Shade: A Fast Instruction-Set Simulator for Execution Profiling

Bob Cmelik
Sun Microsystems, Inc.
`rfc@eng.sun.com`

David Keppel
University of Washington
`pardo@cs.washington.edu`

Abstract

Tracing tools are used widely to help analyze, design, and tune both hardware and software systems. This paper describes a tool called Shade which combines efficient instruction-set simulation with a flexible, extensible trace generation capability. Efficiency is achieved by dynamically compiling and caching code to simulate and trace the application program. The user may control the extent of tracing in a variety of ways; arbitrarily detailed application state information may be collected during the simulation, but tracing less translates directly into greater efficiency. Current Shade implementations run on SPARC systems and simulate the SPARC (Versions 8 and 9) and MIPS I instruction sets. This paper describes the capabilities, design, implementation, and performance of Shade, and discusses instruction set emulation in general.

1. Introduction

Tracing tools are used widely to help in the analysis, design, and tuning of both hardware and software systems. Tracing tools can provide detailed information about the behavior of a program; that information is used to drive an *analyzer* that analyzes or predicts the behavior of a particular system component. That, in turn, provides feedback that is used to improve the design and implementation of everything from architectures to compilers to applications. Analyzers can consume many kinds of trace information. For example, address traces are used for studies of memory hierarchies, register and operand usage for pipeline design, instruction combinations for superscalar and deep-pipe designs, instruction counts for optimization studies, operand values for memoizing studies, and branch behavior for branch prediction.

Several features can improve the utility of a tracing tool. First, the tool should be easy to use and avoid dependencies on particular languages and compilers. Ideally it should also avoid potentially cumbersome preprocessing steps. Second, it should be able to trace a wide variety of applications including those that use signals, exceptions and dynamically-linked libraries. Third, trace generation should be fast, both so that traces can be recreated on demand, instead of being archived on bulk storage, and so that it is possible to study realistic workloads, since partial workloads may not provide representative information [BKW90]. Fourth, a tracing tool should provide arbitrarily detailed trace information so that it is useful for a wide variety of analyzers; in general, this means that it must be extensible [NG88] so that it can be programmed to collect specialized information. Finally, it should be possible to trace applications for machines that do not yet exist.

These features are often at odds with each other. For example, static cross-compilation can produce fast code, but purely static translators cannot simulate and trace all details of dynamically-linked code. Also, improved tracing flexibility generally means reduced performance. An interpreter that saves address trace information may be reasonably fast, but adding control over whether the interpreter saves an address trace will slow the simulation, if at every instruction the simulator must check whether to save trace information. Providing finer control over where to save trace data slows simulation even more; adding the flexibility to save other kinds of trace information slows simulation yet further.

Because of the conflict between generality and performance, most tools provide only a subset of the features listed above. Shade provides the features together in one tool and uses five general techniques to achieve the needed flexibility and performance. First, Shade dynamically cross-compiles executable code for the target machine into executable code that runs directly on the host machine. Second, the host code is cached for reuse so that the cost of cross-compiling can be amortized. Third, simulation and tracing code are integrated so that the host code saves trace information directly as it runs. Fourth, Shade gives the analyzer detailed control over what is traced: the tracing strategy can be varied dynamically by opcode and address range. Shade then saves just the information requested by the analyzer, so clients that need little trace information pay little overhead. Finally, Shade can call special-purpose, analyzer-supplied code to extend Shade's default data collection capabilities.

This paper makes several contributions. We introduce dynamic compilation and caching techniques used for building fast cross-architecture simulators. We show how a tracing tool can be made extensible, and thus more flexible. Finally, we show how simulation and instrumentation code can be integrated to save trace information efficiently. We show these ideas using Shade, which performs cross-architecture simulation, collects many kinds of trace information, allows fine control over the tracing, is extensible, which simulates and traces the target machine in detail (including tricky things like signals and self-modifying code), and which, despite all of the above flexibility, has performance competitive with tools that just cross-simulate without tracing, with tools that do only simple tracing, and even with those that omit details to improve simulation and tracing efficiency. Thus, Shade shows that a general-purpose tool can be efficient enough to effectively replace many other tools. This paper also presents a framework for describing simulation and tracing tools.

To appear in the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.

The remainder of this paper is organized as follows: Section 2 describes the interface seen by programmers who use Shade to write analyzers. Section 3 describes the implementation of Shade, focusing on compilation, caching and instrumentation. Section 4 discusses cross-architecture simulation. Section 5 reports on the performance of Shade both for native and cross-architecture tracing. Section 6 compares the capabilities and implementation of other simulation and tracing tools.

2. Analyzer Interface

A Shade *analyzer* is a program (or that part of a program) which utilizes the simulation and, to varying degrees, the tracing capabilities provided by Shade. Shade analyzers have been used for pure simulation (no tracing), to generate memory address traces for use by other tools, provide a debugger interface to a simulated target machine for compiler cross-development [Evans92], observe instruction operand values [Richardson92], analyze memory cache performance, analyze microprocessor pipeline performance, and analyze Shade's own performance.

Analyzers see Shade as a collection of library functions [Cmelik93]. Analyzers call these functions to identify the application program to be simulated, specify the level of tracing detail, and to simulate one or more application instructions while collecting the specified trace information.

Shade "knows" how to efficiently collect common trace information such as the instruction address and text, data addresses for memory operations, and the contents of registers used by an instruction. Other information may be collected by analyzer-supplied trace functions. Shade arranges for these functions to be called before and/or after simulating an application instruction. The functions have access to the application's simulated registers and memory.

The analyzer may specify what trace information to collect and what trace functions to call on a per-opcode or per-instruction-address basis. So, for example, an analyzer which wishes to analyze memory systems might request tracing of just instruction and data addresses. Tracing selections may change during the course of the simulation. Thus, an analyzer can skip tracing during application initialization, or can trace only in particularly interesting application or library code. The less trace data the analyzer requests, the faster Shade runs.

3. Implementation

This section describes the basic implementation of Shade. Section 3.1 first describes the overall structure of Shade. Section 3.2 describes dynamic compilation of *translations* that directly simulate and trace the application program. Section 3.3 describes how translations are cached for reuse to reduce compilation overhead. Finally, Section 3.4 concludes with some special problems and considerations and the general techniques used in Shade.

3.1. Simulating and Tracing

The heart of Shade is a small main loop that repeatedly maps the current target (application) PC to a corresponding fragment of Shade host (simulator) code, called a *translation*. Each translation simulates the target instruction, optionally saves trace data, updates the target PC and returns to the main loop. Shade builds translations by cross-compiling target instructions into host machine code. Shade translates application memory references to refer to simulated memory and, similarly, translates updates of target registers into updates of simulated registers. Figure 1 summarizes the primary data structures used by Shade.

The main loop, translations, and most utility functions called by translations all share a common register window and stack frame. Several host registers are reserved for special purposes. Register `vs` is a pointer to the application's *virtual state* structure which holds the simulated registers; `vpc` is the application's virtual program counter (this is part of the virtual state, but is used enough to warrant its own host register); `vmem` is the base address of the application's memory; `tr` is a pointer to the current trace buffer entry; `ntr` is the number of unused trace buffer entries; and `tlb` is a pointer to the TLB (described below).

