Draft paper, in submission to PLDI'16

Living on the edge: Rapid-toggling probes with cross modification on x86

Buddhika Chamith Bo Joel Svensson Luke Dalessandro Ryan R Newton
Indiana University
budkahaw@indiana.edu, bo.joel.svensson@gmail.com, ldalessa@indiana.edu, rrnewton@indiana.edu

Abstract

Dynamic probe injection is now a widely used method to debug performance in production. Current techniques for dynamic probing of native code, however, rely on an expensive stop-the-world approach: binary changes are made within a safe state of the program—typically in which all the program threads are halted—to ensure that another thread executing the modified code region doesn't step into a partially-modified code.

Stop-the-world patching is not scalable. In contrast, low overhead, scalable probes that can be rapidly toggled on and off in-place would open up new use cases for statistical profilers and language implementations, even traditional ahead-of-time, native-code compilers. In this paper we introduce safe *cross-modification* protocols that mutate x86 code between threads but do not require quiescing threads, resulting in radically lower overheads than existing solutions. A key problem is handling instructions that straddle cache lines. We empirically evaluate existing x86 architectures to derive a safe policy given current processor behavior, and we argue that future architectures should clarify the semantics of instruction fetching to make cheap cross-modification easier and future proof.

1. Introduction

Modifying program binaries while they are running is an important technique in operating system kernels [1], JIT compilers [20], and simulators [16]. Projects including DynInst [4] and Intel Pin [12] have explored the role of binary instrumentation in performance modeling and analysis. High-quality frameworks for *dynamic probes*—such as DTrace [11]—have also popularized the use of binary modification in interactive performance debugging.

In this paper we focus on probes, specifically, on the question of whether modern x86 hardware can support truly scalable dynamic probes. Semantically, the concept of a probe is simple, and yet their uses are wide-ranging. A probe is merely a conditional function call inserted into an application at runtime or compile time:

```
if (probe_active) (*funptr)( probe_id );
```

The user of the probing library determines the function to attach (`funptr`), and we assume some identifier, `probe_id`, to distinguish *from which* probe site the call originates. Probes inserted *statically* are guarded by a conditional, as above. Yet this incurs overhead—not just branches, but loading distinct `probe_active` flags for each probe site. If we aim to probe all functions some of the time, then an application can contain *millions* of probe sites.

Dynamic instrumentation strategies like DTrace can avoid the overheads of the static approach, allowing a probe to have zero cost when it is deactivated, and thus overhead proportional to the number of *active* probes. And yet, existing approaches to dynamic probes have their own scalability bottlenecks:

- **Stop-the-world code mutation**: dynamic probing provided by DTrace, LTTng [8], DynInst [4] and SystemTap [23] require a *separate* instrumentor process that pauses application threads while activating or deactivating probes. Indeed, even the HotSpot JVM, which controls code generation, applies modifications at "safe points" where bytecode threads are stopped [7].
- **Single-pass design**: Systems like Intel Pin are designed to translate code into a code cache (usually once, at the cost of significant start-up overhead); thus probe insertion is trivial but probe *toggling* is not directly supported and would be extremely expensive if it invalidates the code cache.

Thus, in spite of a great deal of work on binary instrumentation tools for x86 (reviewed in Section 10) to our knowledge there are no solutions for *scalable* probes, scaling to large numbers of threads, probes, and toggle events. Specifically, we seek a solution meeting these criteria:

- Minimal startup overhead on the critical path
- No global barriers across application threads
- Rapid, threadsafe toggling of individual probes
- Low or no overhead for *inactive* probes

The development of such a tool would enable moving some powerful offline analyses online. For instance, the Intel Cilk parallel scheduler contains latent probes to record and analyze the parallel task graph (e.g., work vs span), but

```
// -- OPTION 1 --
Store modified code (as data) into code segment;
Jump to new code or an intermediate location;
Execute new code;

// -- OPTION 2 --
Store modified code (as data) into code segment;
Execute a serializing instruction;
  // For example, CPUID instruction
Execute new code;
```

Figure 1. Portable protocols for self-modifying x86 code. Reproduced from Intel SDM, Section 8.1.3 [15].

only in an expensive offline mode with probes activated in a single pass by Intel Pin [26]. Fast and scalable probe toggling, however, could enable periodically running this analysis online, as in a running server application.

