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

David Keppel amas-bt-2009@xsim.com

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, paging, and so on. Unfortunately, no single simulator implementation works well in all situations. This paper surveys general strategies and specific solutions for efficient simulation of instruction-space modification.

1 Introduction

Instruction-set simulators are used widely to allow computers to run programs written for other computers, 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 runs 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 [Kep96] surveys many types and uses of instruction-space modification.

Unfortunately, simulators often fail to implement instruction-space modification, or 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, yet the simulator must execute the newest one. Although specific coherency cases can be efficient, the general case is difficult.

In the best case, a target provides a *primitive* for signaling coherency, and both hardware and simulator ignore instruction-space changes until the primitive is invoked. For example, an architecture may provide a cache coherency instruction `inval`. When target code executes `inval`, the simulator discards cached forms of all target instructions, so further execution re-decodes each target instruction before use. Thus, the simulator uses the latest version of each instruction any time real hardware is guaranteed to use the latest version.

However, simulators are often needed where the best case is not available. Many architectures lack coherency primitives, including the widely-used and commonly-simulated x86. In other cases, applications circumvent expensive primitives using faster application- and platform-specific code that works reliably on specific platforms, but which fails on a straightforward simulator. Thus, an important obstacle to efficient simulator implementation is detecting instruction space modification.

Finally, there is continued use of paging, dynamic linking, just-in-time compilers, debuggers, and so on. Thus, coherency issues are an ongoing concern, not simply "legacy" problems which will soon go away.

Multiple-processor systems have more complex coherency and concurrency/atomicity issues and are beyond the scope of this paper.

This paper focuses on efficient detection and implementation of instruction space modification. §2 describes some uses of instruction-space modification. §3 describes primitives provided by some architectures and systems, as well as some non-compliant coherency strategies. §4 presents a generic caching simulator and considers problems and opportunities detecting instruction-space modification. §5 describes specific implementations. Finally, §6 concludes.

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 another 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 loaded from secondary storage, or code in one process may be mapped so it is shared with another. In both scenarios, instructions are typically added without removing others, are immutable, read-only, 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. 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 rarely modified; and are discarded as a block, so if one instruction in a sequence becomes invalid, all instructions in that group are invalidated together. Overlays are similar.

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 then patches the stub to branch directly to the callee. An example is shown in Figure 1. In some settings, fragments are updated repeatedly during their 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 for both data and for code generated by BITBLT [PLR85, Loc87].

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; on "continue" the debugger patches in the original instruction, single steps, then re-patches the `trap`. Some dynamic linkers are similar, with the patching of Figure 1 done by the operating system or a separate linker process. An actor may be in the same process, but in a different module or procedure. Or, it may be in the same instruction sequence, as in Figure 2, where one instruction writes a following instruction. In extreme cases, an instruction self-modifies, changing itself as it runs, so subsequent invocation of the instruction behaves differently.

An "actor" that modifies the instruction space need not be a CPU memory write. DMA can modify code via an asycnhronous I/O engine [BKKK03]. Addressing changes can change the instruction space and may occur with e.g., register writes that change addressing base registers or which enable or disable paging [May87, MW94]. `execute` instructions fetch from a register, not memory [Bro60, May87].

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, a dynamic linker as in Figure 1 is typically idempotent to ensure safety for interrupts and multiprocessors. Thus, executing the initial or intermediate code repeatedly costs only performance.

Sometimes, changes can take effect too soon. One application intentionally corrupted a following instruction. It relied on the changed instruction being prefetched, so the uncorrupted version was executed. The goal was to complicate reverse-engineering: single-stepping flushed the prefetch and executed the corrupted instruction. The application was changed when processors with stricter coherence were introduced [Col95]. Another scheme relied on similar behavior to discover the processor model in the absence of a CPU identifier [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 cor-