Shade maps the target PC to its corresponding translation using a data structure called the Translation Lookaside Buffer (TLB). The main loop does a fast, partial TLB lookup. If that fails, a function is called to do a slower, full TLB lookup. If that fails, the translation compiler is invoked to generate a new translation in the Translation Cache (TC) and update the TLB.

The main loop also checks for pending signals that need to be delivered to the application. Depending on how the application wishes to handle the signal, Shade may terminate the application at this point, or arrange for invocation of an application signal handler. In the latter case, Shade continues simulating and tracing, but now in the application's signal handler.

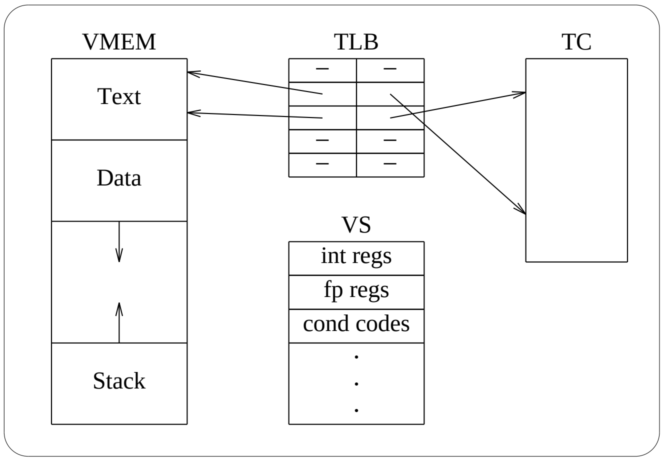

Figure 1. Shade data structures (not to scale)

3.2. Translations

Application instructions are typically translated in chunks which extend from the current instruction through the next control transfer instruction and accompanying delay slot. Translation also stops at tricky instructions such as software trap and memory synchronization instructions and Shade arbitrarily limits the number of application instructions per translation in order to simplify storage allocation. The user's trace buffer size also limits translation size. Therefore, a translation may represent more or less than one basic block of application code, and one fragment of application code may be simultaneously represented by more than one translation. Each translation consists of a prologue, a body with a fragment for each application instruction, and an epilogue.

3.2.1. Translation Prologue

The translation prologue (see Figure 2) allocates trace buffer space for the translation. If there is not enough space, the translation returns control to the main loop, which then returns control to the analyzer. Prologues are generated only for translations that collect trace information for at least one target instruction.

The trace space requirements for each translation could be stored in a data structure and tested by the main loop. That would save the code space now used for translation prologues, but would require executing additional instructions to address and load *count*, and would be inconsistent with translation chaining (described below) in which translations branch directly to each other, bypassing the main simulator loop.

```
prologue:
    subcc   %ntr, count, %ntr
    bgeu    body    ! if enough space, run body
    nop             ! (branch delay slot)
    add     %ntr, count, %ntr
    return to main loop
body:
```

Figure 2. Translation prologue

3.2.2. Translation Body

The translation body contains code to simulate and (optionally) trace application instructions. Simulation consists of updating the virtual state (registers plus memory) of the application program. Tracing consists of filling in the current trace buffer entry and advancing to the next.

Figure 3 shows a sample application instruction, and Figure 4 shows code that simulates it. The translation body first loads the contents of application registers `r1` and `r2` from the application's virtual state structure into host scratch registers `s1` and `s2`. Next, the translation performs the add operation. Then, the translation writes the result in host scratch register `s3` back to the virtual state structure location for application register `r3`. Finally, the translation updates the application's virtual PC.

```
add     %r1, %r2, %r3
```

Figure 3. Sample application code

```
ld      [%vs + vs_r1], %s1
ld      [%vs + vs_r2], %s2
add     %s1, %s2, %s3
st      %s3, [%vs + vs_r3]
inc     4, %vpc
```

Figure 4. Translation body (no tracing)

The code that is generated to actually perform the application operation is very often one and the same instruction, but with different register numbers. Where the host machine is a poor match to the virtual target machine, or where we wish to virtualize the target machine operations, several instructions, or even a call to a simulation function may be used. At the other extreme, no instructions need be generated to simulate useless application instructions (e.g. `nop`).

Shade allocates host registers to represent target registers; allocation is on a per-translation basis and can thus span several target instructions. The host registers hold target register values from one translated application instruction to the next in order to reduce memory traffic to and from the virtual state structure. Host registers are lazily loaded from the virtual state structure, then later lazily stored back, but no later than the translation epilogue.

Conceptually, Shade updates the virtual PC for each application instruction, as shown here. In practice, the virtual PC is only updated in the translation epilogue, or as needed in the translation body for tracing application instruction addresses.

For application instructions that access memory, Shade translates the application memory address to a host memory address by adding a *base address* offset (which applies for all application memory).

3.2.3. Tracing

Shade minimizes the amount of tracing code by giving analyzers precise control over which application instructions should be traced and what information should be collected for each instruction. For example, if the analyzer requests tracing for only data memory addresses from load instructions in a particular library (an address range), then Shade translates the library's load instructions to directly save the memory address in the trace record. No other trace information is saved for load instructions, and no trace information is saved for other instructions or for load instructions outside of the library.

Shade compiles the simulation and tracing code together. For example, Figure 5 shows code that simulates the sample application code, and, under analyzer control, traces the instruction address, instruction text, source and destination registers, and calls both pre- and post-instruction trace functions supplied by the analyzer. Whenever a translation calls an analyzer-supplied trace function, it first returns live application state to the virtual state structure for use by the trace function.

```
st      %vpc, [%tr + tr_pc]  ! trace instr addr
set     0x86004002, %o0
st      %o0, [%tr + tr_iw]   ! trace instr text
ld      [%vs + vs_r1], %s1   ! load 1st src reg
ld      [%vs + vs_r2], %s2   ! load 2nd src reg
st      %s1, [%tr + tr_rs1]  ! trace 1st src reg
st      %s2, [%tr + tr_rs2]  ! trace 2nd src reg

mov     %tr, %o0             ! arg1: trace buf
mov     %vs, %o1             ! arg2: virt. state
call pre-instruction trace function

add     %s1, %s2, %s3        ! simulate add
st      %s3, [%vs + vs_r3]   ! save dst reg
st      %s3, [%tr + tr_rd]   ! trace dst reg

mov     %tr, %o0             ! arg1: trace buf
mov     %vs, %o1             ! arg2: virt. state
call post-instruction trace function

inc     4, %vpc
inc     trsize, %tr  ! advance in trace buffer
```

Figure 5. Translation body (some tracing)

3.2.4. Translation Epilogue

The translation epilogue (see Figure 6) updates the virtual state structure and returns control either to the main simulator loop or jumps directly to the next translation. The epilogue saves host registers that hold modified virtual register values. If the virtual condition codes have been modified, they too must be saved. The epilogue also updates the trace buffer registers `tr` and `ntr` if necessary. The virtual PC remains in a host register across translation calls. Upon leaving a translation, it contains the address of the next application instruction to be executed.

```
epilogue:
    update virtual state structure
    update virtual PC
    inc     count * trsize, %tr
    go to main loop or next translation
```

Figure 6. Translation epilogue

Often, the execution of one translation always dynamically follows that of another. The two translations, predecessor and successor, can be directly connected or *chained* to save a pass through the main simulator loop. The predecessor and successor can be compiled in any order. If the successor is compiled first, the predecessor is compiled to branch directly to the successor. If the predecessor is compiled first, then at the time the successor is compiled the predecessor's return to the main simulator loop is overwritten with a branch to the successor.

