How to Detect Self-Modifying Code During Instruction-Set Simulation

David Keppel
Copyright © 2003, all rights reserved
dk-0001@xsim.com

September 1994; Version 4, revised April 2003

Abstract

The growing maturity and deployment of instruction-set simulators highlights a need for efficient and correct simulation of all processor and platform features. Many applications and operating systems change the instruction space dynamically: dynamic linking, just-in-time compilation, and so on. Unfortunately, no single simulator implementation works well in all situations. This paper presents general strategies and specific solutions for efficient simulation of instruction-space modification.

1 Introduction

Instruction-set simulators are used widely because they allow computers to run programs written for other machines, enable detailed performance analysis, and support sophisticated debugger features. A simulator is most useful when it implements most features of the *target* machine being simulated, and when it operates efficiently on the *host* machine which runs the simulator. The Shade papers [CK93, CK94, CK95] survey simulators, their uses, and implementation techniques.

Many systems use instruction-space modification. Widely-used examples include dynamic linking, just-in-time compilers, debugger breakpoints, and graphics. Both operating systems and applications use it. The granularity ranges from megabytes down to individual bits in instructions. Changes may be one-shot or may occur as often as every time the code is executed. Keppel's dissertation [Kep96] has a survey and bibliography of many types and uses of instruction-space modification.

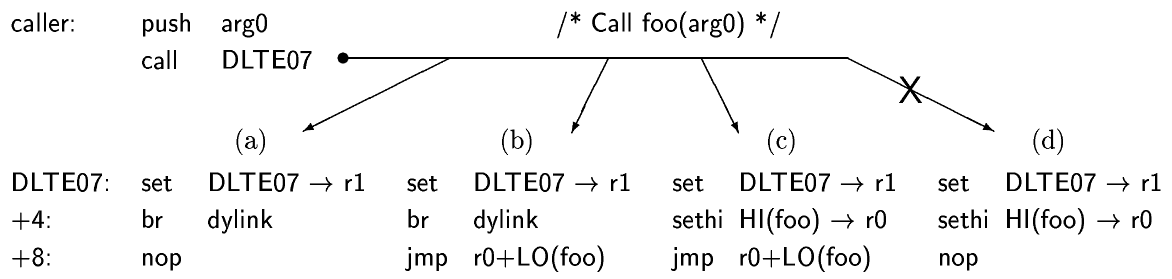
Unfortunately, modern simulators often fail to implement instruction-space modification, or they run much slower in order to implement it. Decoding target instructions is typically expensive, so fast simulators often cache a decoded form of instructions. Caching allows simulators to skip expensive decoding, but also introduces a *cache coherency problem*: when an instruction changes, the cache may hold an old version of the instruction but the simulator must execute the newest one. Although specific coherency cases can be efficient, the general case is difficult.

In the best case, the target system provides a *primitive* for signaling coherency. Both hardware and simulators can then ignore instruction-space changes until the primitive is invoked. For example, an architecture may provide a special inval instruction. A caching simulator ignores instruction-space changes. When target code executes inval, the simulator discards cached forms of all target instructions. Further execution re-decodes each target instruction before caching it. Thus, the simulator uses the latest version of each instruction any time real hardware would use the latest version.

However, simulators are often needed where the best case is not available. Many architectures lack coherency primitives. In other cases, primitives have been circumvented using faster application- and platform-specific code that works reliably on specific platforms, but which does not work on a straightforward simulator. Thus, an important obstacle to efficient simulator implementation is detecting instruction space modification.

Finally, there is a growing use of dynamic linking, just-in-time compilers, debuggers, and so on. At the same time, some of today's most common hardware lacks coherency primitives. Thus, coherency issues are an ongoing concern, not simply "legacy" problems which will soon go away.

This paper focuses on efficient detection of instruction space modification. Section 2 describes some of the many uses of instruction-space modification. Section 3 describes primitives provided by some architectures and platforms, as well as some non-compliant coherency strategies. Section 4 describes a generic caching simulator and considers general difficulties and opportunities in detecting instruction-space modification. Section 5 describes specific implementations. Finally, Section 6 concludes.



```
caller:    push  arg0                       /* Call foo(arg0) */
           call  DLTE07

                  (a)                (b)                  (c)                  (d)
DLTE07:    set   DLTE07 → r1   set  DLTE07 → r1   set   DLTE07 → r1   set   DLTE07 → r1
+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 routine foo; put address of foo in r2 ...
           or    "jmp r0+LO([?])", LO(r2) → r3      /* Form jump to foo */
           st    r3 → r1+8                          /* Install in DLTE */
           call  coherency r1+8                     /* Propagate to instruction caches */
           or    "sethi HI([?]) → r0", HI(r2) → r3  /* Setup for the jump to foo */
           st    r3 → r1+4                          /* Install in DLTE */
           call  coherency r1+4                     /* Propagate to instruction caches */
           jmp   r2                                 /* Jump to foo */
```

Figure 1: A dynamically-linked call to foo(), showing a dynamic link table entry (DLTE) being updated. Initially, (a), it invokes the dynamic linker dylink. During update, (b), the entry is still a valid call to the dynamic linker, so that concurrent calls see a valid instruction sequence and call the dynamic linker. In the final form, (c) the DLTE has been updated to call the dynamically-linked routine. The dynamic linker must perform instruction cache coherency, "call coherency r1+8", 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.

