dynamically, based on the kind of screen expected by the application (monochrome, greyscale, etc.) and the physical screen provided by the host. Also describes how Synthesis system calls may be specialized at application request; the specialization may be so extensive that no parameters are checked on the call, and failures use a separate return address, so that the application need not test the return value.

[Massalin 95] H. Massalin. Personal communication, July 1995.

[Maurer 84] W. D. Maurer. *Apple Assembly Language.* Computer Science Press, 11 Taft Court, Rockville Maryland, USA, 1984.

[May 87] C. May. Mimic: A Fast S/370 Simulator. *Proceedings of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques; SIGPLAN Notices*, 22(6):1–13, June 1987.

[May 92] T. May. Personal communication, May 1992. The bitblt used in Microsoft Windows^{TM} handles many more cases than the "generic" bitblt, for example monochrome to color, 8-bit greyscale to 2-bit greyscale, etc. The inner loop complexity is correspondingly greater than with the "straightforward" bitblt.

[McCarthy 78] J. McCarthy. History Of LISP. *Preprints for the History of Programming Languages Conference; ACM SIGPLAN Notices*, 13(8):217–223, August 1978.

[McFarling 89] S. McFarling. Program Optimization for Instruction Caches. *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, page 183, April 1989.

[McFarling 91] S. McFarling. Procedure Merging with Instruction Caches. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, pages 71–79, June 1991.

[McGee 65] W. C. McGee. On Dynamic Program Relocation. *IBM Systems Journal*, 4(3), 1965.

[McKeeman 65] W. M. McKeeman. Peephole Optimization. *Communications of the Association for Computing Machinery (CACM)*, 8(7):443–444, July 1965.

[Mead 90] C. Mead. Personal communication, 10 May 1990. Describes what is "code" in various categories of neural networks. Compares against conventional machine code.

[Meissner 95] M. Meissner. USENET comp.compilers posting #95-08-120, September 1995. See "ftp://iecc.com/pub/articles".

[Miranda 87] E. Miranda. BrouHaHa - A Portable Smalltalk Interpreter. *OOPSLA 87 Proceedings; SIGPLAN Notices*, 22(12):354–364, December 1987.

[MIT ed] MIT PDP-10 'Info' file. Technical report, Original publication date not listed. An overview of the PDP-10 instruction set. Available as of July 1995 from ftp://ftp.netcom.com/pub/hb/hbaker/pdp-10/pdp-10.html.

[Mitchell 70] J. G. Mitchell. *The Design and Construction of Flexible and Efficient Interactive Programming Systems.* PhD dissertation, Carnegie-Mellon University, June 1970.

[Mody 92] R. P. Mody. Functional Progrmming is not Self-modifying Code. *ACM SIGPLAN Notices*, 27(11):13–14, November 1992.

[Montz et al. 94] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A Communications-Oriented Operating System. Technical Report 94-20, Department of Computer Science, The University of Arizona, 17 June 1994.

[Moon 74] D. A. Moon. *MacLisp Reference Manual, Revision 0.* Project MAC – Massachusetts Institute of Technology, Cambridge, Massachusetts, 8 April 1974.

[Mosberger et al. 95] D. Mosberger, L. L. Peterson, and S. W. O'Malley. Protocol Latency: MIPS and reality. Technical Report 95-02, Department of Computer Science, The University of Arizona, 1995.

[Mosberger et al. 96] D. Mosberger, P. Bridges, L. L. Peterson, and S. W. O'Malley. Improving the I-Cache Effectiveness of Network Software. *The Inaugrual Workshop on Compiler Support for Systems Software (WCSSS)*, 23-24 February 1996.

[Mot 89a] Motorola Corporation. *MC88100 RISC Microprocessor User's Manual*, 1989.

[Mot 89b] Motorola Corporation. *MC88200 RISC CMMU User's Manual*, 1989.

[Mot 90] Motorola Corporation. *MC68030 User's Manual (third edition)*, 1990. Number MC68030UM/AD, Rev. 2.

[Nan 74] Nanodata Corporation, Williamsville, New York. *Hardware Level User's Manual*, second edition, 31 August 1974. Revision 2.

[Naur 65] P. Naur. The Performance of A System For Automatic Segmentation of Programs Wtithin an ALGOL Compiler (GIER ALGOL). *Communications of the Association for Computing Machinery (CACM)*, 8(11):671–676, November 1965.

[Nelson & Hamilton 93a] M. N. Nelson and G. Hamilton. Higher Performance Dynamic Linking Through Caching. Technical Report TR-93-15, Sun Microsystems Laboratories Inc., April 1993.

[Nelson & Hamilton 93b] M. N. Nelson and G. Hamilton. Higher Performance Dynamic Linking Through Caching. *Proceedings of the 1993 Summer Usenix Conference*, pages 253–265, June 1993.

[Nelson 95] C. Nelson. OpenDoc and Its Architecture. *AIXpert*, August 1995. Available as of May 1996 from "http://www.developer.ibm.com/sdp/library/aixpert/aug95/aixpert_aug95_opendoc.html".

[Nielsen 91] R. D. Nielsen. DOS on the Dock. *NeXTWorld*, pages 50–51, March/April 1991.

[Notkin & Griswold 87] D. Notkin and W. G. Griswold. Enhancement through Extension: The Extension Interpreter. In *Procedings of the ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 45–55, June 1987.

[Notkin & Griswold 88] D. Notkin and W. G. Griswold. Extension and Software Development. *Proceedings of the 10th International Conference on Software Engineering*, pages 274–283, April 1988.

[Nov 95] OpenDoc Technical White Paper. Technical report, Novell – OpenDoc for Windows Team, 17 March 1995. Available from "http://www.cilabs.org/pub/novell/tech/whitep23.html".

[O'Keefe 94] R. A. O'Keefe. *USENET* comp.compilers *posting 94-04-08.* 13 April 1994. System security on the Burroughs B6700 depended on compilers generating trusted code. See "ftp://iecc.com/pub/articles".

[Organick 72] E. I. Organick. *The Multics System.* The MIT Press, 1972.

[Organick 73] E. I. Organick. *Computer System Organization – The B5700/B6700 Series.* ACM Monogrpah Series, Acadmeic Press (a subsidiary of Harcourt brace Jovanovich), 1973.

[Padess 64] A. Padess. The Structure of SYSTEM/360, Part IV – Channel design considerations. *IBM Systems Journal*, 3(2):165–180, 1964.

[Pagan 90] F. G. Pagan. Comparative Efficiency of General and Residual Parsers. *ACM SIGPLAN Notices*, 25(4):59–68, April 1990.

[Par 91] *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1991. Published as ACM SIGPLAN Notices 26(9), September 1991.

[Patterson 60] G. W. Patterson. What is a Code? *Communications of the Association for Computing Machinery (CACM)*, 3(5):315–318, May 1960.

[Paxson 90a] V. Paxson. A Local Agent Architecture for Implementing Efficient Breakpoint Debuggers. An unpublished description of a prototype debugger implementation, 10 December 1990.

[Paxson 90b] V. Paxson. A Survey of Support For Implementing Debuggers, 30 October 1990. CS 262 class project, University of California; available as of June 1995 via “ftp://ftp.ee.lbl.gov//papers/debugger-support.ps.Z”.

[Pennello 93] T. J. Pennello. SOM: Truly Re-usable Class Libraries in Large Programs. Technical Report Slides from a MetaWare talk, MetaWare, Incorporated, Santa Cruz, California, USA, 1993.

[Perlis 82] A. J. Perlis. Epigrams on Programming. *ACM SIGPLAN Notices*, 17(9), September 1982.

[Pettis & Hansen 90] K. Pettis and R. C. Hansen. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[Piersol 94] K. Piersol. A Close-Up of OpenDoc. *Byte*, pages 183–184, 186, 188, March 1994.