Translations for conditional branches are compiled with two separate exits instead of a single common exit, so that both legs may be chained. Translations for register indirect jumps and software traps (which might cause a control transfer) cannot be chained since the successor translation may vary.

3.3. Translation Caching

The translation cache (TC) is the memory where translations are stored. Shade simply compiles translations into the TC one after the other, and the translation lookaside buffer (TLB) associates application code addresses with the corresponding translations.

When more TC space is needed than is available, Shade frees all entries in the TC and clears the TLB. Full flushing is used because translation chaining makes most other freeing strategies tedious [CK93]. Since full flushing deletes useful translations, the TC is made large so that freeing is rare [CK93]. Shade also flushes the TC and TLB when the analyzer changes the tracing strategy (typically rare), since tracing is hardcoded into the translations.

If an application uses self-modifying code, the TC, TLB, and translation chaining entries for the modified code become invalid and must be flushed. SPARC systems provide the `flush` instruction to identify code that has changed; many other systems provide equivalent primitives [Keppel91]. When the application executes the modified instructions, Shade compiles new translations for the changed code.

The TLB is an array of lists of <*target*, *host*> address pairs. Each pair associates an application instruction address with the corresponding translation address. To find a translation, Shade hashes the `vpc` to produce a TLB array index, then searches this TLB entry (address pair list) for the given application address. If the search succeeds, the list is reorganized so that the most recently accessed address pair is at the head of the list. If the search fails, a translation is generated, and a new address pair is placed at the head of the list.

Lists are actually implemented as fixed length arrays, which makes the TLB simply a two-dimensional array of address pairs. The TLB may also be thought of as N-way set associative, where N is the list length. Since address pair lists are of fixed length, address pairs can be pushed off the end of a list and lost, which makes the corresponding translations inaccessible via the TLB. The TLB is large enough that this is not usually a problem [CK93] and translations are also likely to still be accessible via chaining from other translations.

3.4. Other Considerations

The decision to simulate, trace, and analyze all in the same process leads to conflicts over the use of per-process state and resources. Conflicts arise between the application program (e.g. the code generated by Shade to simulate the application), the analyzer, and Shade (translation compiler, etc.). The conflicts are resolved in various ways. For example, the host's memory is partitioned so that Shade uses one part of the memory, and the application another. Resource conflicts can also arise from sharing outside of the process. For example, Shade and the application use the same file system so files written by one can accidentally clobber files written by the other. In general, conflicts are resolved by partitioning the resource, by time multiplexing it between contenders, or by simulating (virtualizing) the resource. Some conflicts are unresolved, usually due to an incomplete implementation [CK93].

Shade's target code parser is ad hoc, though machine code parsers can be built automatically [Ramsey93]. Shade uses an ad hoc code generator which generates code in roughly one pass. Some minor backpatching is later performed to chain translations and replace `nop`s in delay slots. The resulting code could no doubt be improved, but the time spent in the user-supplied analyzer usually dwarfs the time spent in Shade's code generation, simulation, and tracing combined.

Many of the implementation issues and choices, as well as some of the implementation alternatives, are described elsewhere [CK93], as are details of the signal and exception handling and implementation of the system call interface.

4. Cross Shades

In the previous section we focused on the Shade (subsequently referred to as Shade-V8.V8) for which the host and target architectures were both Version 8 SPARC, and for which the host and target operating systems were both SunOS 4.x [SunOS4]. Other Shades have been developed. The first (Shade-MIPS.V8) runs UMIPS-V [UMIPSV], MIPS I [Kane87] binaries, and the second (Shade-V9.V8) runs SunOS 4.x, Version 9 SPARC [SPARC9] binaries. The host system for both is SunOS 4.x, Version 8 SPARC. There are also versions of Shade-V8.V8 and Shade-V9.V8 where both the host and target operating systems are Solaris 2.x [SunOS5]. All of these Shades are at least complete to the extent that they can run SPEC89 binaries compiled for the respective target systems.

4.1. Shade-MIPS.V8

Shade-MIPS.V8 provides Shade's custom tracing capabilities for MIPS binaries. Given Shade-V8.V8 and ready access to SPARC systems, SPARC was the natural choice for the host architecture. As a rule, MIPS instructions are straightforward to simulate with just a few SPARC instructions. This is possible because both the MIPS and SPARC architectures are RISC architectures, both support IEEE arithmetic, and the MIPS architecture lacks integer condition codes.

Little attention was paid to simulation efficiency, beyond the efficient simulation techniques already used in Shade. On average,[1] Shade-MIPS.V8 executes about 10 SPARC instructions to simulate a MIPS instruction.

Some differences between the host and target machines make Shade-MIPS.V8 less faithful, slower, or more complicated. For example, MIPS systems support both big-endian and little-endian byte ordering [James90], but V8 SPARC only supports the former. Shade-MIPS.V8 currently runs only code that has been compiled for MIPS systems running in big-endian mode. Shade thus avoids the more complicated simulation of little-endian access. Similarly, Shade-MIPS.V8 does not check for overflows that would cause exceptions on MIPS systems. Several MIPS features such as unaligned memory access instructions and details of floating-point rounding have no direct V8 SPARC counterparts, so Shade-MIPS.V8 simulates them, albeit more slowly. Many immediate fields are 16 bits on the MIPS and 13 bits on the SPARC; where target immediates do not fit in 13 bits, extra SPARC instructions are used to place the immediate value in a host scratch register. This difference complicates the translation compiler.

Some host/target differences help Shade-MIPS.V8's efficiency. In particular, the MIPS architecture employs values stored in general purpose integer registers in place of integer condition codes. This reduces contention for the host condition codes [CK93].

4.2. Shade-V9.V8

Shade-V9.V8 simulates a V9 SPARC target and runs on a V8 SPARC host. The principal problems of simulating V9 applications on V8 hosts are wider integer registers and additional condi-

1. Here and elsewhere, "on average" means the geometric mean of dynamically weighted values over the SPEC89 benchmarks.

Shade	app	native		icount0		icount1		icount2		icount3		icount4		icount5	
		inst	time	inst	time	inst	time	inst	time	inst	time	inst	time	inst	time
V8.V8	gcc	1.0	1.0	5.5	6.1	5.9	6.6	8.8	14.3	13.5	21.7	15.5	31.2	63.7	84.2
	doduc	1.0	1.0	2.8	2.8	2.9	3.1	5.5	8.8	9.4	14.0	11.5	24.1	36.3	60.3
V9.V8	espresso	1.0	NA	9.5	1.2K	9.8	1.2K	11.5	2.2K	15.8	3.0K	17.8	4.8K	42.0	8.5K
	doduc	1.0	NA	6.1	1.1K	6.3	1.2K	8.1	2.4K	11.8	3.3K	13.9	5.4K	38.5	11.5K

Table 1. Dynamic expansion: instructions and CPU time

tion codes. Simulating a 64-bit address space would be a problem, but so far it has been avoided.