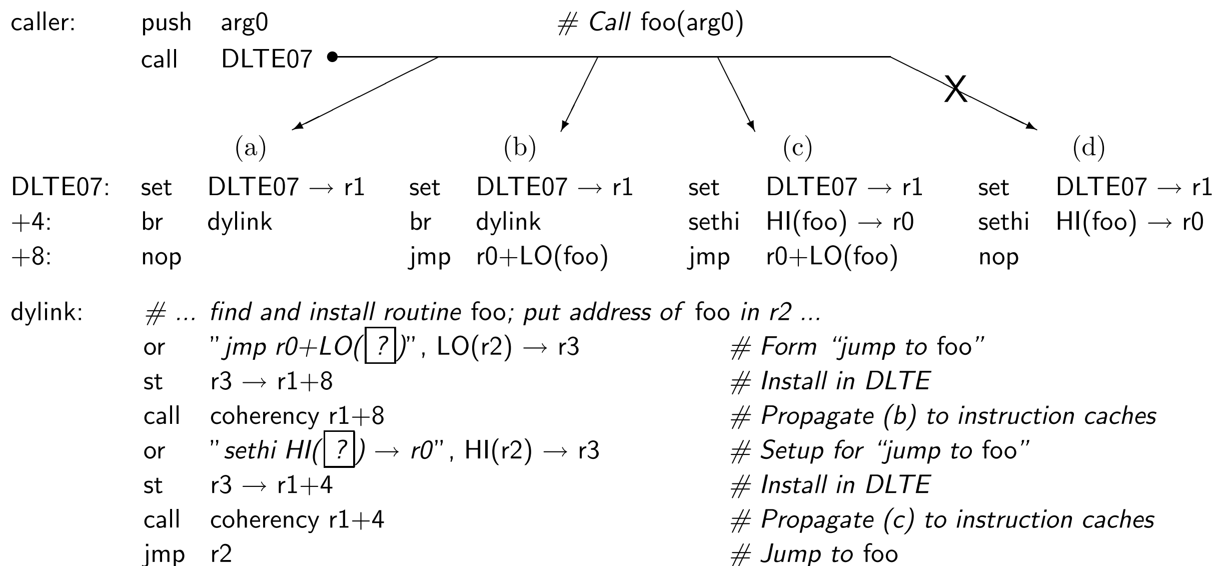

Figure 1: A dynamically-linked call to foo(), showing a dynamic link table entry (DLTE) being updated. Initially, (a), it invokes the dynamic linker dylink. During update, (b), the entry is still a valid call to the dynamic linker, so concurrent calls see a valid instruction sequence and call the dynamic 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.

rect to execute the new sethi with the old br; it is an error to execute some new bits and some old bits from one location.

Code may rely 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 a simulator that caches individual instructions may execute the illegal sequence in Figure 1(d).

Alternatively, code may rely on *non*-atomic updates of groups of instructions. One coherency scheme flushes the instruction cache once, then allocates disjoint code fragments to disjoint cache lines, thus allowing code creation and execution to be interleaved without any coherency operations [Kep96].

There are many variations on these themes [Kep96], but most workloads on most platforms use just a few. Thus, part of simulator design is measuring and analyzing actual workloads to guide selection of coherency implementations.

3 Coherency Primitives

Long ago, few processors used any sort of instruction caching. Therefore, any instruction could modify any other instruction and the processor would reliably execute the modified version. However, pipelines, prefetching, instruction and data caches, write buffers, and other structures may hold unmodified forms of an instruction indefinitely. Thus, many processors have explicit primitives to ensure modified code is propagated to all relevant 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 *coherency event* before they guarantee the next *execute event* for an instruction will execute the new value instead of the old "stale" value.