2 Code Modification Uses

Instruction-space modification can affect a single bit in an instruction, or may replace gigabytes of code. An instruction may modify itself, another instruction in the same process, or instructions in some other process. Unfortunately, no simulator techniques work well for all uses, and simulator techniques which work especially well in certain cases tend to work very poorly in other cases. Thus, a clear understanding of how instruction-space modification is used can help lead to an efficient simulator implementation.

One use of instruction space modification is allocating new chunks of code. Code may be read in from secondary storage, or code which is already loaded for one process may be mapped so it is shared with another. In both scenarios, instructions are typically added to a process without removing any others, and instructions are immutable and remain valid as long as the program is executing. At a system level, these uses often change page mappings, which in turn typically means updating the translation lookaside buffer or *TLB* before they are valid.

Another common use is generating new chunks of code. Such uses involve writing instructions, rather than reading them in. Systems often generate dozens to millions of instructions at once. The instructions are typically write-once, but often the memory is reclaimed, and new instructions are written to the same locations. Typically, instructions are valid for a long time; are used many times; are discarded rather than being modified; and are discarded as a block, so if one instruction in a sequence becomes invalid, all instructions in that group are invalidated together.

Finer-grained changes may change one or a few instructions in a sequence. For example, a dynamic linker may use a set of stubs. The first time each stub is invoked, it branches to the dynamic linker. The dynamic linker resolves the callee and patches the stub to branch directly to the callee. An example is shown in Figure 1. In some settings, a fragment may be updated repeatedly during its lifetime.

Very fine-grained modification may change part of an instruction. For example, on register-constrained machines it is common to write a value to the immediate field of a following instruction rather than saving the value to memory then later reloading it. Such usage is shown in Figure 2. The instruction may be written once and used repeatedly, or may be modified every time it is executed.

Some memory regions are used in particular ways. For example, many systems use stack memory alternately for data and for code generated by BITBLT [PLR85, Loc87, FvDFH90].

```
sub    r0 ← r1,r2
  ...
add    r9 ← r0,r8
```

a

```
sub    r0 ← r1,r2
st     r0 → 18[gp]
  ...
ld     r0 ← 18[gp]
add    r9 ← r0,r8
```

b

```
sub    r0 ← r1,r2
st     r0 → (OFFSET(X))[pc]
  ...
add    r9 ← r8,X:[?]
```

c

Figure 2: Code that generates, then uses a value. In (a), there are plenty of free registers, and the value is simply held in a register. In (b), there are insufficient registers, so static code saves, then reloads the value. In (c), the value is saved as an immediate of the instruction that consumes the value. Although doing so takes space in the instruction stream, the value is prefetched and used without an explicit reload.

The "actor" which modifies an instruction may also vary. Cross-process patching is common. For example, debuggers often implement breakpoints by replacing a debuggee instruction with a trap instruction. Some dynamic linkers are similar, with the patching of Figure 1 done by the operating system rather than a user-space dynamic linker. The actor may be in the same process, but may be in a different module or procedure. Or, the actor may be in the same sequence of instructions, as in Figure 2, where one instruction writes an immediate to a later instruction. In extreme cases, an instruction may self-modify, changing itself as it runs, so that subsequent invocation of that instruction will behave differently.

Usually, instruction changes must be propagated quickly to ensure the system executes the most recent version of the instructions. Sometimes, it is only necessary that the change is recognized eventually. For example, if the dynamic linker in Figure 1 is idempotent, the initial code may be executed repeatedly with only a loss of performance.

Sometimes, changes can take effect too soon. One application intentionally corrupted a following instruction. It relied on the instruction having already been prefetched, so that the unmodified version would always be executed. The goal was to complicate reverse-engineering: single-stepping the program would flush the prefetch and thus execute the corrupted instruction. Unfortunately, processors and simulators with stricter coherency also executed the corrupted instruction and the application would fail [Col95]. Another scheme relied on similar behavior to discover the processor model [Lei93].