[Pike et al. 85] R. Pike, B. N. Locanthi, and J. F. Reiser. Hardware/Software Trade-offs for Bitmap Graphics on the Blit. *Software—Practice and Experience*, 15(2):131–151, February 1985.

[Pittman 87] T. Pittman. Two-Level Hybrid Interpreter/Native Code Execution for Combined Space-Time Program Efficiency. In *Proceedings of the ACM SIGPLAN 1987 Conference on Interpreters and Interpretive Techniques*, pages 150–152, 1987.

[Pittman 95] T. Pittman. The RISC Penalty. *IEEE Micro*, pages 5, 76–80, December 1995.

[Poletto et al. 96] M. Poletto, D. R. Engler, and M. F. Kaaschoek. tcc: A Template-Based Compiler for 'C. *The Inaugrual Workshop on Compiler Support for Systems Software (WCSSS)*, 23-24 February 1996.

[Pollack 91] F. Pollack. *The Future of the Intel 386 ISA – Distinguished Lecturer Series.* University of Washington, Department of Computer Science and Engineering, 25 Oct 1991. A principle architect of the Intel 80386.

[Pollock & Soffa 84] L. L. Pollock and M. L. Soffa. Incremental Compilation of Locally Optimized Code. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages (POPL-12)*, pages 152–164, 1984.

[Popescu et al. 91] V. Popescu, M. Shultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman. The Metaflow Architecture. *IEEE Micro*, pages 10–13, 63–73, June 1991.

[Pountain 88] D. Pountain. Rekursiv: An Object-Oriented CPU. *Byte*, pages 341–342, 344, 346–349, November 1988.

[Pountain 94] D. Pountain. Parallel Course. *Byte*, pages 53–54, 56, 58, 60, July 1994.

[Pow 94] *AMD Shows Off Home-grown Pentium Killer - With RISC Core*. PowerPC News, October 1994. Electronic subscriptions via "add@power.globalnews.com".

[Przybylski 88] S. A. Przybylski. *Performance-Directed Memory Hierarchy Design*. PhD dissertation, Stanford University, September 1988. Available as Stanford technical report CSL-TR-88-366, see pages 21-22, 28-29.

[Przybylski et al. 88] S. Przybylski, M. Horowitz, and J. Hennessy. Performance Tradeoffs in Cache Design. *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 290–298, May 1988.

[Pu & Massalin 91] C. Pu and H. Massalin. Quaject Composition in the Synthesis Kernel. *IEEE Proceedings of the International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, pages 29–34, October 1991.

[Pu & Walpole 93] C. Pu and J. Walpole. A Study of Dynamic Optimization Techniques: Lessons and Directions in Kernel Design. Technical Report OGI-CSE-93-007, Oregon Graduate Institute of Science and Technology, 1993.

[Pu et al. 87] C. Pu, H. Massalin, J. Ioannidis, and P. Metzger. The Synthesis System. Technical Report CUCS-259-87, Columbia University, 1987.

[Pu et al. 88] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis Kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[Pu et al. 94] C. Pu, J. Walpole, and C. Consel. Making Production OS Kernels Adaptive: Incremental Specialization in Practice, May 27 1994.

[Ramsey & Hanson 92] N. Ramsey and D. R. Hanson. A Retargetable Debugger. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI)*, pages 22–31, July 1992.

[Ray 92] T. S. Ray. Evolution and Optimization of Digital Organisms. Technical Report Santa Fe Institute working paper 92-08-042, 1992. Bibliography and further information available as of September 1995 from "http://www.hip.atr.co.jp/~ray/".

[Raymond 93] E. S. Raymond, editor. *The New Hackers Dictionary*. Number ISBN 0-262-18154-1. MIT Press, 1993. Also widely distributed electronically as "The Jargon File 3.0", see, e.g., "http://www.eps.mcgill.ca/jargon/jargon.html".

[Rees & IV 82] J. A. Rees and N. I. A. IV. T: A Dialect of Lisp or, LAMBDA: The Ultimate Software Tool. *Proceedings of the ACM Symposium on Lisp and Functional Programming (LFP, LISP)*, 1982.

[Reigel et al. 72] E. W. Reigel, U. Faber, and D. A. Fisher. The Interpreter – A Microprogrammable Building Block System. *Spring Joint Computer Conference*, pages 705–723, 1972.

[Robbi 72] A. D. Robbi. Microcache: A Buffer Memory for Microprograms. *CompCon 1972 Digest of Papers*, pages 123–125, September 1972.

[Robinson & Owen 73] A. Robinson and G. J. Owen. DEBUG – An On-Line Program Debugging Aid. In *Proceedings of the Confrence on Software Engineering for Telecommunications Switching Systems*, pages 135–143, 2-5 April 1973.

[Rodriguez Jr. 91] L. H. Rodriguez Jr. Coarse-Grained Parallelism Using Metaobject Protocols. Technical Report SSL-91-06, Xerox Palo Alto Research Center (PARC), 1991. Also published as a Masters thesis, Massachusetts Institute of Technology.

[Rosenblum et al. 95] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, 3(4):34–43, Winter 1995.

[Roskos & Winner 81] J. E. Roskos and R. I. Winner. Toward User Sharing of the Microprogramming Level Under Unix on the Perkin-Elmer 3220. *IEEE Micro-14*, pages 67–73, 1981.

[Rossi 92] T. Rossi. USENET "alt.lang.asm" posting, 29 January 1992. Describes a protection system he observed in which code was both decrypted on demand and then later replaced with the encrypted original.

[Rowe 86] L. Rowe. *Class Lecture, CS164*. University of California, Berkeley; Department of Electrical Engineering and Computer Science, 1986. Description of a COBOL compiler that ran on PC-class machines, in which overlay usage was driven based on available memory.

[Russell 95a] S. R. Russell. Personal communication, August 1995. (Original designer and implementor of Spacewar.).

[Russell 95b] S. R. Russell. Personal communication, July 1994, December 1995.

[Russell 95c] S. R. Russell. Personal communication, October 1995.

[Saal & Weiss 79] H. J. Saal and Z. Weiss. A Software High Performance APL Interpreter. *Quote Quad*, 9(4):74–81, June 1979.

[Sabatella 90] M. Sabatella. Issues in Shared Libraries Design. In *Proceedings of the Summer 1990 USENIX Conference*, pages 11–23, June 1990.

[Saltz et al. 90] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and Runtime Compilation. *Proceedings of the International Workshop on Compilers for Parallel Computers*, December 1990.

[Samples et al. 86] A. D. Samples, D. Ungar, and P. Hilfinger. SOAR: Smalltalk Without Bytecodes. *OOPSLA '86 Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, 21(11):107–118, September 1986.

[Samuelsson 94] D. Samuelsson. System Level Interpretation of the SPARC V8 Instruction Set Architecture. Technical Report R94:23, Swedish Institute of Computer Science, 1994.

[Sand 95a] D. Sand. Personal communication, 1 May 1995. A detailed discussion of the technical merits and demerits of writable microcode compared to hardwired control store, including both ROM and random logic.

[Sand 95b] D. Sand. Personal communication, 10 May 1995. The Burroughs B1800 used a microcode cache.

[Sandberg et al. 85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In *Proceedings of the Summer 1985 USENIX Conference*, 1985.

[Santi 89] O. Santi. Retargeting of an Incremental Code Generator to the MC68020. Technical Report LiTH-IDA-R-89-13, Linköping University, Department of Computer and Information Science, S-581 83 Linköping, Sweden, March 1989.

[Seeley 90] D. Seeley. Shared Libraries as Objects. In *Proceedings of the Summer 1990 USENIX Conference*, pages 25–37, June 1990.