The new V9 instructions present few new problems, but there are many new instructions. As a rough measure of relative *simulation* complexity, consider that, given Shade-V8.V8, it took about 3 weeks to develop Shade-MIPS.V8 and about 3 months to develop Shade-V9.V8 to the point where each could run SPEC89.

Shade usually generates a short sequence of V8 instructions for each V9 instruction. For example, Figure 7 shows the translation body fragment for a V9 `add` instruction. Complicated instructions are compiled as calls to simulation functions.

```
ldd     [%vs + vs_r1], %s0 ! s0/s1: virt. r1
ldd     [%vs + vs_r2], %s2 ! s2/s3: virt. r2
addcc   %s1, %s3, %s5      ! add lower 32 bits
addx    %s0, %s2, %s4      ! add upper 32 bits
std     %s4, [%vs + vs_r3] ! virt. r3: s4/s5
inc     4, %vpc
```

Figure 7. Shade-V9.V8 translation body

The V9 target's 64-bit registers are simulated with register pairs on the V8 host. This doubles memory traffic for each register moved between the virtual state structure and the host registers. It also increases the number of such moves, since only half as many target registers can be cached in the host's registers.

V9 SPARC has two sets of condition codes. One set is based on the low order 32 bits of the result (just as in V8), and the other on the full 64 bits of the result. The host integer condition codes are often required (as in the `add` example above) to simulate 64-bit operations which themselves do not involve condition codes. This increases the number of contenders for the host condition codes [CK93].

Shade-V9.V8's performance is likely to degrade as compilers take advantage of more V9 features. For example, V9 supports more floating point registers and floating point condition codes than V8. V9 compilers that make better use of these registers will increase register pressure on the V8 host. Also, under Shade-V9.V8, applications are only allowed access to the lower 4GB of virtual memory. Thus, although programs manipulate 64-bit pointers, Shade-V9.V8 ignores the upper 32-bits of addresses during the actual accesses (load, store, register indirect jump, system call). Shade-V9.V8 will run slower if and when it needs to simulate a full 64-bit address space.

5. Performance

This section reports on the performance of Shade. For Shade-V8.V8, performance is reported relative to native execution. Since SPARC V9 platforms are still under construction, Shade-V9.V8 figures do not include relative performance. The standard Shade configuration used in these tests is a 4MB TC that holds 2^{20} host instructions, and a 256KB TLB that holds 2^{13} (8K) lines, each with 4 address pairs.

The benchmarks are from SPEC89, compiled with optimizations on. For Shade-V8.V8, the *001.gcc1.35* and *015.doduc* benchmarks were used; for Shade-V9.V8, *008.espresso* and *015.doduc* were used.

The measurements use six Shade analyzers, each performing a different amount of tracing. The analyzers use Shade to record varying amounts of information, but everything Shade records is then ignored. This "null analysis" was done to show the breakdown of time in Shade. With real analyzers, analysis dominates the run time and Shade is not the bottleneck. The analyzers are:

`icount0`: no tracing, just application simulation.

`icount1`: no tracing, just update the traced instruction counter (`ntr`) to permit instruction counting.

`icount2`: trace PC for all instructions (including annulled); trace effective memory address for non-annulled loads and stores. This corresponds to the tracing required for cache simulation.

`icount3`: same as `icount2` plus instruction text, decoded opcode value, and, where appropriate, annulled instruction flag and taken branch flag.

`icount4`: same as `icount3` plus values of all integer and floating point registers used in instruction.

`icount5`: same as `icount4` plus call an empty user trace function before and after each application instruction.

Table 1 shows how much slower applications run under Shade compared to native execution. The *inst* column shows the average number of instructions that were executed per application instruction. The *time* column shows the CPU (user + system) time; for Shade-V8.V8 as a ratio to native time, for Shade-V9.V8 as absolute time in seconds.[2]

Shade is usually more efficient on floating-point code (*doduc*) than on integer code (*gcc* and *espresso*). Floating-point code has larger basic blocks, which both improves host register allocation and reduces the number of branches and thus the number of lookup operations to map the target PC to the corresponding translation. Floating-point code also uses more expensive operations, so relatively more time is spent doing useful work. The relative costs are closer for higher levels of tracing, since the overhead of tracing is nearly independent of the instruction type.

Shade-V9.V8 is less efficient than Shade-V8.V8, and less efficient for integer than floating point applications. The wider V9 words cause more memory traffic and more contention for host registers. V9 also has more condition codes and is thus more work to simulate. On average, Shade-V9.V8 simulates V8 (*sic*) integer SPEC89 benchmarks 12.2 times slower than they run native, and V8 floating point SPEC89 benchmarks 4.0 times slower. Shade-V8.V8 simulates these same benchmarks 6.2 and 2.3 times slower than they run native, respectively.

Table 2 shows how much larger dynamically (i.e. weighted by number of times executed) a translation is than the application code it represents. *Input size* is the dynamically-weighted average size of a target basic block. *Output size* is the dynamically-weighted average number of instructions in a translation and the

2. Instruction counts were gathered by running Shade on itself: the superior Shades ran the `icount1` analyzer while the subordinate (traced) Shades ran the indicated analyzers and benchmarks. Overall running times were collected using elapsed time timers on the host. Percentage time distribution (shown below) was measured using conventional profiling with `cc -p` and `prof`.

Shade	app	input size	output size					
			icount0	icount1	icount2	icount3	icount4	icount5
V8.V8	gcc	5.1	20 4.7x	26 6.2x	41 9.1x	67 15x	77 17x	193 40x
	doduc	12.5	33 4.1x	39 5.1x	73 8.0x	126 13x	153 15x	427 39x
V9.V8	espresso	6.1	44 8.2x	49 9.5x	61 11.6x	91 17x	104 19x	246 44x
	doduc	13.6	63 5.5x	69 6.4x	94 8.3x	147 13x	177 15x	432 37x