Some code relies on atomic updates of the instruction stream. For example, code in Figure 1 may be executed concurrently by several threads. It is correct if sethi has been written yet the older br gets executed; it is an error to execute some bits from one instruction and some bits from the other. Some code relies on atomic update of groups of instructions. For example, Figure 1 can omit "call coherency r1+8" if entries at +4 and +8 are always on the same cache line. But simulators that cache individual instructions may execute the illegal sequence in Figure 1(d).

Readers should note that there are many variations on the above themes [Kep96], but most workloads on most platforms use just a few. Thus, part of simulator design and development is measuring and analyzing actual workloads in order to choose appropriate coherency implementations.

3 Coherency Primitives

There are many different architectural and/or platform primitives used to indicate changes to the instruction space. In addition, some primitives are expensive, so some applications use custom routines which work on selected platforms but which do not work across all members of the family. Unfortunately, these custom routines rarely work on straightforward caching simulators. A further complication is that primitives are often ill-specified or the actual behavior does not match the specification [Kep96].

Long ago, few processors used any sort of instruction caching. Therefore, any instruction I_1 could modify any other instruction, I_2, and the processor would reliably execute the modified version of I_2. However, pipelines, prefetching, instruction and data caches, write buffers, and other such structures may hold unmodified forms of an instruction indefinitely. Thus, many processors have explicit primitives to ensure that modified code is propagated to all relevant caching structures. Each change to the instruction space must also call the primitive to ensure the change is executed correctly.

Along with the primitives is a coherency model. Code modification usually starts with a *write event*, typically a store. Most models require a separate *target coherency event* before they guarantee the next *execute event* for an instruction will execute the new value instead of the old "stale" value.

Different systems have a wide variety of primitives [Kep91, Kep96]. Primitives include special instructions, reads or writes of special registers, coprocessor commands, special traps, calls to special routines, and so on. Primitives may depend on both the processor and the system platform: some caches are integral with the processor, while others depend on support chips, the system board, and so on. Primitives may also depend on the operating system and on run-time configuration of the hardware resources.

Some examples demonstrate the range of primitives. One common coherency primitive is simply an indicator that "coherency is needed". Such primitives are common on machines with small instruction caches that are refilled from a coherent source. A more sophisticated primitive indicates a region of change. Often, the region is an aligned block such as "32 bytes starting at a 32-aligned address". Other primitives indicate a base and length of coherency.

Coherency running times vary greatly. For example, the SPARC `iflush` instruction signals coherency on an aligned fixed-size region. Although `iflush` is guaranteed to work on all SPARC processors, some implementations may complete in just a few cycles, while platforms with off-processor caches may trap and run tens of thousands of instructions. Thus, using `iflush` to ensure coherency of a large memory region may be cheap or may be quite expensive.

Primitives vary greatly, and some require privileged operation. Therefore, many operating systems provide abstract interfaces that work with any underlying implementation. Simulators can discover instruction space modification using platform primitives, abstract interfaces, or both.

Some applications bypass the provided primitives because they are slow compared to what is possible with custom code. For example, some applications effect coherency of a modified instruction by executing another instruction which maps to the same line in a direct-mapped cache [Kep96]. Such "coherency by code placement" may be an order of magnitude faster on hardware, but simulators have trouble recognizing such use as instruction space modification, especially because coherency is effected using ordinary instructions. Thus, *every* instruction is potentially being executed in order to effect coherency.

Some platforms update caching structures implicitly. That is, there is no explicit coherency primitive. Instead, the processor provides a guaranteed constraint. For example, many members of the IA-32 processor family have a 16-byte prefetch. The processor guarantees it will execute a modified instruction provided that the instruction write is followed by a branch or at least 16 bytes of other instructions. Newer IA-32 processors have special hardware to force coherency, and appear as if there were no prefetching or pipelining at all [Kep96].

Some instruction-space modifications rely in part on changes to the processor's address mapping. Therefore, simulators may need to track the paging structures. For example, virtual address `0x1000` may initially map to instruction `A` at physical address `0x4000`, then be remapped to `B` at `0x5000`. Although no memory values have changed, the instruction space *has* changed, and any values cached with the tag `0x1000` must be made coherent.

Finally, note that some applications "happen" to work on a given implementation, but are actually buggy with respect to the hardware coherency model. Thus, a simulator or hardware may correctly implement the coherency model but still be unable to run some applications (see "tunable discard", §5.3).

4 Caching Simulators

Conceptually, instruction-set simulators operate by repeatedly reading the current machine instruction, decoding it to determine what it "means", then performing the effect of that instruction. Decoding is often the most expensive part. Fast instruction-set simulators often cache a decoded form of instructions so repeated decoding may be avoided.. For this discussion, static-translation simulators [Fla94, CK95, CHH$^+$98] count as "caching" simulators.