[Segal & Frieder 89a] M. E. Segal and O. Frieder. Dynamic Program Updating: A Software Maintenance Technique for Minimizing Software Downtime. *Software Maintenance: Research and Practice*, 1(1):59–79, September 1989.

[Segal & Frieder 89b] M. E. Segal and O. Frieder. Dynamically Updating Distributed Software: Supporting Change in Uncertain and Mistrustful Environments. *Proceedings of the Conference on Software Maintainance–1989*, pages 254–261, 16-19 October 1989.

[Segal & Frieder 93] M. E. Segal and O. Frieder. On-The-Fly Program Modification: Systems for Dynamic Updating. *IEEE Software*, pages 53–65, March 1993.

[Segal 92] M. E. Segal. Extending Dynamic Program Updating to Support Distributed Systems that Communicate via Remote Evaluation. *Proceedings of the International Workshop on Configurable Distributed Systems*, March 1992.

[Seltzer et al. 96] M. Seltzer, C. Small, and M. D. Smith. Symbiotic Systems Software: Fast Operating Systems for Fast Applications. *The Inaugrual Workshop on Compiler Support for Systems Software (WCSSS)*, 23-24 February 1996.

[Seq 87] *Symmetry Technical Summary*, pages A7–A10. Sequent Computer Systems Incorporated, 1987. Revision 1.5.

[SGI 94] *IRIX Systems Programming Guide*. Silicon Graphics, Incorporated, 1994.

[Shaw 74] A. C. Shaw. *The Logical Design Of Operating Systems*. Prentice-Hall, 1974.

[Shipley 91] W. J. Shipley. Personal communication, 7 March 1991. Implemented a graphic programming language for data analysis and manipulation; user expressions were preprocessed into C, then compiled and dynamically linked.

[Shub 90] C. M. Shub. Native Code Process-Originated Migration in a Heterogenous Environment. In *Proceedings of the 1990 Computer Science Conference*, pages 266–270, February 1990.

[Siewiorek et al. 82] D. P. Siewiorek, C. G. Bell, and A. Newell, editors. *Computer Structures: Readings and Examples*. McGraw-Hill, Incorporated, 1982.

[Simonyi 94] C. Simonyi. Personal communication, December 1994. Describes code-mofification techniques used with the URAL and GIER computers.

[Sirer et al. 96] E. G. Sirer, M. Fiuczynski, P. Pardyak, and B. N. Bershad. Safe Dynamic Linking in an Extensible Operating System. *The Inaugrual Workshop on Compiler Support for Systems Software (WCSSS)*, pages 141–148, 23-24 February 1996.

[Sites 92] R. L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[Sites 93] R. Sites. Alpha AXP Architecture. *Communications of the Association for Computing Machinery (CACM)*, 36(2):33–44, February 1993.

[Sites et al. 93] R. L. Sites, A. Chernoff, M. B. Kerk, M. P. Marks, and S. G. Robinson. Binary Translation. *Communications of the Association for Computing Machinery (CACM)*, 36(2):69–81, February 1993.

[Sloan 90] K. Sloan. Personal communication, 1990.

[Sloman 88] A. Sloman. USENET comp.compilers posting 88-01-029, 30 January 1988. See "ftp://iecc.com/pub/articles".

[Smith 82] B. C. Smith. *Reflection and Semantics in a Procedural Language*. PhD dissertation, Massachusetts Institute of Technology, January 1982. Available as technical report MIT-LCS-TR-272.

[Smith 84] B. C. Smith. Reflection and semantics in Lisp. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages (POPL-11)*, page 23, January 1984.

[Smotherman 94] M. Smotherman. USENET comp.compilers posting 94-10-126, 16 October 1994. Survey of instruction cache performance tuning literature. See "ftp://iecc.com/pub/articles".

[Snyder 82] L. Snyder. The Configurable Highly Parallel Computer. *IEEE Computer*, pages 47–56, January 1982.

[Sosič & Johnson 94] R. Sosič and R. R. Johnson. Computational Properties of Self-Reproducing Automata. Technical Report CIT-94-7, School of Computing and Information Technology, Griffith University, Nathan, Queensland, Australia, 1994.

[Sosič 92] R. Sosič. Dynascope: A Tool for Program Directing. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI)*, pages 12–21, July 1992.

[Sosič 94] R. Sosič. Design and Implementation of Dynascope, a Directing Platform for Compiled Programs. Technical Report CIT-94-7, School of Computing and Information Technology, Griffith University, 1994.

[SPA 92] SPARC International, Incorporated. *The SPARC Architecture Manual, Version Nine*, 1992.

[Spacek 72] T. R. Spacek. A Proposal To Establish a Pseudo Virtual Memory via Writable Overlays. *Communications of the Association for Computing Machinery (CACM)*, 15(6):421–426, June 1972.

[Stallman 86] R. M. Stallman. *GNU Emacs Manual*. Free Software Foundation, Cambridge, Massachusetts, October 1986. Fifth Edition, Emacs Version 18.

[Stallman 87] R. M. Stallman. *GDB Manual: The GNU Source-Level Debugger*. Free Software Foundation, Cambridge, Massachusetts, January 1987.

[Stallman 89] R. M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Cambridge, Massachusetts, 12 September 1989.

[Stallman 93] R. M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Cambridge, Massachusetts, 1993.

[Stallman 95] R. M. Stallman. Personal communication, July 1995. Describes implementation of PDP-10 TECO.

[Stamos & Gifford 90] J. W. Stamos and D. K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[Standish 80] T. A. Standish. *Data Structure Techniques*. Addison-Wesley, 1980.

[Steel Jr. 82] G. L. Steel Jr. An Overview of Common LISP. *Proceedings of the ACM Symposium on Lisp and Functional Programming (LFP, LISP)*, 1982.

[Stemple et al. 92] D. Stemple, R. B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G. N. C. Kirby, L. Fegaras, R. L. Cooper, R. C. H. Connor, M. P. Atkinson, and S. Alagic. Type-Safe Linguistic Reflection: A Generator Technology. Technical Report FIDE/92/49, ESPRIT BRA Project 3070 FIDE, 1992.

[Stemple et al. 93] D. Stemple, R. Morrison, G. N. C. Kirby, and R. C. H. Connor. Integrating Reflection, Strong Typing and Static Checking. *Proceedings of the 16th Australian Computer Science Conference*, pages 83–92, 3–5 February 1993.

[Stern 81] N. Stern. *From ENIAC to UNIVAC – An Appraisal of the Eckert-Mauchly Computers*. Digital Press, 1981.

[Stone & Cocke 91] H. S. Stone and J. Cocke. Computer Architecture in the 1990s. *IEEE Computer*, pages 30–38, September 1991.

[Stonebraker & Rowe 85] M. Stonebraker and L. A. Rowe. The Design of POSTGRES. Technical Report Memoradum No. UCB/ERL 85/95, Electronics Research Laboratory, University of California, Berkeley, 15 November 1985.

[Stonebraker et al. 90] M. Stonebraker, L. A. Rowe, and M. Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[Strack 95] S. Strack, editor. *Core War Frequently Asked Questions (USENET* rec.games.corewar *FAQ)*. 10 June 1995. Available as of July 1995 from "http://www.stormking.com/~koth/corewar-faq.html" and "ftp://rtfm.mit.edu/pub/usenet/news.answers/games/corewar-faq".

[Strong et al. 58] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel. The Problem of Programming Communication with Changing Machines; A Proposed Solution. *Communications of the Association for Computing Machinery (CACM)*, 1(8, 9):12–18, 9–15, 1958.

[Sun 89] Sun Microsystems, 2550 Garcia Avenue, Mountain View, California, USA. *SunOS 4.1 Manual*, September 1989. See the entry for "dlopen()".