Table 2. Code translation expansion, dynamically weighted

Shade	app	location	icount0	icount1	icount2	icount3	icount4	icount5
V8.V8	gcc	Compiler	8.77%	10.14%	5.82%	4.59%	3.86%	25.05%
		TC	51.13%	52.56%	74.62%	82.22%	86.33%	61.26%
		Sim	39.00%	36.09%	19.01%	12.83%	9.55%	4.97%
		Analyzer	0.00%	0.03%	0.01%	0.00%	0.00%	8.61%
	doduc	Compiler	0.22%	0.35%	0.16%	0.11%	0.08%	0.06%
		TC	80.69%	81.56%	91.50%	95.20%	96.37%	87.87%
		Sim	19.04%	17.97%	8.32%	4.67%	3.55%	2.11%
		Analyzer	0.00%	0.07%	0.00%	0.00%	0.00%	9.96%
V9.V8	espresso	Compiler	0.30%	0.35%	0.23%	0.20%	0.13%	0.10%
		TC	61.92%	62.73%	78.98%	84.26%	89.90%	81.27%
		Sim	37.74%	36.84%	20.76%	15.52%	9.95%	5.43%
		Analyzer	0.00%	0.04%	0.00%	0.01%	0.00%	13.19%
	doduc	Compiler	0.11%	0.15%	0.09%	0.07%	0.05%	0.05%
		TC	61.92%	64.35%	80.55%	85.93%	90.65%	85.82%
		Sim	37.96%	35.47%	19.35%	14.00%	9.30%	4.77%
		Analyzer	0.00%	0.03%	0.00%	0.01%	0.00%	9.36%

Table 3. Run-time execution profile summary

Shade	app	icount0	icount1	icount2	icount3	icount4	icount5
V8.V8	gcc	179.7	171.25	127.2	94.0	87.5	51.4
	doduc	245.4	271.2	162.6	111.9	102.1	58.9
V9.V8	espresso	451.1	486.7	423.6	331.3	308.4	191.7
	doduc	123.5	151.4	123.7	91.9	84.2	84.6

Table 4. Code generator instructions per instruction generated

code space expansion over the input size. Output sizes don't directly correlate to running time, since portions of most translations are conditionally executed, and since some instructions are executed outside of the TC in the translation compiler, simulation functions, and the analyzer.

Table 3 shows the percentage of total run time spent in various phases of execution. *Compiler* denotes the time spent in the translation compiler, *TC* the time spent executing code in the Translation Cache, *Sim* the time spent in functions which are called from the TC to simulate, or assist in simulating application instructions, and *Analyzer* the time spent in the user's analyzer, including user trace functions which are called from the TC.

The time distribution is determined by several factors. Better optimization takes longer and produces faster running code, both of which increase the percentage of time spent in code generation. The simulation time (*Sim*) comes mostly from saving and restoring condition codes [CK93], simulating `save` and `restore`, and from main loop execution; larger target basic blocks tend to reduce condition code and main loop overheads. A small TC increases the frequency with which useful translations are discarded. A small or ineffective TLB increases the frequency with which useful translations are lost. Translations that collect a lot of information take longer to run, and thus reduce the percentage of time spent in simulation functions, even though their absolute running time is unchanged. All analyzers used in these tests are trivial, though `icount5` includes null functions that are called before and after each application instruction.

Table 4 shows the average number of instructions that are executed by the code generator in order to generate one host instruction. The number of instructions per instruction in the code generator is a function of the instruction set architecture of the host and target machines and the level of tracing. Note that without translation caching, the compiler would be invoked every time a target instruction was run and applications would run hundreds or thousands of times slower. Measurements of the TC and TLB effectiveness are reported elsewhere [CK93].

6. Related Work

This section describes related work and summarizes the capabilities and implementation techniques of other simulators, virtual machines and tracing tools. In most cases we try to evaluate the capabilities of each tools' technology, but as we are evaluating actual tools, we sometimes (necessarily) evaluate limits of a particular implementation.

6.1. Capabilities and Implementation

Table 5 summarizes the capabilities and implementations for a number of tools. The columns show particular features of each tool and are grouped in three sections. The first group, *Purpose* and *Input Rep.* describe the purpose of the tool and how a user prepares a program in order to use the tool. The second group of columns, *Detail*, *MD*, *MP*, *Signals* and *SMC OK*, shows the level of detail of the simulation, and thus the kinds of programs that can be processed by the tool. The third group of columns, *Technology* and *Bugs OK* shows the implementation technology used and the tool's robustness in the face of application errors. The columns are described in more detail below.

Purpose describes how the tool is used: for cross-architecture simulation (*sim*); debugging (*db*); for address tracing or memory hierarchy analysis (*atr*); or for other, more detailed kinds of tracing (*otr*). Tools marked tb_C are tool-building tools and usually use C as the extension language [NG88].

Input describes the input to the tool. Processing a high-level language input (*hll*) can have the best portability and best optimization but the tool can only be used for source programs written in the supported languages [VF94] and can't generally be used for

Name	Reference(s)	Purpose	Input Rep.	Detail	MD	MP	Signals	SMC OK	Technology	Bugs OK
Accelerator	[AS92]	sim	exe	us	Y	N	Y	Y	scc+gi	Y
ATOM	[SE94]	tb_C	exe*	u	N	N	Y	N	aug	N
ATUM	[ASH86]	sim/atr	exe	us	Y	$Y_=$	Y	Y	emu	Y
dis+mod+run	[FC88]	sim/atr	asm	u	N	N	N	N	scc	N
Dynascope	[Sosič92]	db/atr/otr	hll	u	N	N	S	Y	pdi	Y
Executor	[Hostetter93]	sim	exe	u	N	N	Y	Y	pdi	Y
g88	[Bedichek90]	sim/db	exe	usd	Y	N	Y	Y	tci	Y
gsim	[Magnusson93, Magnusson94]	sim/db/atr/otr/tb_C	exe	usd	Y	Y_1	Y	Y	tci+dcc	Y
Mable	[DLHH93]	sim/db/atr	exe	u	N	Y_1	N	Y	ddi	N
mg88	[Bedichek94]	sim/db/atr/otr/tb_C	exe	usd	Y	Y_1	Y	Y	tci	Y
Migrant	[SE93]	sim	exe	u	Y	N	Y	Y	scc+emu	Y
Mimic	[May87]	sim	exe	u	N	N	N	N	dcc	N
MINT	[VF94]	atr	exe	u	N	Y_1	Y	N	pdi+dcc	Y*
Moxie	[CHKW86]	sim	exe	u	N	N	Y	N	scc	N
MX/Vest	[SCKMR93]	sim	exe	u	N	$Y_=$	Y	Y	scc+gi	Y
Purify	[HJ92]	db	exe*	u	N	N	Y	N	aug	Y
qp/qpt	[LB94]	atr/otr	exe	u	N	N	N	N	aug	N
SELF	[CUL89]	sim	exe	u	N	N	Y	Y	dcc	Y
SoftPC	[Nielsen91]	sim	exe	u(s)d	N	N	Y	Y	dcc	Y
Spa	[Irlam93]	atr	exe	u	N	N	S	Y	ddi	N
SPIM	[HP93]	sim/atr	exe	u	N	N	Y	N	pdi	Y
ST-80	[DS84]	sim	exe	u	N	N	Y	Y	dcc	Y
MPtrace	[EKKL90]	atr	asm	u	N	$Y_=$	S	N	aug	N
Pixie	[MIPS86]	atr	exe*	u	Y	N	Y	N	aug	N
Pixie-II	[Killian94]	atr/otr/db	exe*	us	Y	N	Y	S	scc	N
Proteus	[BDCW91]	atr	hll	u	N	Y_1	N	S	aug	N
RPPT	[CMMJS88]	atr	hll	u	N	Y_1	N	N	aug	N
Titan	[BKW90]	atr	exe	us	Y	N	Y	N	aug	N
TRAPEDS	[SJF92]	atr	asm	us	Y	$Y_=$	S	N	aug	N
Tango Lite	[GH92]	atr	asm	u	N	Y_1	N	S	aug	N
WWT	[RHLLLW93]	atr/otr	exe	u	Y	Y_+	Y	N	emu+aug+ddi	Y
Z80MU	[Baumann86]	sim	exe	u(s)	N	N	Y	Y	ddi	Y
Shade	[CK93]	sim/atr/otr/tb_C	exe	u	N	N	Y	Y	dcc	N