Or, in another example, scalable probes could enable narrowly focused interactive performance analysis techniques to become always-on unattended measurements. Today, a performance engineer can log into a server and ask precise questions in real time with DTrace (a *microscope*). And conversely Google gathers coarse statistical profiling data [25] for entire data centers (a *macroscope*), but sufficiently cheap probes enable bringing certain precision measurements from the former, to the latter. We demonstrate an example of this with *latency profiling* in Section 9.

Viewed another way, cheap, scalable dynamic probes have the potential to bring online profiling opportunities—some of which are already available to JITs—to compilers and language runtimes that use ahead-of-time native code compilation, e.g., for C++, Fortran, Haskell, Go, Rust, etc. For example, in the case of the Intel Cilk runtime, mentioned above, dynamic probes run *custom code* to traverse data structures—functionality that cannot be provided by traditional PC-sampling and interrupt-driven profiling.

The fundamental challenge Why do existing solutions use expensive strategies—out-of-place binary translation of all instructions, `ptrace` to stop processes, calls into the kernel on every probe? One reason is the need to support arbitrary program rewrites, not just dynamic probes. But another fundamental reason is the combination of mutating code *in place* and running code on multiple cores can be unsafe. If a thread mutates a single, 5-byte CALL instruction across threads. When will the other threads see it? If it crosses a cache-line boundary, will the other thread see some bytes before others?

This paper asks the question: what does it take to make cross-modification of dynamic probes safe in practice on *current* hardware? And, further, what clarifications of instruction fetch semantics would make cheap, scalable probes officially supported by future processors? We quantify the benefits of these cross modification techniques as an argument for this future clarification.

```
// -- Action of Modifying Processor --
memory_flag = 0;
Store modified code (as data) into code segment;
memory_flag = 1;

// -- Action of Executing Processor --
WHILE (memory_flag != 1)
Wait for code to update;
ELIHW;
Execute serializing instruction; // e.g: CPUID
Begin executing modified code;
```

Figure 2. Portable protocol for cross-modifying x86 code. Reproduced from Intel SDM, Section 8.1.3 [15].

In this paper, we make the following contributions:

- We develop a model for x86 instruction fetch and determine empirically that it is correct on modern x86 implementations.
- We use this model to create novel cross-modification algorithms for x86 that do not rely on global barriers and demonstrate that they outperform previous approaches.
- We provide libraries for modifying arbitrary instructions in a word (*wordpatch*), and enabling/disabling specific CALL instructions (*callpatch*).
- We show how to use `wordpatch` and `callpatch` capabilities to build a proper user-facing probing library (*libfastinst*), and we then evaluate these libraries in terms of (1) microbenchmarks and (2) in the context of an example instrumentation-based profiler, which we apply to parallel and sequential C/C++ applications.

2. Background: Memory Model

Imperative programmers are familiar with the traditional sequential execution model in which the system executes instructions in program order, one at a time, where each instruction completes before the next one starts. Modern microprocessor architectures contain networks, caches, write buffers, and out-of-order superscalar pipelines that do everything they can to accelerate sequential programs short of violating this expectation—*except when it comes to self-modifying code*.

Instruction fetch and issue are often a bottleneck in a super-scalar pipeline, particularly for complex instruction sets like x86 [28]. Much of Chapter 2 in Intel's Software Developer's Manual (SDM) is devoted towards high-level descriptions of optimizations like prefetching, macro-op fusion, μop fusion, μop caching, and loop streaming, all of which transparently accelerate read-only code execution but violate sequential execution for read-write code execution [15].