[Sun 91] Sun Microsystems, 2550 Garcia Avenue, Mountain View, California, USA. *The SPARC Architecture Manual*, January 1991. Part Number 800-1399-12.

[Sun 94] Sun Microsystems, 2550 Garcia Avenue, Mountain View, California, USA. *Debugging a Program*, 1994. Product #802-3517-10.

[Sutherland 91] I. Sutherland. Personal Communciation, March 1991. Describes applications of the techniques described in [Brunvand & Sproull 89].

[Theimer & Hayes 92] M. Theimer and B. Hayes. Heterogeous Process Migration by Recompilation. Technical Report CSL-92-3, Xerox Palo Alto Research Center (PARC), Palo Alto, California, March 1992.

[Thekkath & Levy 93] C. A. Thekkath and H. M. Levy. Limits to Low Latency Communication on High-Speed Networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[Thekkath 91] C. Thekkath. Personal communication, 7 March 1991.

[Thompson & Ryan 94] T. Thompson and B. Ryan. PowerPC 620 Soars. *Byte*, pages 113–114, 116, 118, 120, November 1994.

[Thompson 68] K. Thompson. Regular Expression Search Algorithm. *Communications of the Association for Computing Machinery (CACM)*, 11(6):419–422, June 1968.

[Thompson 84] K. Thompson. Reflections on Trusting Trust (Turing Award lecture). *Communications of the Association for Computing Machinery (CACM)*, 27(8):761–763, August 1984.

[Thornton 64] J. E. Thornton. Parallel Operation in the Control Data 6600. *American Federation of Information Processing Societies (AFIPS) Proceedings of the Fall Joint Computer Conference (FJCC)*, 26(II):33–40, 1964. Reprinted in [Siewiorek et al. 82].

[Torrellas et al. 92] J. Torrellas, A. Gupta, and J. Hennesy. Characterizing the Caching and Synchronization performance of a Multiprocessor Operating System. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 162–174, October 1992.

[Tucker & Flynn 71] A. B. Tucker and M. J. Flynn. Dynamic Microprogramming: Processor Organization and Programming. *Communications of the Association for Computing Machinery (CACM)*, 14(4):240–250, April 1971.

[Turley 94] J. L. Turley. Thumb Squeezes ARM Code Size (New Core Module Provides Optimized Second Instruction Set). *Microprocessor Report*, pages 1, 6–9, 27 March 1994.

[Turner 79] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software—Practice and Experience*, 9(1):31–49, January 1979.

[Uhlig et al. 94] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest. Trap-driven Simulation with Tapeworm II. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 132–144, San Jose, CA., 5-7 October 1994.

[Ungar & Patterson 87] D. Ungar and D. Patterson. What Price Smalltalk? *IEEE Computer*, 20(1):67–74, January 1987.

[Ungar et al. 84] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson. Architecture of SOAR: Smalltalk on a RISC. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA-11)*, pages 188–197, Ann Arbor, MI, June 1984.

[Ungar et al. 92] D. Ungar, R. B. Smith, C. Chambers, and U. Hölzle. Object, Message, and Performance: How They Coexist in Self. *IEEE Computer*, pages 53–64, October 1992.

[van Dorn et al. 95] L. van Dorn, P. Homberg, and A. S. Tanenbaum. Paramecium: An Extensible Object-Based Kernel. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, May 1995.

[Veenstra & Fowler 94] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 201–207, January 1994.

[Veenstra 93] J. E. Veenstra. Mint Tutorial and User Manual. Technical Report 452, University of Rochester Computer Science Department, May 1993.

[Vegdahl & Pleban 89] S. Vegdahl and U. F. Pleban. The Runtime Environment for Screme, a Scheme Implementation on the 88000. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 172–182, April 1989.

[von Neumann 45] J. von Neumann. *First Draft of a Report on the EDVAC*. University of Pennsylvania, 30 June 1945. Appears in [Stern 81], pp. 177-246.

[Wagner et al. 94] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. Accurate Static Estimators for Program Optimization. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 85–96, Orlando, Florida USA, 20-24 June 1994.

[Wahbe 92] R. Wahbe. Efficient Data Breakpoints. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 200–212, October 1992.

[Wahbe et al. 93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Symposium on Operating System Principles*, pages 203–216, 5–8 December 1993.

[Wahbe et al. 94] R. Wahbe, S. Lucco, and S. L. Graham. Adaptable Binary Programs. Technical Report CMU-CS-94-137, Carnegie Mellon University School of Computer Science, April 1994.

[Wakeling 95] D. Wakeling. A Throw-away Compiler for a Lazy Functional Language. In *Proceedings of the Fuji Internation Workshop on Functional and Logic Programming*, pages 203–216, Susono, Japan, July 1995.

[Wall & Schwartz 91] L. Wall and R. L. Schwartz. *Programming Perl*, pages 60–64, 189–190. O'Reilly and Associates, Incorporated, 1991. See "When Subroutines Aren't Enough".

[Wall 89] D. W. Wall. Link-Time Code Modification, September 1989. DEC WRL Research Report 89/17.

[Wall 90] D. W. Wall. Predicting Program Behavior Using Real or Estimated Profiles. Technical Report WRL Technical Note TN-18, Digital Equipment Corporation, Western Research Laboratory, December 1990.

[Wall 91] D. W. Wall. Predicting Program Behavior Using Real or Estimated Profiles. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, June 1991.

[Wand 88] M. Wand. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. *Lisp and Symbolic Computation*, 1(37):11–37, 1988.

[War 95] Warning on MSN Icons. *Information Week*, page 24, 27 August 1995. As cited in Edupage, "info@ivory.educom.edu", 28 August 1995.

[Ward 95] B. Ward. The Linux Kernel HOWTO. Technical report, March 1995. Available as of August 1995 from "http://sunsite.unc.edu/mdw/HOWTO/Kernel-HOWTO.html".

[Weston 95] S. Weston. Netscape 2.0b2 Allows For Invasion of Privacy (USENET posting), November 1995. Netscape 2.0b2 "LiveScripts" are allowed to access entire session history, including passwords, then download that history to a remote site (fixed in 2.0b3).

[Wharton 91] J. Wharton. A Software Emulation Primer: How Will Application Program Rehosting Affect the Quality of Life? *Microprocessor Report*, 5(18):17–18, October 2, 1991.

[Wharton 92] J. Wharton. Recycling Programs: Software Rehosting Part 2. *Microprocessor Report*, page 23, 27 May 1992.

[Wheeler 50] D. J. Wheeler. Programme Organization and Initial Orders for the EDSAC. *Proceedings of the Royal Society, Series A*, 202:573–589, 1950.

[White 95] J. L. White. Personal communication, July 1995.

[Wilkes 69] M. V. Wilkes. The Growth of Interest in Microprogramming: A Literature Survey. *Computing Surveys*, 1(3):139–145, September 1969.

[Wilkes 72] M. V. Wilkes. The Use of a Writable Control Memory in a Multiprogramming Environment. *Fifth Workshop on Microprogramming*, pages 62–65, 1972.

[Wilkinson et al. 90] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris Architecture and Implementation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):63–75, March 1990.

[Wilner 72] W. T. Wilner. Design of the Burroughs B1700. *Fall Joint Compuer Conference*, pages 489–497, 1972.

[Winroth 92] H. Winroth. External Object Representation. Technical report, 16 November 1992. Available as of May 1995 via "http://www.bion.kth.se/tests/xor/report/report.html".

[Yarvin & Sah 93] C. Yarvin and A. Sah. Personal communication, 6 October 1993.

[Yarvin & Sah 94] C. Yarvin and A. Sah. Binary Optimization for Portable Runtime Code Generation in C. Technical Report UCB-CSD-94-792, University of California Berkeley, Department of Electrical Engineering and Computer Science, 1994.