Figure 3a shows sample code for a decode-and-dispatch simulator. It fetches and decodes each instruction on each execution. Figures 3b and 3c show sample code for a caching translator. Figure 3b shows the common case, in which the target program counter maps directly to a decoded handler. Figure 3c shows the cache miss case, where the cache is reloaded by fetching and decoding an instruction.

Caching can reduce decode costs, but it also introduces coherency problems: when a target instruction is modified, the simulator must execute the modified version rather than the unmodified original. The simulator in Figure 3a always fetches and decodes the latest version of the instruction, but the simulator in Figures 3b and 3c only fetches and decodes the instruction on a cache miss.

For example, suppose address `0x40` initially holds instruction `0x54ea340c` which is an `add`. When the simulator executes that instruction, it loads `cache` with the association {`0x40`, `sim_add`}. Later, the application overwrites `0x40` with a branch. However, as long as the original association stays cached, the simulator executes `sim_add` instead of `sim_br`. Better caches can give higher simulator performance but may also hold more "stale" instructions.

Therefore, caching simulators typically must do extra work to detect code modification. In the simplest case, the target processor or platform provides a coherency primitive to ensure coherency of hard-

```
loop {
    i = fetch (vs.pc)
    h = decode (i)
    (*h)(vs)
}
```

a

```
loop {
    h = find (cache, vs.pc)
    if (!h) { h = xlate (vs)
              save (cache, h, vs.pc) }
    (*h)(vs)
}
```

b

```
xlate (vs) {
    i = fetch (vs.pc)
    h = decode (i)
    return h
}
```

c

Figure 3: Common simulator implementations. Fragment (a) is a decode-and-dispatch simulator. Instructions are fetched and decoded every time they are executed. Decoding resolves to a handler, h, which simulates the effect of the instruction, updating the virtual (simulated) state vs. Fragments (b) and (c) show a caching simulator. Each mapping from an instruction address vs.pc to a handler is saved in cache. In the common case, fetch and decode are skipped. Mappings for a given vs.pc are created by calling xlate().

ware caches; applications use that primitive; and the primitive is a good match to the simulator. For example, the SPARC provides an iflush instruction that performs coherency of a 32-byte region whose virtual address is in register a0. Simulators can implement iflush by discarding decoded instructions for that address range. The simulator of Figure 3b, for example, might clear all mappings from cache which are tagged with an address in the 32-byte range starting at a0.

More complicated situations have a poor match between the target, target workloads, the coherency primitive, and the simulator. For example, Shade is a SPARC simulator that generates and caches host-code *translations* of target-code sequences. The translation cache can be searched by virtual address, but some reachable translations cannot be found by the search [CK93]. Therefore, Shade implements iflush by discarding all translations, not just those near the address in a0. The strategy works well provided that iflush is used rarely. When iflush is used frequently, Shade discards many valid translations. Translations are re-created when they are next needed, but re-creation overhead slows simulation.

Sometimes, simulator caches are invalidated without executing a store. For example, caches may be invalidated when base registers change [May87] or when page mappings change [MW94].

Finally, note that an *emulator* is a simulator with hardware support [Tuc65]. Such support may speed coherency. For example, with multiple write-protect bits per page, one bit can implement normal write protections, and the others trap writes to locations for which there are translations [KCW01, BAG+02].

5 Simulator Implementations

A simulator may implement coherency using any strategy that makes sense. An instance of code modification usually consists of a target instruction write event; a target coherency event if the target supports it; and a target instruction execute event. A caching simulator has internal events when it reads and caches instructions, and when it dispatches to and executes code that implements a decoded instruction.

Efficient detection of instruction-space modification typically depends on recognizing key modification events, then mapping those events on to details of the simulator implementation. A simulator may use any, all, or none of the events to detect and implement instruction-space modification. It may also use events in an "unintended" manner. For example, a simulator may use the target coherency event to mark which items need coherency, but defer actual coherency until the next execute event. Finally, a simulator may may switch between several strategies, each tuned to different uses.

In researching various strategies, it may be useful to study the sequence of operations performed by hardware or a non-caching simulator and compare that against the caching simulator. Such a comparison often highlights which events are weakened or removed by the caching simulator, and can thus help point to an appropriate solution.

Consider, for example, simulating target locations A, B, C. Figure 4a shows a trace of operations performed by a decode-and-dispatch interpreter. Notably, the simulator decodes A every time it is executed. If A is executed, changed, then re-executed, the simulator will re-decode A when it is re-executed, and thus will correctly notice changes to A.

Figures 4b and 4c show operations performed by a caching simulator. emit saves away the result of the decode without actually executing it. Figure 4b generates Figure 4c, so all operations in Figure 4b are executed before any in Figure 4c. If locations A, B, C are re-executed, Figure 4c is reexecuted directly without re-executing Figure 4b.