There are many different architectural and/or plat-

```
(a)
sub   r0 ← r1,r2
 ...
add   r9 ← r0,r8

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

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

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.

form 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, custom routines often fail on straightforward caching simulators. A further complication is primitives are often ill-specified, or actual behavior does not match the specification [Kep96].

Primitives vary widely and 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 runtime configuration of the hardware resources.

Some examples demonstrate the range of primitives. One common primitive is simply an indicator that "coherency is needed". Such primitives are common on machines with small instruction caches refilled from coherent memory. A more sophisticated primitive indicates a region of change, such as an N-byte aligned N-byte block, or a base address and length of coherency.

Primitive running times vary greatly. For example, the SPARC iflush instruction signals coherency on an aligned fixed-size region. iflush works on all SPARC processors, but some implementations take just a few cycles, while platforms with off-processor caches may trap and run tens of thousands of instructions. Thus, repeated calls to 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 slow primitives and use custom code. For example, coherency of a modified instruction may be effected by executing another instruction that 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, since coherency is effected using ordinary instructions. Thus, *every* instruction is potentially being executed to ensure coherency.

Some platforms guarantee coherency without an explicit primitive. Instead, the processor provides a guaranteed constraint. For example, some x86 processors have a 16-byte prefetch. The processor guarantees it will execute a modified instruction, provided the instruction write is followed by a branch or at least 16 bytes of other instructions. Newer IA-32 processors force immediate 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 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, some applications are buggy with respect to the hardware coherency model, yet "happen" to work on specific implementations. Thus, a simulator or hardware may correctly implement the coherency model yet 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, so fast simulators often cache a decoded form of instructions to avoid repeated decoding. For this discussion, static translation and persistent caching [May87, SCK+93, Fla94, CHH+98, RMD08] 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 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 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 0x54ea340c which is an add. When the simulator executes that instruction, it loads cache with the association {0x40, sim_add}. Later, the application overwrites location 0x40 with a branch. However, as long as the original association is cached, the simulator executes sim_add instead of sim_br. Larger caches can give higher simulator performance but can also hold more "stale" instructions.

Caching simulators must, therefore, do extra work to detect code modification. In the best case, a target processor or platform provides a coherency primitive to ensure coherency of hardware 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 target, target workloads, the coherency primitive, and simulator. For example, the Shade SPARC simulator generates and caches host-code *translations* of target-code sequences. The translation cache is searched by virtual address, but some reachable translations cannot be found by the search [CK93], making it hard to find all translations invalidated by a given iflush. Therefore, Shade's iflush implementation discards all translations, not just those near the address in a0. The strategy works well when iflush is used rarely. When iflush is used frequently, Shade discards many valid translations, and re-creation overhead slows simulation.

Finally, hardware support may speed coherency. For example, with multiple write-protect bits per page, one bit can implement target write protection and the others can trap writes to locations the simulator has cached [EAGS01, KCW01, BAG+02].

5 Simulator Implementations

A simulator may implement coherency using any strategy that makes sense. A code modification instance 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 a target coherency event to mark items needing coherency, but defer actual coherency

```
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. The common case skips fetch and decode. Mappings for a given vs.pc are created by calling xlate().

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 them 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 a good 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 re-decodes A when it is re-executed, and thus correctly notices changes to A.

Figures 4b and 4c show operations performed by a caching simulator. emit saves the result of the decode without 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 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 an instruction N times thus fetches it $N+1$ 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 strategies in more detail. §5.1 describes basic techniques. §5.2 and §5.3 describe ways to tune or combine techniques for better performance in specific situations. Though described in terms of target code, external actors 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 [May87] or iflush instructions; and so on.

Target Coherency Primitive For systems that have and use a target coherency primitive, use it to mark or discard cached forms which may cover the indicated region. Some target primitives indicate in-

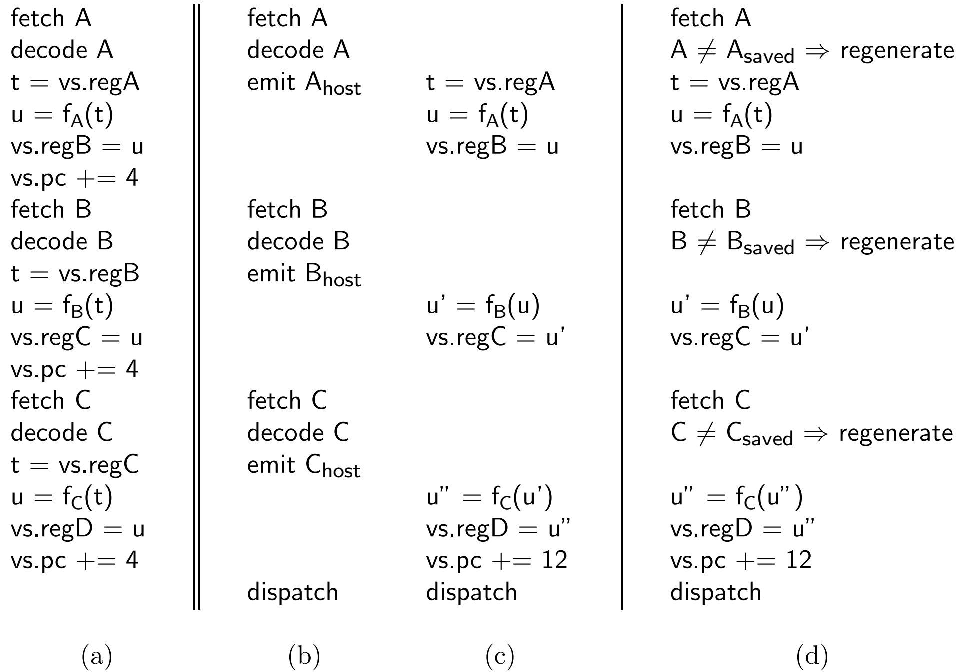

Figure 4: Operations performed by various simulators while executing locations A, B, C. Code (a) fetches, decodes, and simulates every instruction on every execution. Code (b) fetches and decodes instructions, then emits (c), which simulates the instructions. Note that (c) runs without fetching or decoding: 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 fetches and compares each instruction against their values 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.

coherency, but do not say what is incoherent. Thus, straightforward use of such primitives may lead to excess invalidation, because all cached values are discarded, not just those which are incoherent.