Table 5. Summary of some related systems

studying the behavior of other translation tools (compilers, etc.). Consuming assembly code (*asm*) is less portable than a high-level language but can provide more detailed information. To the extent that assembly languages are similar, such tools may be relatively easy to retarget, though detailed information may still be obscured. Finally, using executable code as input (*exe*) frees the user from needing access to the source and the (possibly complex) build process. However, information is usually reported in machine units, not source constructs. Some tools use symbol table information to report trace information symbolically. Others also need symbolic information to perform translation (*exe*∗).

Detail describes how much of the machine is simulated. Most tools work with only user-level code (*u*); some also run system-level code (*s*); and system mode simulation generally requires device emulation (*d*). Some target machines have no system mode, so simulation can avoid the costs of address translation and protection checks; these machines have the system mode marked in parenthesis.

MD reports whether the tool supports multiple protection domains and multitasking (multiple processes per target processor). This usually implies support for system mode operation and address translation. Target systems that multitask in a single protection domain are listed as *N*. *MP* tells whether the tool supports multiple processor execution; Y_1 indicates that the tool uses a single host processor, $Y_=$ indicates that the tool runs as many target processors as host processors, Y_+ that it can run more target processors than host processors. Simulating a multiprocessor generally introduces additional slowdown at least as big as the number of target processors divided by the number of host processors.

Supporting signals is generally difficult since execution can be interrupted at any instruction and resumed at any other instruction, but analysis and instrumentation may use groups of instructions to improve simulation efficiency. The *Signals* column is *Y* for tools that can handle asynchronous and exceptional events. *S* indicates that the tool is able to deal with some but not all aspects; for example, signals may be processed so the program's results are correct, but no address trace information is generated.

SMC OK describes whether the tool is able to operate on programs where the instruction space changes dynamically. Dynamic linking is the most common reason, but there are a number of other uses [KEH91]. Static rewrite tools can sometimes (*S*) link dynamically to statically-rewritten code, but the dynamically-formed link can't be rewritten statically and thus may go untraced.

Technology describes the general implementation techniques used in the tool [Pittman87]. An "obvious" implementation executes programs by fetching, decoding, and then interpreting each instruction in isolation. Most of the implementations optimize by predecoding and then caching the decoded result; by translating to host code to make direct use of the host's prefetch and decode hardware [DS84]; and by executing target instructions in the context of their neighbors so that target state (e.g. simulated registers) can be accessed efficiently (e.g. from host registers) across target instruction boundaries. The implementations are:

- Hardware emulation including both dedicated hardware and microcode (*emu*).
- The "obvious" implementation, a decode and dispatch interpreter (*ddi*).
- Predecode interpreters (*pdi*) that pre-convert to a quick-to-decode intermediate representation. The IR can be many forms; a particularly fast, simple, and common form is threaded code (*tci*).

Name	Reference(s)	Translation Units	Assumptions	Performance (Slowdown)	Notes
Accelerator	[AS92]	ebb	nr, bo, ph, regs	3	pages
dis+mod+run	[FC88]	bb	nr	10	
Executor	[Hostetter93]	proc	nr	10	mixed code
`g88`	[Bedichek90]	i	nr, bo	30	pages
`gsim`	[Magnusson93, Magnusson94]	bb	nr, bo	30	pages
Mable	[DLHH93]	i		20-80	
`mg88`	[Bedichek94]	i	nr, bo	80	pages
Migrant	[SE93]	ebb	nr,bo	–	
Mimic	[May87]	ebb	nr, bo, regs	4	no fp, no align, +compile
Moxie	[CHKW86]	bb	nr	2	
MX/Vest	[SCKMR93]	ip	bo	2	mixed code, fp prec
SELF	[CUL89]	ip	none	N/A	VM spec
SoftPC	[Nielsen91]			10	
SPIM	[HP93]	i	nr, bo	25	
ST-80	[DS84]	proc	none	N/A	VM spec
Z80MU	[Baumann86]	i	nr, bo, regs	–	mixed code
Shade	[CK93]	ebb	nr, bo	3-6 8-15	same machine different machines (tracing off)

Table 6. Summary of some cross-architecture simulators

- Static cross-compilation (*scc*) which decodes *and dispatches* during cross-compilation, avoiding essentially all runtime dispatch costs. As a special case, where the host and target are the same, the static compiler merely annotates or *augments* (*aug*) the original program with code to save trace data or emulate missing instructions. Note that conversion is limited by what the tool can see statically. For example, dynamic linking may be hard to instrument statically. Limited static information also limits optimization. For example, a given instruction may *in practice* never be a branch target, but proving that is often hard, so the static compiler may be forced to produce overly-conservative code.
- Dynamic cross-compilation (*dcc*) is performed at runtime and thus can work with any code including dynamically-linked libraries. Also, dynamic cross-compilers can perform optimistic optimizations and recompile if the assumptions were too strong [Johnston79, SW79, May87, HCU91, CK93]. However, since the compiler is used at run time, translation must be fast enough that the improved performance more than pays for the overhead of dynamic compilation [KEH91]; in addition, code quality may be worse than that of a static cross-compiler [AS92, SCKMR93] since dynamic code analysis may need to "cut corners" in order to minimize the compiler's running time.

Where interpreter specifics are unavailable the tool is listed as using a general interpreter (*gi*). Many tools listed as *aug* and *emu* execute most instructions using host hardware.

Note that input forms lacking symbolic information — *exe* especially — can be hard to process statically because static tools have trouble determining what is code and what is data and also have trouble optimizing over multiple host instructions [May87, LB94]. By contrast, tools that perform dynamic analysis (including both interpreters and dynamic cross-compilers) can discover the program's structure during execution. Translation techniques can be mixed by using one technique optimistically for good performance and another as a fallback when the first fails. However, such implementations have added complexity because they rely on having two translators [AS92, SCKMR93, Magnusson94, VF94].

Bugs OK describes whether the tool is robust in the face of application errors such as memory addressing errors or divide-by-zero errors. Typically, a simulator that checks for addressing errors requires extra checks on every instruction that writes memory. In some systems the checks are simple range checks; tools that support multiple address spaces and sparse address spaces generally require full address translation [Bedichek90]. *Y** indicates that checking can be turned on but performance is worse.