[Yeh & Patt 91] T. Yeh and Y. Patt. Two-Level Adaptive Branch Prediction. In *Proceedings of the 24th International Symposium on Microarchitecture*, pages 51–61, November 1991.

[Yellin & Stroom 88] D. Yellin and R. Stroom. INC: A Language for Incremental Computations. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pages 115–124, Atlanta, Georgia, USA, 22-24 June 1988.

[Yokote 92] Y. Yokote. The Apertos Reflective Operating System. In *OOPSLA '92 Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, October 1992.

[Ziv & Lempel 78] J. Ziv and A. Lempel. Compression of Individual Squences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, II-24(5):530–536, September 1978.

Appendix A

Glossary

This appendix provides definitions and further explanations for terms used in this dissertation.

ABI: An *Applications Binary Interface* is an instruction set and encoding for those instructions. The ABI is standardized across members of an architectural family. A program that is ABI-compatible may be executed on any ABI-compliant machine without recompilation. Note that transfers out of the program (e.g., system calls, traps) have undefined effect. Note also that a given machine may also implement features that are not part of the ABI; applications that take advantage of these features are not ABI-compliant and may not work across machines.

API: The *Applications Program Interface*, which is a set of services (e.g., provided by an operating system) plus a standard convention for using them.

APPLICATION: A code fragment, program, or collection of programs and operating system code that together perform useful work that is thought of as a single unit. For example, a file server and collection of client programs might together comprise a database application. Applications may also be components of other applications.

ARCHITECTURE: When applied to a processor, the *architecture* describes the results (side effects on application-visible registers and memory) of executing various instructions, but it does not explain the timing of the various operations or the effects on hidden state (e.g. caches). When applied to a computer system, *architecture* describes how the system components (processors, memories, busses, secondary storage, I/O devices) are connected and thus describes various bottlenecks qualitatively but not quantitatively. Compare to implementation.

BACK END: The part of a compiler which transforms a machine-independent compiler IR into executable machine code. Called the "back end" because it is one of the last stages to execute. Compare to "front end".

CISC: An acronym for *Complex Instruction Set Computer*, and contrasted against RISC, *Reduced Instruction Set Computer*. There is no universal definition of CISC or RISC, but common features of CISCs include variable-length instruction encodings, multiple side-effects per instruction,[1] and memory-to-memory operations. Common features of RISCs include segregated integer and floating-point register files, larger numbers of registers than CISCs, and an instruction set that accesses memory using only load and store instructions. For the purposes of defining CISC and RISC, the Intel 80x86, Motorola 680x0 and DEC VAX are all CISCs; the DEC AXP, MIPS and SPARC are all RISCs.

DYNAMIC OPTIMIZATION: Work performed at runtime for the express reason of reducing the execution time of the application. Optimizations can include code optimizations that produce new code and data optimizations that reduce the number of invocations of some code fragment. A classic example of the latter is that multiplying three matrices, `A*B*C`, can have drastically different execution time depending on whether the code is executed as `(A*B)*C` or as `A*(B*C)` [Standish 80].

[1] Most processors actually have multiple side-effects per instruction, since both the operation result and program counter update are computed by the instruction. A large number of machines also update condition code registers. Here, "multiple side-effects" means that more than one of the ordinary registers and/or memory locations are updated by an instruction, with the exception that updates caused by "load multiple" and "store multiple" instructions are considered to be a single update.

FRONT END: One of the first stages of a compiler, usually responsible for lexing, parsing, and other early manipulations of the code as it enters the compiler. Compare to "back end".

GENERALLY: Can mean either "usually" or "in the most general case". Used in the latter sense only in this thesis.

IMPLEMENTATION: The *implementation* of a processor or system is the real hardware that executes programs. For the purpose of measurement, a given program may run correctly on several implementations, but have different performance characteristics.

INFINITE LOOP: See: "infinite loop", pg. 227.

INSTRUCTION MIX: The instruction set of a processor may be divided into categories, such as loads, stores, arithmetic and so on. The *instruction mix* is the average number instructions executed from each of the categories. The instruction mix is thus a bulk statistic; and is useful for discerning certain types of bottlenecks. It does not, however, necessarily provide low-level information such as the frequency of write stalls, how often superscalar issue is possible, and so on.

INTERPOSITION: *Interposition* is when an extra routine is inserted between caller and callee. Interposition is used for a variety of reasons including functional extension, monitoring and so on. For example, a file system naming scheme may be extended by interposing a name translator between the caller to the file-open routine and the file-open routine itself; the name translation routine uses the "virtual" name passed by the caller in order to generate a "physical" name that is passed to the real file-open.

ITERATION SPACE: For a given set of variables, the iteration space is all combinations of all values of the variables. For example, if `i` can take on the values {`1, 2, 3`} and `j` the values {`42, 53`}, then the iteration space is {`1, 42`}, {`2, 42`}, {`3, 42`}, {`1, 53`}, {`2, 53`}, {`3, 53`}.

LOCALLY-OPTIMAL CODE GENERATION: "Local" describes basic blocks or extended basic blocks (with a single entry but possibly multiple exits). That is, local code generation does not consider inter-block information in making code generation decisions and generates code for each block in isolation. "Optimal" typically refers to specific metrics, typically static metrics, such as the size of the code or the number of instructions and not necessarily execution time. "Locally-optimal" code is always best with respect to the specific metrics and window of code being considered; that is, no other code generator can produce code that is better in those metrics except by considering information outside of the block.

MODEL: An estimate of a thing's behavior, characterizing a metric of interest without providing full information about other details. For example, a model of an instruction's execution time is one cycle per instruction. A better model includes instruction pipeline and memory stalls. Neither model includes details about, say, the side-effects of actually executing the instruction.

N-CODE: Native machine code; that is, code that will be executed by hardware. Usually, n-code is executed by hardware, whereas v-code is normally executed by an interpreter or compiled to machine code. Compre to "v-code".

OPTIMIZE: Used in the sense of "improvement" rather than in the sense of optimality, "to improve to be as good as is theoretically possible", or in the dictionary sense "to view with optimism."

An optimization may be characterized as a *predicate* and a *transformation*. The latter says how to restructure the code to implement the optimization, while the former describes preconditions that must be true in order for the optimization to be possible and profitable.

P-CODE: Code for a pseudo-machine . See "v-code".

PARTIAL EVALUATION: A procedure may be specialized with respect to some of its arguments; the process of specialization is called *partial evaluation*. A *self-applicable* partial evaluator may be applied simultaneously to itself and to a general-purpose routine in order to produce a special-purpose specializer called a *generating extension*.

RISC: Originally an acronym for "Reduced" or "Regular Instruction-Set Computer", it is currently used to identify processor architectures that perform memory, ALU and branch operations using separate instructions, and where there are a limited number of side-effects per instruction (typically one plus condition codes for ALU operations). Compare to "CISC".

ROUTINE: A procedure or function.

RUNTIME: Computations that happen after (some of) the dataset becomes available are considered to happen at runtime. Computations that happen before the dataset is available are called *static* computations.

As with "what is code", answering "what is runtime" requires agreement about what is the "dataset". Sometimes, for example, data values are pre–bound in order to improve the efficiency of statically-compiled code. Note also that some runtime computations don't depend on program data.

For example, dynamically-linked libraries may be bound at runtime but before any data values are read by the program.

RUNTIME CONSTANT: A variable whose value is unchanged over some part of a program's execution. Note that all variables are constant over sufficiently small regions of execution, such as a few machine instructions. The interesting feature of runtime constants is that a runtime constant is unchanged over the time that some code fragment is executed repeatedly; it may thus be used to drive runtime specialization.