How can we modify a caching simulator to detect instruction space modifications? One solution is

```
fetch A        || fetch A                                fetch A
decode A       || decode A                               A ≠ A_saved ⇒ regenerate
t = vs.regA    || emit A_host     t = vs.regA            t = vs.regA
u = f_A(t)     ||                 u = f_A(t)             u = f_A(t)
vs.regB = u    ||                 vs.regB = u            vs.regB = u
vs.pc += 4     ||
fetch B        || fetch B                                fetch B
decode B       || decode B                               B ≠ B_saved ⇒ regenerate
t = vs.regB    || emit B_host
u = f_B(t)     ||                 u' = f_B(u)            u' = f_B(u)
vs.regC = u    ||                 vs.regC = u'           vs.regC = u'
vs.pc += 4     ||
fetch C        || fetch C                                fetch C
decode C       || decode C                               C ≠ C_saved ⇒ regenerate
t = vs.regC    || emit C_host
u = f_C(t)     ||                 u" = f_C(u')           u" = f_C(u")
vs.regD = u    ||                 vs.regD = u"           vs.regD = u"
vs.pc += 4     ||                 vs.pc += 12            vs.pc += 12
               || dispatch        dispatch               dispatch

      a              b                 c                      d
```

Figure 4: Operations performed by various simulators while executing locations A, B, C. Code (a) fetches, decodes, and simulates every instruction on every execution. Code (b) fetches and decodes instructions, then emits (c), which simulates the instructions. Note that (c) runs without fetching or decoding: when A, B, C is re-executed, only the operations in fragment (c) are repeated, so changes to A, B, or C go unnoticed. Code (d) is like (c), but it fetches and compares each instruction against the value when the sequence was created. If any have changed, the sequence is regenerated. Each f_i corresponds to a specific value of h in Figure 3. The dispatch operation finds the next sequence to execute; it corresponds approximately to Figure 3b.

shown in Figure 4d: the generated code is augmented to fetch each instruction and compare the current value against the value of the instruction when the code was created [May87]. Executing the instructions N times thus causes the instruction to actually be fetched $N + 1$ times. But where comparing is cheaper than decoding, the overall cost of Figures 4b and 4d can be less than that of Figure 4a.

Following sections describe coherency alternatives in more detail. Section 5.1 describes basic techniques. Sections 5.2 and 5.3 describe ways to tune or combine techniques for better performance in specific situations. The techniques are described in terms of target code, but external agents such as debuggers, I/O, etc. are similar.

5.1 Basic Techniques

Interpret – No Cached Form Simulators may avoid coherency problems by avoiding cached forms, but performance tends to be worse. For example, Spa is an interpreter for a straightforward target that runs on an identical underlying host. It is coded in assembly and requires about 40 instructions to simulate each instruction [Irl93]. Shade is a caching simulator; in similar use, it is often an order of magnitude faster [CK94]. Thus, removing the cached form altogether may be unacceptable, but avoiding caching in certain situations may solve coherency problems while still giving good performance for common use.

Discard A simulator may generate and execute a decoded form, then immediately discard it [CK94]. The effect is similar to interpretation, but may be simpler to implement given an existing translator. Generating code is typically slower than interpreting, so discard performance is usually worse. Several instructions may be translated together, so care is needed to maintain dependencies. For example, failures can occur translating more than 16 bytes on an IA-32 (§3); past writes of S/370 base registers [May87]; past execute [Bro60, May87] or iflush instructions; and so on.

Target Coherency Primitive For systems which have and use a target coherency primitive, use the primitive to mark or discard all of the cached form which might cover the indicated region. Some target primitives indicate incoherency, but do not say what is incoherent. Thus, straightforward use of such primitives may lead to excess invalidation, because all cached information is discarded, not just the portion which is incoherent.

Mutator Checking Code does not change by itself; some agent must change it. Executing such mutators can signal simulator coherency. It is generally hard to determine which instructions change other instructions, but identifying specific cases can be useful [Hay94].

Tag Checking Cached forms are tagged. Most systems tag by instruction address, so coherency can compare addresses. For example, a store address can be compared against addresses which have cached forms; the address of the next-to-execute cached form can be compared against addresses of prior memory writes; or the address of the current target instruction may be compared against known mutators. Note that checking all stores may be expensive in general, but some cases may be handled quickly. For example, code write checks may be integrated with other page write checks [MW94]. Other tags include address spaces, timestamps, and so on.

Value Checking Stores cause incoherence by changing values with a cached forms. Thus, coherency can be signaled by looking for value changes. For example, a simulator can record the value of memory when a decoded form is cached; when the decoded form is executed, it compares the saved and current values of memory. If the values differ, coherency is needed. Value checking saves the original target instructions, and may read both saved and current instruction values as data each time a target instruction is executed [May87, DGB$^+$03]. Thus, execution overhead and memory pressure can be high.