6.2. Cross-Architecture Simulation

Table 6 summarizes various features of tools that are used for cross-architecture simulation. The *Translation Units* column shows translation-time tradeoffs between analysis complexity and performance. *Assumptions* shows assumptions about the relationship between the host and target machines; these assumptions are usually used to simplify and speed the simulator. *Performance* shows the approximate slowdown of each tool compared to native execution. *Notes* shows special or missing features of each simulator. The columns are described in detail below.

Translation units are the number of (target) instructions that are translated at a time. Using bigger chunks reduces dispatching costs and increases opportunities for optimization between target instructions. Larger translation units also typically require better analysis or dynamic flexibility in order to ensure that the program jumps always take a valid path [May87, SCKMR93, LB94]. Translation units include: individual instructions (*i*), basic blocks (*bb*), extended basic blocks with a single entry but many exits (*ebb*), procedural (*proc*), or interprocedural (*ip*).

Assumptions describes assumptions that a tool makes about the relationship between the host and target machines, including byte ordering (*bo*); numeric representation (*nr*), including size and format; the number of registers on the host and target machines (*regs*), and access to the host machine's privileged hardware (*ph*) in order to implement system-level simulation.

Performance is an **estimate** of the number of (simple) instructions executed per (simple) simulated instruction. *N/A* indicates "not applicable" because the target is a virtual machine. A dash (−) indicates unknown or unreported performance. These estimates are necessarily inexact because performance for the different tools is reported in different ways.

Notes describes particular features: *pages* for detailed memory simulation; *no fp* for simulation that omits floating-point numbers; *fp prec* for simulation that can be set either to run fast or to faithfully emulate the target machine; *no align* for tools that omit simulation of unaligned accesses; *+compile* for dynamic compilers where compile time is not included in the performance but where it would likely have a large effect; *VM spec* for tools that emulate a virtual machine that has been designed carefully to improve portability and simulation speed; *mixed code* for simulators that can call between host and target code so that the application can, e.g., dynamically link fast-running host-code libraries.

6.3. Comparison

Shade improves over many other tools by simulating important machine features such as signals and dynamic linking. It improves over many dynamic compilation tools by using techniques that reduce simulation overhead while maintaining the flexibility and code quality of dynamic compilation. It improves over many tracing tools by dynamically integrating cross-simulation and tracing code so that it can trace dynamically-linked code, can handle dynamic changes in tracing level, and yet can still save detailed trace information efficiently.

Most tools avoid cross-architecture execution or omit some machine features. These choices improve execution efficiency but limit the tool's applicability. Some exceptions are g88 derivatives [Bedichek90, Magnusson93, Bedichek94, Magnusson94] which are somewhat less efficient than Shade and also Accelerator [AS92] and MX/Vest [SCKMR93] which do not perform any tracing and which use two translators, one optimistic and one conservative, to achieve high efficiency. Shade supports cross-architecture execution, and faithfully executes important machine features such as signals and self-modifying code (and thus dynamic linking), so it can be used on a wide variety of applications.

Simulators that use dynamic compilation are typically flexible and the compiled code performs well. However, many previous systems have imposed limitations that Shade eliminates. For example, Mimic's compiler [May87] produces high-quality code, but at such an expense that overall performance is worse than Shade; Shade reduces compilation overhead by allowing multiple translations per application instruction, by using chaining to reduce the cost of branches, and using a TLB to minimize the space overhead of branches. MINT [VF94] is unable to simulate changing code and never reclaims space used by inactive translations.

Tracing tools typically produce only address traces, and often run only on the machine for which the trace is desired. Even tools that allow cross-architecture simulation tend to limit the generality of the machine simulation or of the tracing facilities in order to maintain efficiency [FC88, HP93]. Shade supports cross-architecture tracing and simulates user-mode operation in detail. It currently lacks kernel-mode tracing facilities provided by some other tools though some of these tools limit machine features and/or require hand-instrumentation of key kernel code. Shade collects more trace information than most other tools, though it lacks the timing-level simulation of mg88 [Bedichek94]. With Shade, the analyzer can select the amount of trace data that it collects, and analyzers that consume little trace data pay little tracing overhead. Thus, it is typically the analysis tools that limit overall performance.

Of the tool building tools listed, all permit extended tracing; Shade provides the most efficient yet variable extensibility, and only Shade also inlines common trace operations. Shade analyzers have used both C and C++ as the extension language [NG88]. We note also that although Shade is not designed for debugging, Shade-V9.V8 has been used as the back end of a debugger [Evans92].

Shade's flexibility and performance does come at a penalty. For example, Shade performs inter-instruction analysis and host code generation; this makes Shade more complex and less portable than, e.g., g88. Shade also presently lacks multiprocessor and kernel mode; supporting them would make Shade slower since they complicate simulation (e.g. with address translation on loads and stores) and would increase translated code size.

7. Conclusions

Shade is a custom trace generator that is both fast and flexible, providing the individual features of other tracing tools together in one tool. Shade achieves its flexibility by using dynamic compilation and caching, and by giving analyzers detailed control over data collection. Thus analyzers pay for only the data they use. Since Shade is fast, analyzers can recreate traces on demand instead of using large stored traces. Shade's speed also enables the collection and analysis of realistically long traces. Finally, Shade simulates many machine details including dynamic linking, asynchronous signals and synchronous exceptions. By providing a detailed simulation and by freeing the user from preprocessing steps that require source code and complicated build procedures, Shade satisfies a wide variety of analysis needs in a single tool.

8. Acknowledgements

Shade owes much to its predecessors, particularly its immediate predecessor Shadow, which was created by Peter Hsu [Hsu89]. Robert Cmelik developed Shade, with numerous suggestions from David Keppel. Steve Richardson, Malcolm Wing, and others in the Shade user community provided useful user interface feedback and helped debug Shade. Robert Bedichek, Alex Klaiber, Peter Magnusson and the anonymous SIGMETRICS referees gave helpful comments on earlier versions of this paper. Finally, authors of many of the systems in the related work section went out of their way to help us understand their tools; we apologize for errors and omissions. This work was supported by Sun Microsystems, NSF #CDA-8619-663 and NSF PYI #MIP-9058-439.

References

[AS92] Kristy Andrews and Duane Sand, "Migrating a CISC Computer Family onto RISC via Object Code Translation," *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, 213-222, Oct. 1992.

[ASH86] Anant Agarwal, Richard L. Sites, and Mark Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode," *Proc. of the 13th International Symposium on Computer Architecture*, 119-127, Jun. 1986.

[Baumann86] Robert A. Baumann, "Z80MU," *Byte*, 203-216, Oct. 1986.

[BDCW91] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl, "PROTEUS: A High-Performance Parallel-Architecture Simulator," MIT/LCS/TR-516, Massachusetts Institute of Technology, 1991.

[Bedichek90] Robert Bedichek, "Some Efficient Architecture Simulation Techniques," *Winter 1990 USENIX Conference*, Jan. 1990.

[Bedichek94] Robert Bedichek, "The Meerkat Multicomputer: Tradeoffs in Multicomputer Architecture," Doctoral Dissertation, University of Washington Department of Comp. Sci. and Eng., 1994 (in preparation).