Mutator Checking Code does not change by itself; some mutator must change it, and executing mutators can signal simulator coherency. Typically, all writes are assumed to be mutators, as it is generally hard to determine precisely which instructions change other instructions, though target coherency primitives can help to identify mutators, and identifying specific cases can be useful [Hay94].

Tag Checking Cached forms are tagged. Many systems tag cached forms by instruction address and check each store's address against tags. Checking all store addresses is 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 tagging checks are possible, such as comparing the current form's tag against preceding store addresses. or tagging with 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 can be tuned to a particular situation. This section describes several general optimization hints 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 cause simulator performance to vary within a workload or across workloads. Third, performance improvement is limited by the cost of choosing a 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 can include data cache, instruction cache, paging, 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, so 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 can force coherency for the whole page. Beware that conservative approaches may lead to problems like *false sharing*, where cached forms near a change are coherent, yet incur the same cost as incoherent forms.

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/Be Lazy A given instance of code modification is composed of many events (§5). The handling of coherency may be staged across multiple events. For example, at a write event, record the address of a potential incoherency; at a target coherency event, record the address of expected coherency; at an execute event, implement coherency on just those locations which are 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 writes 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 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.

Use a Hierarchical Implementation If a cheap operation is sometimes inadequate, try the cheap form first and fall back to the more expensive. For example, first check if an address is in read-only memory; further test for 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 forms and load the host TLB [RHWG95] with the lesser of target and simulator write enables, thus making common write checks faster.

Hybridize Use several cheap strategies instead of one expensive one. For example, for ordinary code, tag it with base register addresses and for `execute`, copy the value of the instruction [May87].

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 for each page where coherency operations are frequent, switch to a finer-grained strategy with more expensive checking but lower overall costs. Adaptation typically requires ongoing measurement to determine when, where, and how 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. For example, adaptation can "hunt", continually trying new strategies, but never settling on one and paying both the cost of a bad strategy and the cost of adaptation.

Approximate Real systems crash; faulty approximate solutions may be "good enough" if they fail much less often than the overall system. For example, compute and compare a CRC insteady of actual

values. Use a model to compute the odds of a different memory value but the same summary value. Use approximation only if it is much more reliable than the accepted 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 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 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 application.

Tune A simulator can allow the user to tune the implementation by choosing between simulator mechanisms or by providing their own [Kep93]. For example, Shade usually uses iflush to signal codespace changes. Applications omitting the required iflush can be run using an optional slow-but-correct strategy that discards each translation from writable pages immediately it runs. Tuning choices include correct mechanisms with differing performance characteristics, and mechanisms that are correct for some workloads, but which fail for others.

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 execute register values; checking to requalify a fragment is typically much faster than recreating it [May87].

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

5.3 Specific Implementations

This section describes several schemes in more detail, to highlight tradeoffs.

Target Coherency Primitive; Combine In g88, target instruction cache invalidation primitives cause translated threaded code to be replaced with a pointer to a primitive to re-decodes the invalidated instructions [Bed89]. Like ST-80's receiver caching [DS84], the retranslation is coded in the callee pointer, rather than requiring extra checks in the caller. g88 maps memory using a multi-level DAG so decoded instruction dispatch uses no conditionals.

Target Coherency Primitive, Conservative Shade translation lookup uses a cache, so translations may be reachable by chaining from other translations, yet not otherwise findable to invalidate them selectively when the target performs an iflush. Shade could keep additional data structures, but instead implements iflush by discarding all translations [CK93]. This approach is simple and reliable, and performance is good as long as iflush is rare.

Write-Time Discard A system can check all writes and discard cached forms that overlap with writes. Various systems include explicit checks in the write simulation. SimOS sets page protection to the lesser of target and simulator write enables, so host writes fault on target code changes [RHWG95]. Crusoe and DAISY for PowerPC use hardware with a "coherency" bit per page to speed write checks [EAGS01, KCW01]. DAISY for x86 and S/390 uses hardware with one coherency bit per translatable unit, 1 bit per byte for x86 and 1 bit per 2 bytes for S/390 [EAGS01]. A Bull GCOS 8 simulator uses several coherency bits per 36-bit translatable unit [MNC03].