5.2 Optimization Strategies

None of the above techniques excels in all settings. However, most schemes can be tuned to a particular situation. This section describes several general optimization hints [Lam84, Kep93], applied to coherency.

Beware that optimization typically risks several problems. First, performance depends on choosing the right optimization; the wrong one may make performance uniformly worse. Second, common-case costs are usually improved at the expense of uncommon-case costs, so optimization may make some workloads faster but others much slower. Third, available performance improvement is limited by the cost of choosing a given strategy, as well as the cost of the strategy itself. Fourth, some schemes incur additional space costs, and using several schemes together usually incurs further space costs. Space cost overheads can include data cache costs, instruction cache costs, paging overheads, write buffer stalls, and so on. Fifth, caching optimizations can introduce their own coherency problems. Sixth, using several implementations increases the risk of bugs at the same time that it reduces the coverage of any given implementation. Thus, using several implementations may lead to reliability problems and dramatically increased development time and costs.

Be Conservative Sometimes it is expensive to be precise If coherency is conservative, it often need not be exact. For example, any write to a page might force coherency for the whole page. Beware that conservative approaches may lead to problems like *false sharing*, where operations near a cached form do not cause coherency problems, yet incur the same cost as operations on an actual cached form.

Speculate Coherency can be performed speculatively, anticipating a need. For example, the first write to a page speculatively discards all cached forms for that page, so further writes to the page can omit coherency until subsequent execution from the page.

Stage A given instance of code modification is composed of many events (Section 5). The handling of coherency may be staged across multiple events. For example, it may be cheapest to mark coherency when a target coherency primitive is executed, but to defer actual coherency until the cached form is used.

Be Lazy Defer operations or parts of operations to minimize up-front cost and maximize the number of things that are checked together. For example, at a write event, record the address of a potential incoherency; at a target coherency event, record the address of an expected coherency; at an execute event, implement coherency on just those locations which were both changed and expected to be coherent.

Cache Caching can make common operations fast. For example, it may be expensive to check if a write touches any interesting page. If a write usually touches the same page repeatedly, it may speed the common case to first compare against the last page touched. Beware of coherency issues.

Combine With Other Operations Some coherency checks can be combined with other simulator operations so that a single operation checks several conditions simultaneously. For example, write checks for page protection and coherency may be combined; or, tests for dispatch and coherency may be combined. When combined tests fail, perform separate tests to determine the root cause.

Compose Some operations are best built using several cooperating strategies. For example, the refetching implementation of Figure 4d combines interpretation and translation, but it only performs those parts of interpretation which are needed to allow translation to operate correctly.

Use a Hierarchical Implementation If a cheap operation is sometimes inadequate, try the cheap form first and fall back on the more expensive. For example, first check if an address is in read-only memory; test for value changes only in writable memory.

Use Host Hardware It may be possible to use cheap host operations to implement parts of coherency. For example, write-protect target pages with cached from, then load the host TLB [RHWG95] with the lesser of target and simulator write enables, thus making common write checks go faster.

Hybridize Use several cheap strategies instead of one expensive one. For example, use page-level checks outside of the stack segment, where false sharing of code and data should be rare, and use fine-grained checking or interpretation for code in the stack segment, where false sharing is likely.

Adapt Change strategies while executing. Three general adaptive strategies are to deoptimize, to reoptimize with new assumptions, or to choose among several specialized implementations [Kep96]. For example, a simulator may default to cheap page-level coherency, but if coherency operations are frequent, it might switch to a finer-grained strategy with more expensive checking but lower overall coherency costs. Adaptation typically requires ongoing measurement to determine when and where adaptation is required. Thus, there is often measurement overhead. Also, it may be impossible to measure exactly, so measurement error needs to be considered in choosing adaptation thresholds. Note that adaptation may "hunt", continually trying new strategies, but never settling on one. Thus, adaptation can sometimes lead to worst-case performance as it both chooses bad implementations, then also pays the cost of adapting.

Approximate Real systems crash; faulty approximate solutions may be "good enough" if they fail much less often than the overall system. For example, approximate value checking by computing and comparing checksums or CRCs instead of actual values. Use the size of summarized value, the summarizing algorithm, and a plausible/conservative write rate to compute the odds that a write produces a different exact value but the same summary value. Use approximation only if it is much more reliable than the overall system failure rate.

Phase Divide execution into phases and keep a virtual timestamp to indicate system or implementation behavior. Comparing against a timestamp may be cheaper than other tests. For example, a system which tracks writes may note the "time" of the last write to a page. Newer cached forms are, by definition, coherent and need no further checking.