SAFE OPTIMIZATIONS: An optimization is *safe* if it is only applied where it can be proven to produce correct results. An *unsafe* optimization can sometimes produce faster code, but may occasionally produce erroneous code. Optimization safety is generally stated with respect to a particular programming language; for example, interchanging the order of two memory reads that have no intervening operations is generally safe in C but often unsafe in assembler, where reads may reference e.g. device control registers.

SEMANTICS: Throughout, "semantics" is used in the shallow sense of computer science and not in the deep sense of linguistics, philosophy or cognitive science.

SPECIALIZATION VARIABLE: A variable whose current value is used to make code faster and less general. For example, if `f (a, b)` is called with `a=3`, then it is possible to generate `f' (b)`, which uses `a` as a specialization variable.

STATIC: See runtime.

STATICALLY-TYPED: In this dissertation, "statically-typed" means "type" used in the sense of C or FORTRAN. In particular, it refers to programming languages which lack constructs that are implemented using dynamic type dispatch. For example, a conventional definition might describe C++ as statically-typed; for the purposes of this dissertation, however, C++ is excluded because it has type dispatch overhead.

As discussed in Part II, RTCG can be applied to many kinds of applications. The reason for using such a narrow definition of "statically-typed" is that the dissertation measurements in Part I thus show only improvements from elimination of value dispatch overhead.

TYPE: *Type* is an abstraction of an entity's characteristic or behavior, intended to capture just the important features of the entity (§9.5). Since importance depends on the application, the information conveyed by a type may vary based on application; for example, in one circumstance the size may be unimportant while the operations are crucial, whereas in another situation the size may be crucial while the set of operations is unimportant.

There is a fuzzy distinction between type and value. For example a value range might be considered either a value or a type, and individual values that cause singularities might be considered as special subtypes. For the purposes of this dissertation, "type" and "statically typed" are used in the sense of C [Kernighan & Ritchie 88] and other such Algol variants.

V-CODE: Code for a virtual machine instruction set, also known as "pseudocode" or "p-code". It is executed using an interpreter or by translating it to another form such as n-code. n-code is typically executed by hardware. Compare to "n-code"; see also the discussion of code in Section 9.1.

WALLCLOCK TIMER: A timer that maps real-time, or the elapsed time seen by an independent observer watching a clock on the wall. Also called an *elapsed time timer*. Captures all effects including I/O overhead, but suffers from interference from other tasks that compete for machine resources. Wallclock timers are to be compared to timers that measure time only when e.g. the process is running, but which are stopped whenever the process is stopped.

Appendix B

Further Explanations

In all pointed sentences, some degree of accuracy must be sacrificed to conciseness.

–Samuel Johnson

The three dots "..." here suppress a lot of detail — maybe I should have used four dots.

–Donald Knuth

Previous chapters make several simplifications and omissions in order to make the descriptions more understandable. This chapter fixes those sleights by providing a variety of further explanations.

B.1 Exponential Costs

Previous chapters described both static and VSO'd code as having properties that are exponential in the number of variables (e.g., §1.1, §7.1.3, §7.2). Although most of these situations have genuinely exponential behavior, "exponential" is actually an approximation, with exact details depending on the application.

If there are N variables and each variable i has V_i values, then the number of combinations is $V_1 \times V_2 \times \cdots \times V_N$. Since all variables have at least two values, the above expression is bounded below by 2^N. If all variables have the same number of values, then there are V^N possible combinations.

For common-casing, combinations are only formed using some fraction of the values for each variable. If each variable i has $1/F_i$ of the values used for common-casing then there are $V_1/F_1 \times V_2/F_2 \times \cdots \times V_N/F_N$ combinations. Thus, growth is exponential in the number of variables for which there is more than one common case. Where all variables have the same number of values used for common casing, the number of combinations grows as $(V/F)^N$.

Sometimes a variable may take on several values that have the same effect with respect to a given property. For example, bitblt line lengths are expressed in bits, while unrollings are expressed in words. Thus, left-aligned 52-bit and 53-bit lines use the same specialized code. In addition, combinations of values may have identical effect. For example, all values of `a` and `b` for which `a+b` has the value `53`. Thus, calculations of the number of cases should consider the number of distinct cases. Chapter 4.1 reports the number of distinct code cases for each application in the dissertation workload; for bitblt and debugger, the number of value combinations is much larger, but many combinations map to the same code cases.

B.2 Stability

Increasing the number of values used for specialization also increases the number of specialized implementations of a source code fragment. Thus, specialization sometimes leads to *instability*, which is when consecutive calls to a given routine use different implementations of the code. For example, making several copies of a rectangle using a generic bitblt uses just one code fragment. Performing the same set of copy actions with a specialized implementation of bitblt may use several code sequences, depending on the source and destination alignment with respect to memory and with respect to each other. Stability is analogous to the working set of a caching or paging scheme, but captures temporal effects of specialization. In particular, small code that is frequently recompiled can have a small working set and yet still suffer from instability and thus high recompilation rates.[1]

Stability depends on both the application and dataset. For example, a highly-specialized bitblt might have very good stability when called upon to tile a screen background with even-width rectangle, but very poor stability when called upon to build a mosaic screen background of random-sized tiles combined using randomly-selected combination operators.

Stability is related to the degree of specialization. One model of specialization is that all variables are concatenated to form a long bit string, as shown in Figure B.1. The bit string is then divided in two parts. The left part contains the most significant bits of the value and is used to select the specialized code case. The right part contains the least significant bits and is used as the parameter to the specialized code. Increasing the degree of specialization means moving the partition right, which effectively moves bits from the the "value" group to the "specialization" group.

Each additional bit that is used for specialization has the effect of doubling the number of specialized cases. This explains why the number of cases grows exponentially. Correspondingly, when there are two calls to a routine using similar parameters, increasing the degree of specialization also increases the likelihood that some of the bit values will be used to select a specialized code case. As a result, it becomes less and less likely that two similar calls will use the same code.

Instability affects both statically-optimized and dynamically-optimized code. For statically-specialized code, cache locality decreases, profiles become less accurate, individual cases become less important, and common-casing becomes less successful. For VSO, the process of selecting slow-changing variables is essentially a mechanism for ordering the parameters, so that stable values are used generate the VSO'd code, while the fast-changing values are used as parameters. Decreased stability makes it harder to select slow-changing variables and also means that the VSO'd code must be recompiled more frequently, thus reducing the benefit of VSO.

[1]Unfortunately, stability does not guarantee that these problems disappear, as stable code may still thrash in the cache, and so on.

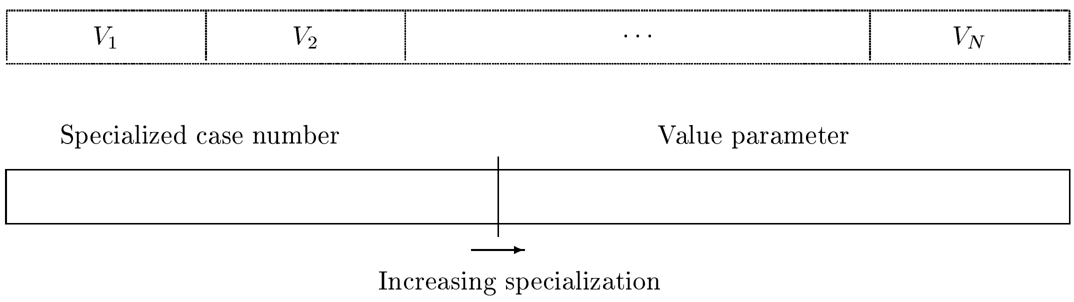


Figure B.1: A schematic view of the effects of specialization.