Adaptive Discard DynamicRIO groups contiguous pages, and discards translations for a group when any page in the group is written, adaptively splitting a group when code on one page writes code on another page in the same group [BA05].

Hierarchical Discard QEMU protects pages at translation time, and page writes discard translations for the page. When running with a software MMU, frequent invalidations cause QEMU to generate a bitmap for the page and only discard translations that overlap with stores [QEM09]

Tunable Discard The SPARC ABI requires an iflush on code modification. Some old applications fail

to use it, yet run on some SPARC hardware [CK93]. Shade has a command-line option instructing Shade to generate, execute, then discard translations of code from writable pages [Cme93]. This approach risks high retranslation overhead, but works well in practice as most time is spent executing read-only code.

Undo Value Checking Mimic tags code generated for each value of the execute instruction. If an instruction reappears, the cached form is revalidated using a cheap comparison instead of generating new code. Typically, few distinct values are used for any given application run [May87]. Crusoe tags translations on frequently-written (sub-)pages with their memory value when translated. When a new translation for the region is needed, Crusoe first checks a list of previously-valid translations [BAG+02, DGB+03].

Adaptive and Conservative Value Checking For frequent code page writes, Crusoe generates "self-requalifying" translations tagged with aligned, fixed-size copies of target code. If subpage trap rates are too high, Crusoe leaves the page writable, and generates "self-checking" translations that check memory on each invocation. Self-checking avoids false sharing, and while more expensive per invocation, avoids trap overheads. If self-check fails too often, Crusoe generates translations that invoke the interpreter [BAG+02, DGB+03].

Hierarchical Write Checking and Execution-Time Undo Crusoe hardware has hardware subpage coherency protection. At translation, subpages are marked read-only. Subpage writes trap, the page is marked writable, and affected translations are moved to a save list. When the subpage is next executed, it is again marked read-only, and translations whose tags match current memory are again made executable [DGB+03].

Mutator Specialization FlashPort simulates a target code generator with an application-specific host code generator. Translator performance is improved, as it avoids the indirect path of generating target code then translating it to host code. Generated code performance is also improved, as host code is generated directly by an application-specific host code generator, instead of using an application-specific target code generator followed by a general-purpose target-to-host translator. Finally, the specialized mutator ensures coherency directly, rather than relying on detection of code changes [Hay94].

Hybrid 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. For this case, Crusoe generates translations that interpret the immediate by fetching it from the target instruction [DGB+03]. Various systems use static translation for the fast cached form [SCK+93, CHH+98]. An interpreter may also use caching, with lower coherency costs than the faster cached form [RMD08].

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 a target with direct-mapped caches, applications sometime ensure instruction A is coherent by executing an instruction B that maps to the same cache location [Kep96]. A simulator can use a map with the same structure as the hardware cache and perform value checking only when the map misses.

Hierarchical Host/Software Testing SimOS maps target memory at host addresses using the host MMU. Writable target pages with translations are marked read-only to trap on write (see "Write-Time Discard", above). The host MMU must map simulator pages, leaving them exposed to stray references from the simulator. Thus, each target access is "guarded" with a software range check, with remaining checks performed using the host MMU [RHWG95].

Conservative Target Coherency Primitive Nirvana uses the IA-64 coherency primitive as a hint for faster execution, but modified code is executed correctly even if the primitive is omitted [BCdJ+06].

Phased Target Coherency Primitive PowerPC's ICBI instruction implements coherency by invalidating an instruction cache block. On page load, the AIX operating system runs ICBI for all blocks on the page. BOA performs coherency at the page level, so detects repeated ICBI from a page without interleaved translation from that page, thus speeding common use [AG06].

Phased Translation A Bull GCOS 8 simulator uses interpretation during OS boot, when code modification rates are high, then switches to translation for normal operation [MNC03].

6 Conclusions

Fast instruction-set simulators are used for cross-machine simulation, dynamic optimization, data collection for performance analysis, and to implement sophisticated debugger features. Caching techniques help the simulators go fast, but target workloads use dynamic instruction space modification, which can violate caching assumptions. This paper presents approaches to maintain the benefits of caching, while supporting workloads with dynamic code. There are many approaches and no clear winner, but the wide variety of approaches provides the simulator writer with a set of tradeoffs and that can give 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

[AG06] E. Altman and M. Gschwind. Dynamic compilation at the ssystem level, 2006. Slides from CGO talk.