Pool/Aggregate/Cluster Some policies work best if like things are considered together. For example, cached locations with especially high or low modification rates may be considered together so that thresholds or adaptation rates can be adjusted to reflect system usage rather than that of an isolated location.

Specialize Rather than solving the general problem, solve a specific problem well. If necessary, adaptively change the kind or degree of specialization. For example, instead of solving the general problem of detecting coherency by code placement (§3), focus on solving the particular placement code known to appear in a particular workload. Similarly, FlashPort simulates a target code generator by creating and installing new host code. Doing so simplifies detection of code changes, and allows use of fast host code generators tailored to the specific use [Hay94].

Tune A simulator can implement various mechanisms and allow the user to tune the implementation by selecting between mechanisms or providing their own [Kep93]. Choices include correct mechanisms with differing performance characteristics, and mechanisms that are correct in some situations but which fail in others. For example, Shade provides a slow-but-correct strategy for executing applications which omit a required iflush (see "tunable discard", §5.3).

Undo Save incoherent versions of the cached form and the conditions for which it is valid. When possible, *requalify* the form and reinstate it, rather than recreating it. For example, Mimic caches multiple translations corresponding to different values of the execute register; checking to requalify a fragment is typically much faster than recreating it [May87].

Finally, note that many of the techniques described here are commonly used for implementing programming languages. The overlap is not coincidence: machine code is simply another programming language. Many techniques apply broadly, so studying other programming systems may provide hints about how to improve simulators.

5.3 Specific Implementations

This section describes in more detail several existing and hypothetical coherency schemes, in order to highlight tradeoffs.

Target Coherency Primitive In g88, target instruction cache invalidation primitives cause the corresponding threaded code to be replaced with a pointer to a primitive which re-decodes the invalidated instructions [Bed89].

Target Coherency Primitive, Conservative Shade translations are variable-size, making it hard to identify which translations contain a given address. Also, translations "fall off" of the indexing data structure, making them hard to discover, yet they may be reached by chaining from other translations (§4). Thus, Shade implements the target iflush instruction by discarding all translations [CK93].

Tunable Discard The SPARC architecture defines a target coherency primitive, iflush. Some older applications fail to use iflush, yet ran on older SPARC hardware [CK93]. Shade has a command-line option that instructs Shade to generate then discard translations for code outside of the read-only text segment [Cme93]. Discarding code is potentially expensive, but works well in practice because most time is spent executing from the read-only text segment.

Undo Value Checking Tag the code generated for each use of the execute instruction. If an instruction reappears, the cached form is revalidated using a simple comparison, which is cheaper than generating new code. Typically, few distinct values are used for any given application run [May87].

Instruction Address Checking When a mutator (§5.1) can be identified, a simulator can implement coherency based on executing target instruction addresses. Such "bracketing" is only guaranteed correct if all mutators of a fragment can be identified. Thus, it may be desirable to use a hierarchical approach, with a slower conservative strategy to catch the cases missed by the faster bracketing mechanism.

Composed Caching and Interpretation Where instruction changes are frequent but limited in scope, use fast cached execution for slow-changing parts, and interpretation for frequently-changing parts. For example, in Figure 2c, only the immediate varies from invocation to invocation. Thus, the add can be predecoded, but the immediate interpreted [DGB$^+$03].

Conservative Value Checking Value checking can examine just the bytes which make up the cached form, but using larger regions can reduce storage and checking costs: one test can cover several instructions and code can be tuned for fixed-size chunks. Disadvantages include false sharing overheads.

Hierarchical Dispatch-Time Value Checking Straightforward value checking performs a load, compare, and branch for each use of each instruction. Consider checking only when the target cache is filled. For example, some targets have direct-mapped caches and ensure code A is coherent by executing some other code B that maps to the same cache location [Kep96]. A simulator can use a map with the same structure as the hardware cache; the simulator performs value checking only when the map misses.

Conservative Write-Time Checking Straightforward write-time checking can be expensive because each simulated write requires a lookup on the cached form. A simulator can instead "bin" memory and blindly discard any cached form which falls in the bin of a store. False sharing may increase re-decode costs, but writes are so frequent that the cost of precise write checks may be higher.

Hierarchical Host/Software Testing Often, a host MMU can map target memory [RHWG95] so pages with a cached form will trap on write. However, many MMUs map only part of the address space. Also, simulator pages must be accessible to the simulator but invisible to the simulated target. Thus, each application access may be "guarded" with a software range check. Usually, target accesses touch locations mapped by the MMU, and will thus go fast.

Hierarchical Write Checking and Execution-Time Undo Each write conservatively removes cached forms from the corresponding bin in a fast map and places them on a slow list. Misses in the fast map check the slow list. Hits in the slow list are requalified: those which pass are reinstated in the fast list, while those which fail are either discarded or kept for later requalification [DGB$^+$03].