The values of the routine parameters are concatenated to form one large value. The left part of the value is used to select a particular specialized case, and the right part is used as a parameter to that specialized code. With increasing specialization, more and more bits are used to select the specialized case.

B.3 RTCG As A Cache

Many RTCG applications implicitly duplicate, or *caches*, the types and values of specialization variables. Changing the variables makes the code *incoherent*, or out-of-date with respect to the variables. These RTCG applications thus introduces two cache consistency problems. First, they require recompilation whenever a specialization variable changes. Although both hardware and software caching are commonplace, RTCG's caching is unusual in that a cache refill involves generating new code. Second, updating the code in memory also generally requires updating the code in the instruction caches, as discussed in Chapter 11.

Viewing code updating as a caching problem helps to clarify some of the problems and opportunities. For example, coherency in general may be divided into detecting a coherency problem and performing an operation to restore coherency. For RTCG, detection may take place anywhere between the time when the variables are updated and the time that the dependent code is invoked. Likewise, the coherency operation may be performed any time between detection and invocation. A coherency "transaction" may likewise be split into analysis and code generation.

The systems described in Part I all compile new code before every invocation and also discard the code on return (especially Section 2.4; see Figure 2.15). This policy corresponds approximately to flushing the instruction cache each time a VSO'd routine returns. However, there are a wide variety of other possible implementations.

For example, a *write update* protocol recompiles the code whenever a specialization variable is modified. Thus, instead of specializing on each call, the code is specialized each time it becomes incoherent. A *write invalidate* protocol sets a flag when any of the specialization variables change. Each call checks the flag, recompiling only if the flag has been set (see "Fast Conditionals, pg. 121).

Thus, instead of specializing each time the implementation becomes incoherent, it is specialized when the existing implementation is both incoherent and about to be used. These protocols show how splitting detection may improve its granularity. Where the RTCG'd code is invoked several times for each change of the specialization variables, improved granularity may reduce the total coherency cost.

B.4 Memoizing – Dynamic Common Casing

VSO'd code may be viewed as simply a function of the values of the specialization variables. If a given combination of values occurs frequently, it may be useful to *memoize*; that is, to retain several different versions of the code, each tagged with the values that are implicit in the fragment [May 87]. When the specialization variables return to values they have had before, code generation may be skipped and the existing memoized code can be invoked directly.

Memoizing implements dynamic common casing, which may be profitable when there are a few cases that are used commonly in one program invocation. In contrast, static common casing must produce cases that are common over the lifetime of the compiled program binary, which is much harder. For example, during one part of a program's execution, a given rectangle may be tiled with bitblt (§4.1.1), leading to heavy use of a particular copy width and a small number of alignments and transfer operations. The various cases can be a small percentage of the total number of bitblt cases ever used by the many invocations of a program binary, but may still be used heavily within a narrow part of one particular invocation of the program.

B.5 Simultaneous Specializations

Section 1.1 oversimplified and claimed that, although there are exponentially many specialized cases, "only one of them is in use at any given time." The cache simulator, however, is an example of an application that uses several VSO'd versions of a fragment at the same time (§4.1.2). The precise statement, then, is that there is one case in use at any given time per instance of the specialization variables. For cache simulation, there is one copy of the specialization variables per simulated cache, and thus one VSO'd code fragment per simulated cache.

B.6 Static Allocation and Code Placement

As described in Sections 3.3, 5.2.4 and 10.6.4, code placement substantially affects performance, and dynamically-generated code will not generally lie in the same place as its static counterpart (§3.3). However, some cases should allow better placement of dynamically-generated code than of its statically-generated counterpart.

Static allocation of dynamically-allocated code may at first seem difficult. However, RTCG'd code sizes are often predictable. For example, the applications used in this dissertation have code that is usually small, even compared to the size of small first-level instruction caches (Figure 5.5). In addition, most show only minor variations in code size depending on the values of the specialization variables. Static placement may be harder in the presence of multiple concurrent specializations (§B.5), but only the cache simulation application had that problem.

Static space allocation can reduce the runtime cost of VSO. Better, it also allows the static compiler to perform some instruction cache optimizations [McFarling 89, Gupta & Chi 90, McFarling 91, Torrellas et al. 92, Smotherman 94], even though the code itself is unavailable statically. In contrast, greater degrees of code static specialization increase the number of code cases, reduce the frequency with which any given case is used, and thus make static placement more difficult.

B.7 Static Code and Data Sizes

Section 5.2.4 shows that the VSO'd code itself is typically small. However, a potential disadvantage of VSO is that the application grows due to the added code and data that implement the application-specific compiler (including libraries called by the application-specific compiler), and these libraries all add to the working set. The extra code and data may cause increased paging, cache contention and so on.

In practice, template compilers and most parts of tree compilers are small. Both tree and template compilers mimic the structure of the *unspecialized* static code that they optimize, so their size is related to the size of the unspecialized static code. Although the application-specific compilers may call helper code libraries to copy and patch the IR, those routines are typically simple and thus small. For tree compilers, however, the size of the general-purpose code generator may dwarf the other code sizes. The effects of the code generator size are unclear.

The data sizes for the template and tree compilers are also typically small. Again, the data tends to mimic the static code that's being optimized. For the applications in this dissertation, simulation results show that the the application's bottleneck code has instruction and data cache misses that depend on the inner loop behavior, not on the compiler (§6.3.1). Note, however, that reducing VSO costs allows finer interleaving of compilation and execution, and thus increases the importance of compilation-related cache effects.

A traditional concern with self-modifying code is that the code cannot be shared between processes, and sharing is a good mechanism for reducing the effective size of a collection of programs. For VSO, the static code and constant data may be shared among concurrent invocations of a program. The only code that is truly private to a program invocation is that which is generated at application runtime. That code is typically quite small (Figure 5.5).

B.8 IR Inlining

Section 3.2.1 shows how templates can be inlined in a template compiler. Inlining is possible with any IR representation. For example, trees in a tree compiler can also be inlined as a collection of instruction immediates. To be profitable, however, the representation as code must be noticeably faster compared to the compilation cost. Thus, only small data structures and very fast compilers are likely to show benefit.

B.9 Deriving Templates Using Partial Evaluation

Section 2.3.2, describes how templates may be built by first building trees and then running the code generator over each tree, with some extra sophistication for holes and register allocation. An alternative view, which is more theoretically complete, is that the trees are built and then the code generator is partially evaluated with respect to each tree [Leone & Lee 93, Leone & Lee 94].

B.10 Specialization

Specialized code is not necessarily faster than unspecialized code. For example, both static and dynamic specialization may promote variables to constants in instruction immediates and thus improve register allocation and also eliminate loads. However, promotion is of little value where there were already sufficient registers and where the values were already loaded into registers. Specialized code may even be slower than unspecialized code, since larger code can enlarge working sets (§10.6.4, §B.2) and may lead to slower encodings (see Figure 5.1).

B.11 Inter-Domain Optimizations

This dissertation focuses on optimizations within a single domain, in particular where the whole application is available for analysis and where any part of the application may be reimplemented using VSO. It is also possible to optimize between domains, where "domain" encompasses module (abstraction) boundaries, binding time (linkage) boundaries, and protection (security) boundaries (§10.5.12). A problem with cross-domain optimizations, though, is that analysis and implementation tend to be more complicated.

For example, a client/server pair may be structured so that the client invokes the server through a library that is linked into the client. The server may set values in a shared memory segment that are then invariant over a loop in the client-linked server library. Optimization is more complex, however, because the client and server may bind the shared segment differently and because the server's writes to the segment may be invisible to the client. These problems are examples of a more general problem of maintaining abstractions (here, "protection" counts as an abstraction) while at the same time releasing implementation-level information needed to obtain better performance [Keppel 93b].