[BA05] D. Bruening and S. Amarasinghe. Maintaining consistency and bounding capacity of software code caches. In *Proc. of the International Symposium on Code Generation and Optimization (CGO)*, 2005.

[BAG+02] J. Banning, H. P. Anvin, B. Gribstad, D. Keppel, A. C. Klaiber, and P. Serris. Fine grain translation discrimination. United States Patent #6,363,336, Issued 2002.

[BCdJ+06] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Conference on Virtual Execution Environments*, page 154, Jun. 2006.

[Bed89] R. Bedichek. Some efficient architecture simulation techniques. *Proc. of the 1989 USENIX Conference*, pages 53–63, Oct. 1989.

[BKKK03] P. Boyle, D. Keppel, A. C. Klaiber, and E. Kelly. Software direct memory access, U.S. Patent #6,668,287, Issued 2003.

[Bro60] F. P. Brooks, Jr. The execute operations – a fourth mode of instruction sequencing. *Communications of the Association for Computing Machinery (CACM)*, 3(3):168–170, Mar. 1960.

[CHH+98] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. FX!32 – a profile-directed binary translator. *IEEE Micro*, 18(2), Mar./Apr. 1998.

[CK93] R. F. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. Sun Microsystems Laboratories, Inc., Technical Report SMLI 93-12; University of Washington Dept. of Comp. Sci. & Eng., UWCSE 93-06-06, 1993.

[CK94] R. F. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[CK95] R. F. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In T. M. Conte and C. E. Gimarc, editors, *Fast simulation of computer architectures*, chapter 2, pages 5–46. Kluwer Academic Publishers, 1995.

[Cme93] R. F. Cmelik. The shade user's manual, February 1993.

[Col95] R. Colwell. *The P6 microprocessor.* Distinguished lecturer series. University of Washington Dept. of Comp. Sci. & Eng., 9 Nov. 1995.

[DGB+03] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. C. Klaiber, and J. Mattson. The Transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. *Proc. of the 2003 International Symposium on Code Generation and Optimization*, pages 15–24, Mar. 2003.

[DS84] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proc. of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, Jan. 1984.

[EAGS01] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6), Jun. 2001.

[Fla94] Flashport technology overview, 1994.

[Hay94] D. Hayden. Personal communication, Aug. 1994.

[Irl93] G. Irlam. Personal communication to R. F. Cmelik, 1993.

[KCW01] E. Kelly, R. F. Cmelik, and M. Wing. Translated memory protection apparatus for an advanced microprocessor. United States Patent #6,199,152, Mar. 2001.

[Kep93] D. Keppel. Managing abstraction-induced complexity. Technical Report 93-06-02, University of Washington, Dept. of Comp. Sci. & Eng., Jun. 1993.

[Kep96] D. Keppel. *Runtime code generation.* PhD thesis, University of Washington, 1996.

[Lei93] R. Leinecker. Processor detection schemes. *Doctor Dobb's Journal*, pages 46–49, 126–127, Jun. 1993.

[Loc87] B. Locanthi. Fast Bitblt with asm() and CPP. *European Unix Users Group Conference Proc. (EUUG)*, Sept. 1987.

[May87] C. May. Mimic: a fast S/370 simulator. *Proc. of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques*, 22(7):1–13, Jun. 1987.

[MNC03] G. Mann, B. Noyes, and R-J. Chevance. Method and apparatus for emulating self-modifying code, 4 Feb. 2003. U.S. Patent 6516295.

[MW94] P. Magnusson and B. Werner. Some efficient techniques for simulating memory. Technical Report R94:16, Swedish Institute of Computer Science, Sept. 1994.

[PLR85] R. Pike, B. Locanthi, and J. Reiser. Hardware/software trade-offs for bitmap graphics on the Blit. *Software—Practice and Experience*, 15(2):131–151, Feb. 1985.

[QEM09] *QEMU internals.* http://www.nongnu.org. 2009.

[RHWG95] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the SimOS approach. *IEEE Parallel and Distributed Technology*, 3(4):34–43, Winter 1995.

[RMD08] M. Reshadi, P. Mishra, and N. Dutt. Hybrid compiled simulation: an efficient technique for instruction-set architecture simulation. 2008.

[SCK+93] R. Sites, A. Chernoff, M. Kerk, M. Marks, and S. Robinson. Binary translation. *Communications of the Association for Computing Machinery*, 36(2):69–81, Feb. 1993.