Hierarchical Write-Time/Execution-Time Address/Value Checking Each write sets a flag in a map; each time a cached form is executed, it polls the corresponding flag. If set, the cached form is requalified. With one bit per instruction, the scheme is accurate; with several per bit it is conservative.

Hybrid Adaptive Interpretation Sophisticated pre-decoding can yield high asymptotic performance, but may be expensive where code changes are common. The simulator can try a sophisticated approach but also track usage and costs. Where too expensive, the simulator selects successively cheaper but asymptotically slower forms of pre-decode, eventually switching to interpretation [DGB$^+$03].

6 Conclusions

Fast instruction-set simulators can perform many useful functions such as cross-machine simulation, data collection for performance analysis, and sophisticated debugger features. To do so, the simulators must be fast. Caching techniques help simulators go fast, but target workloads use dynamic instruction space modification, which then violates caching assumptions. This paper has presented approaches which help maintain the benefits of caching, while extending the workloads which can run on those simulators. There are many approaches and no clear winner, but the wide variety of approaches provides the simulator writer with a good set of tradeoffs and thus good performance across a variety of workloads.

7 Acknowledgements

Thanks to Bob Cmelik for discussing these ideas and to Bob and Stephen Russell for reviewing earlier drafts of this paper. This work supported in part by NSF PYI Award #MIP-9058-439, Sun Microsystems, and Stephen Russell.

References

[BAG$^+$02] John Banning, H. Peter Anvin, Benjamin Gribstad, David Keppel, Alex Klaiber, and Paul Serris. Fine Grain Translation Discrimination. United States Patent #6,363,336, March 2002.

[Bed89] Robert Bedichek. Some Efficient Architecture Simulation Techniques. *Proceedings of the 1989 USENIX Conference*, pages 53–63, 26 October 1989.

[Bro60] Fredrick 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.

[CHH$^+$98] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32 – A Profile-Directed Binary Translator. *IEEE Micro*, 18(2), March/April 1998.

[CK93] Robert F. Cmelik and David 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.

[CK94] Robert F. Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[CK95] Robert F. Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In Thomas M. Conte and Charles E. Gimarc, editors, *Fast Simulation of Computer Architectures*, chapter 2, pages 5–46. Kluwer Academic Publishers, 1995.

[Cme93] Robert F. Cmelik. The Shade User's Manual, February 1993.

[Col95] Robert 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.

[DGB$^+$03] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing™ Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. *Proceedings of the 2003 International Symposium on Code Generation and Optimization*, pages 15–24, March 2003.

[Fla94] FlashPort Technology Overview, 1994.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics — Principles and Practice*, pages 988–992. Addison-Wesley, 1990.

[Hay94] Dave 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 [Fla94] by making it generate and execute host code.

[Irl93] Gordan Irlam. Personal communication to Robert F. Cmelik. See also [CK94], 1993.

[KCW01] Edmund J. Kelly, Robert F. Cmelik, and Malcolm J. Wing. Translated Memory Protection Apparatus for an Advanced Microprocessor. United States Patent #6,199,152, March 2001.

[Kep91] David 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 2003/01, code at "ftp://ftp.cs.washington.edu/pub/pardo".

[Kep93] David Keppel. Managing Abstraction-Induced Complexity. Technical Report 93-06-02, University of Washington, Department of Computer Science and Engineering, June 1993.

[Kep96] David Keppel. *Runtime Code Generation.* PhD thesis, University of Washington, 1996.

[Lam84] Butler W. Lampson. Hints for Computer System Design. *IEEE Software*, 1(1):11–28, January 1984.

[Lei93] Richard C. Leinecker. Processor Detection Schemes. *Doctor Dobb's Journal*, pages 46–49, 126–127, June 1993.

[Loc87] Bart N. Locanthi. Fast BitBlt With asm() and CPP. *European Unix Users Group Conference Proceedings (EUUG)*, September 1987.

[May87] Cathy May. Mimic: A Fast S/370 Simulator. *Proceedings of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques; SIGPLAN Notices*, 22(7):1–13, June 1987.

[MW94] Peter Magnusson and Bengt Werner. Some Efficient Techniques for Simulating Memory. Technical Report R94:16, Swedish Institute of Computer Science, September 1994.

[PLR85] Rob Pike, Bart N. Locanthi, and John F. Reiser. Hardware/Software Trade-offs for Bitmap Graphics on the Blit. *Software—Practice and Experience*, 15(2):131–151, February 1985.

[RHWG95] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, 3(4):34–43, Winter 1995.

[Tuc65] S. G. Tucker. Emulation of Large Systems. *Communications of the Association for Computing Machinery (CACM)*, December 1965.