B.12 Meta-Problems

VSO defers some operations in order to take advantage of better runtime information. In deferring the operations, however, VSO introduces an additional level of indirection, which leads to a variety of *meta*-problems.

For example, staging may seem circular: it seems to say that VSO enables optimizations by moving them to the runtime and then makes them efficient by moving them back into the static compiler (§B.18). The real story, however, is that staging can move *parts* of the optimizer's execution back into the static compiler. The optimizer is still being invoked dynamically, and staging is simply used to reduce (though not eliminate) the cost of runtime optimization. Put another way, performing compiler operations at runtime introduces a level of indirection (extra runtime work), and staging is used to solve the *meta-problem* of making the new dynamic work run as fast as possible.

Likewise, where there are many candidate optimizers, VSO relies on static information in order to minimize the runtime cost of choosing the optimizations (§7.1.9). Thus, while better runtime information is the motivation for VSO, building runtime optimizers sometimes finds approximate static information to be more useful than exact dynamic information. The interplay is not entirely circular, because the static information needed for VSO is not the same as the runtime information needed for optimization. Nonetheless, the costs are related (there is some circularity), and the need for the static information is a necessary problem that is caused by providing runtime flexibility in implementation, while simultaneously trying to minimize the runtime cost.

B.13 Hybrids

Although static specialization and common-casing are preferable to VSO (§7.1.3), they are not always possible, due to space constraints. The choice, however, is not either/or: a system may use both static specialization and RTCG optimizations by statically optimizing those cases that are demonstrably common while relying on RTCG to handle cases that might be common for some application runs. For example, the cache simulator could have been implemented so that the most likely combinations of associativity, line size and total cache size would be handled using fully-specialized static code, with uncommon cases implemented using VSO instead of general static code.

Also, VSO'd code should be backed by well-optimized static code, so that even cases below the breakeven point that aren't profitable with VSO will run as quickly as possible. For example, the VSO'd bitblt described in Section 4.1.1 uses heavily specialized code for cases that don't reach the breakeven point.

B.14 Errors

The preceding chapters ignore many kinds of errors. First, eliminating some operations is incorrect [Auslander et al. 95, Auslander et al. 96]. For example, eliminating `x = x+0` at first appears to be a safe optimization, but may generally cause exception, for example, when adding zero to an undefined numeric value. Second, moving the binding time of operations may move the appearance of related errors. The application-specific compiler is constructed so that compiler errors do not appear at runtime and runtime errors do not appear at compile time. However, errors may move from their original position in the loop body, appearing, for example, in the application-specific compiler.

B.15 Stores vs. Changing Values

Section 7.1.2 describes analysis that identifies reuse rates based on stores to candidate variables. However, some stores do not change the value and thus do not require recompilation of the VSO'd code. Where possible, both candidate analysis (§7.1.2) and specialization should differentiate writes that leave the value unchanged. Distinguishing such cases can reduce recompilation rates. Identifying such cases during analysis is important, because cases with low reuse rates will dropped from further consideration (§7.1.9).

A related issue is that these analyses are based on the rate of change of the value of a variable; in practice, however, dynamic optimization is also effective where the value changes frequently but takes on a limited number of values (§B.4).

B.16 Stores Inside Loops

The preceding chapters simplified analysis by ignoring variables that are changed inside of loops (especially §7.1.2). There are at least three challenges to recognizing specialization variables that change inside of a loop. First, it is difficult to discover such candidates without introducing a large number of unprofitable candidates; such candidates can explode static compilation costs, even if they are ultimately be discarded (§7.2). Second, it must be possible to determine cheaply at runtime when the specialization variables change; in contrast, it is simpler to check variables set outside of loops. Third, the runtime compiler and runtime system must be able to move control between implementations; while less complex than general control flow problems [Shub 90, Hölzle et al. 92, Theimer & Hayes 92], it may still be necessary to stop a loop, compile a new implementation, and then restart the new implementation in the middle of the loop.

B.17 Rejected Workloads

The rejected workloads in Section 4.1.8 have inner loops that are difficult to optimize using VSO because the static code already performs close to the minimum work per input character. For Boyer-Moore string matching, prototypical static inner loop code is shown in Figure B.2. A VSO'd version of the same loop could be created by partially evaluating the loop with respect each entry of the table `tbl`. The VSO'd version would eliminate the conditional branch `while (skip > 0)`, but at the expense of making the lookup `tbl[c]` into a scaled index which, on many machines, is more expensive than an unsacled (byte) index. Likewise, the static code inner loop for regular expression matching, shown in Figure B.3, could be VSO'd by partially evaluating with respect to `state`, which would eliminate the inner loop conditional. However, the VSO'd version would replicate the inner loop 256 times per state or require double-indirection for state transition, and (to support incremental construction) a VSO'd version would also require a mechanism for patching the already-generated code whenever a new DFA state was created.

```
do{
   do{
      c = *in              /* Unrolled iteration #1 */
      skip = tbl[c]
      in += skip
      c = *in              /* Unrolled iteration #2 */
      skip = tbl[c]
      in += skip
      c = *in              /* Unrolled iteration #3 */
      skip = tbl[c]
      in += skip
   } while (skip > 0)

   ... check rest of string for match, or buffer more input ...

}while (not match)
```

Figure B.2: Static code inner loop for Boyer-Moore string searching.

The variable in is used to read from the input buffer, which is terminated with a sentinel. The current input character is used to index a skip table, and the input pointer is advanced by the skip distance. On a (partial) match or end of input, the skip distance is zero, so the loop may be unrolled correctly; since partial matches are typically rare, unrolling is also typically profitable. A typical iteration is: load, add, load, add (load, load, add on a machine with register+register addressing) plus the amortized cost of a conditional branch on nonzero.

```
newstate = build_state (dfacache, nfa, 0)   /* Generate start state. */
do {
   do {
      state = newstate                      /* Remember current state if miss, etc. */
      c = *in                               /* Read input .. */
      in += 1                               /* .. and advance input pointer. */
      newstate = state[c]                   /* Transition to new (DFA) state. */
   } while (newstate)                       /* While next state is cached. */

   ... check for final state, state cache miss, or buffer more input ...

} while (newstate)                          /* Until final state match or end-of-input. */
```

Figure B.3: Inner loop for incrementally-built DFA regular expression matching.

DFA states are built incrementally using an NFA description of a regular expression. The inner loop reads an input character and indexes the current state to transition to the next state. The same encoding is used for all three loop exit conditions: final state (match), end of input and DFA state cache miss. A typical inner loop iteration is: move, load, add, shift, add, load, and conditional branch on nonzero; the shift is required to scale the add.

B.18 Staging

Dynamic execution is flexible, but less efficient than static execution. The idea of *staging* a computation is to determine which parts of the computation are needed for flexibility, defer them to runtime, and then execute the remaining parts statically.[2] Computations that can be staged effectively can thus have both good flexibility and good performance. Staging is used in Section 7.1 to minimize the runtime work required to select whether static or VSO'd code will produce the fastest code.

Although staging and special-casing may appear similar, they are actually quite different. In particular, staging is used in VSO to move code from the compiler's optimizer into the program's runtime. In contrast, special-casing (§2.1.3) simply reorders computations from the original program. It may change the form of the final code but all computations are still derived from the original program.

Staging applies to VSO profitability analysis because performing all VSO cost analysis dynamically is generally too expensive, while performing all analysis statically is too restrictive, because it would only allow VSO to be applied where the breakeven point is zero or heuristically is "always" profitable. The goal, then, is to stage the cost analysis so that a simple runtime computation, such as "$x > 20$" will summarize the effects of more sophisticated static analysis.

[2]Staging is a particular instance of a broader class of binding time decisions.

Vita

David Keppel has a 1991 Master of Science degree from the University of Washington.