[BKW90] Anita Borg, R. E. Kessler, and David W. Wall, "Generation and Analysis of Very Long Address Traces," *Proc. of the 17th Annual Symposium on Computer Architecture*, 270-279, May 1990.

[CHKW86] F. Chow, M. Himelstein, E. Killian, and L. Weber, "Engineering a RISC Compiler System," *IEEE COMPCON*, Mar. 1986.

[CK93] Robert F. Cmelik and David Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," SMLI 93-12, UWCSE 93-06-06, Sun Microsystems Laboratories, Inc., and the University of Washington, 1993.

[Cmelik93] Robert F. Cmelik, *The Shade User's Manual*, Sun Microsystems Laboratories, Inc., Feb. 1993.

[CMMJS88] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, "The Rice Parallel Processing Testbed," *ACM SIGMETRICS*, 4-11, 1988.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee, "An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes," *OOPSLA '89 Proceedings*, 49-70, Oct. 1989.

[DLHH93] Peter Davies, Philippe LaCroute, John Heinlein, and Mark Horowitz, "Mable: A Technique for Efficient Machine Simulation," (to appear), Quantum Effect Design, Inc., and Stanford University.

[DS84] Peter Deutsch and Alan M. Schiffman, "Efficient Implementation of the Smalltalk-80 System," *11th Annual Symposium on Principles of Programming Languages*, 297-302, Jan. 1984.

[EKKL90] Susan J. Eggers, David Keppel, Eric J. Koldinger, and Henry M. Levy, "Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor," *ACM SIGMETRICS*, 37-47, May 1990.

[Evans92] Doug Evans, *Personal comm.*, Dec. 1992.

[FC88] Richard M. Fujimoto and William B. Campbell, "Efficient Instruction Level Simulation of Computers," *Transactions of The Society for Computer Simulation,* 5(2): 109-124, 1988.

[GH92] Stephen R. Goldschmidt and John L. Hennessy, "The Accuracy of Trace-Driven Simulations of Multiprocessors," CSL-TR-92-546, Stanford University Computer Systems Laboratory, Sep. 1992.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar, "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches," *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, Jul. 1991.

[HJ92] Reed Hastings and Bob Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," *Proc. of the Winter Usenix Conference*, 125-136, Jan. 1992.

[Hostetter93] Mat Hostetter, *Personal comm.*, Jul. 1993.

[HP93] John Hennessy and David Patterson, *Computer Organization and Design: The Hardware-Software Interface* (Appendix A, by James R. Larus), Morgan Kaufman, 1993.

[Hsu89] Peter Hsu, *Introduction to Shadow,* Sun Microsystems, Inc., 28 Jul. 1989.

[Irlam93] Gordon Irlam, *Personal comm.*, Feb. 1993.

[James90] David James, "Multiplexed Busses: The Endian Wars Continue," *IEEE Micro Magazine*, 9-22, Jun. 1990.

[Johnston79] Ronald L. Johnston, "The Dynamic Incremental Compiler of APL\3000," *APL Quote Quad*, 9(4): 82-87, Association for Computing Machinery (ACM), Jun. 1979.

[Kane87] Gerry Kane, *MIPS R2000 RISC Architecture,* Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[KEH91] David Keppel, Susan J. Eggers, and Robert R. Henry, "A Case for Runtime Code Generation," University of Washington Comp. Sci. and Eng. UWCSE TR 91-11-04, Nov. 1991.

[Keppel91] David Keppel, "A Portable Interface for On-The-Fly Instruction Space Modification," *Proc. of the 1991 Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, 86-95, Apr. 1991.

[Killian94] Earl Killian, *Personal comm.*, Feb. 1994.

[LB94] James R. Larus and Thomas Ball, "Rewriting Executable Files to Measure Program Behavior," *Software – Practice and Experience*, 24(2): 197-218, Feb. 1994.

[Magnusson93] Peter S. Magnusson, "A Design For Efficient Simulation of a Multiprocessor," *Proc. of the First International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, La Jolla, California, Jan. 1993.

[Magnusson94] Peter S. Magnusson, "Partial Translation," Swedish Institute of Computer Science, Mar. 1994.

[May87] Cathy May, "Mimic: A Fast S/370 Simulator," *Proc. of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques; SIGPLAN Notices*, 22(6): 1-13, Jun. 1987.

[MIPS86] MIPS, *Languages and Programmer's Manual*, MIPS Computer Systems, Inc., 1986.

[NG88] David Notkin and William G. Griswold, "Extension and Software Development," *Proc. of the 10th International Conference on Software Engineering*, 274-283, April 1988.

[Nielsen91] Robert D. Nielsen, "DOS on the Dock," *NeXTWorld*, 50-51, Mar./Apr. 1991.

[Pittman87] Thomas Pittman, "Two-Level Hybrid Interpreter/Native Code Execution for Combined Space-Time Program Efficiency," *ACM SIGPLAN Symposium on Interpreters and Interpretive Techniques*, 150-152, Jun. 1987.

[Ramsey93] Norman Ramsey, *Personal comm.*, Jun. 1993.

[RHLLLW93] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers on Measurement and Modeling of Computer Systems," *ACM SIGMETRICS*, 48-60 , Jun. 1993.

[Richardson92] Stephen E. Richardson, "Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation," SMLI TR92-1, Sun Microsystems Laboratories, Inc., Sep. 1992.

[SCKMR93] Richard L. Sites, Anton Chernoff, Matthew B. Kerk, Maurice P. Marks, and Scott G. Robinson, "Binary Translation," *CACM*, 36(2): 69-81, Feb. 1993.

[SE93] Gabriel M. Silberman and Kemal Ebcioğlu, "An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures," *IEEE Computer*, 39-56, Jun. 1993.

[SE94] Amitabh Srivastava and Alan Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proc. of the 1994 ACM Conference on Programming Language Design and Implementation (PLDI)*, 1994 (to appear).

[SJF92] Craig B. Stunkel, Bob Janssens, and W. Kent Fuchs, "Address Tracing of Parallel Systems via TRAPEDS," *Microprocessors and Microsystems*, 16(5): 249-261, 1992.

[Sosič92] Rok Sosič, "Dynascope: A Tool for Program Directing," *Proc. of the 1992 ACM Conference on Programming Language Design and Implementation (PLDI)*, 12-21, Jun. 1992.

[SPARC9] "The SPARC Architecture Manual, Version Nine," SPARC International, Inc., 1992.

[SunOS4] *SunOS Reference Manual,* Sun Microsystems, Inc., Mar. 1990.

[SunOS5] *SunOS 5.0 Reference Manual,* SunSoft, Inc., Jun. 1992.

[SW79] H. J. Saal and Z. Weiss, "A Software High Performance APL Interpreter," *APL Quote Quad*, 9(4): 74-81, Jun. 1979.

[UMIPSV] *UMIPS-V Reference Manual*, MIPS Computer Systems, Inc., 1990.

[VF94] Jack E. Veenstra and Robert J. Fowler, "MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors," *Proc. of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 201-207, Jan. 1994.