In order to preserve sequential execution, portable self-modifying x86 code is required to follow one of the two protocols laid out in Section 8.1.3 of the SDM, reproduced in Figure 1. Ignoring these protocols results in model-specific behavior. Historically however, they are adequate for self-modifying code.

Shifting to parallel programs dramatically complicates the picture. Parallel programmers often expect shared memory systems to be sequentially consistent [19], naturally assuming that there is global total order of memory operations that is consistent with program order in each thread. Unfortunately, the architectural optimizations detailed above that work so well for sequential code permit execution histories that violate sequential consistency.

Modern imperative programming languages like Java and C++ account for this using programmer-centric memory consistency models that allow application developers to synchronize their programs and determine if they have data races or are data-race free, and thus guaranteed to result in sequentially consistent executions [2, 5, 21]. Low-level programs written in assembly (or the object code generated by a compiler) must instead base their expectation of behavior on the hardware-specific memory consistency model for their architecture [10].

The x86 memory consistency model, described in the SDM and formalized in [27], provides the `lock` prefix to provide atomicity to loads, stores, exchanges, and some read-modify-write operations like `compare-and-swap`, and provides memory fence instructions to constrain memory access ordering. While complicated, this model allows experienced low-level programmers to synchronize their applications and predict the possible execution histories–*except when it comes to cross-modifying code*.

As with self-modifying code, the Intel SDM provides the portable protocol in Figure 2 for cross-modifying code—the label they give to self-modifying code that may modify instructions being run on a separate thread. This protocol leverages the x86 memory consistency model to synchronize threads on normal data, safely converting cross-modification to a type of "signaled" self-modification. Unlike the protocol in Figure 1, Figure 2 is entirely inadequate for interesting use cases. For our purposes of binary instrumentation, it cannot be used to update arbitrary executing code as the `memory_flag` metadata, and `while` control flow, need to exist *before* the modification occurs, a catch 22 given that we are trying to modify arbitrary code without that existing infrastructure. Furthermore, while the described protocol supports a trivial one-shot style cross-modification, at least quiescence is required to correctly toggle code locations—there is no other way to ensure that a serializing instruction will be executed by every thread each time a code location is toggled.

Its scalability and/or latency notwithstanding, quiescence may be reached in a number of ways, e.g., through synchronous barrier-like code, timer or interprocessor interrupts, through `ptrace`, or methods such as those used in, e.g., userspace read-copy-update algorithms [9].

We reject these techniques as unsuitable for our purpose. For scalable probes we must be able to perform cross-modification without quiescence. The Intel SDM clearly states that ignoring the protocol for cross-modification, i.e., introducing a modification race, will result in implementation-specific behavior. Our task then is to characterize the instruction fetch and dispatch operations of the x86 architecture according to a useful abstract model, verify this model on a range of x86 implementations, and show how this model can be used to effectively allow cross-modification on the x86 platform.

3. Formal Requirements

The cross modification approach to instruction patching that we describe in Section 5 will rely on two assumptions about the instruction fetch pipeline, (1) that there exists an upper bound on the time between when one processor stores a single byte to a code location and all other processors observe this change, and (2) that processors do not observe instruction byte values that were never written. Furthermore, the actual algorithms defined in Sections 5 and 6 depend on the assumption that stores to words within cache lines are atomic with respect to instruction fetch, i.e., that we can store up to eight bytes (in x86_64) to a single cache line in a single instruction and no processor will see a partial value from these eight bytes. We formalize these assumptions relative to Sewell et al.'s x86-TSO [27] here and validate them on a variety of x86_64 implementations in Section 8.1.

x86-TSO models a global memory plus a store-buffer per hardware thread, and provides an event-based semantics structured around six events: write, read, memory fence, lock, unlock, and internal progress of writes from store buffers to shared memory. Writes, $W_p[a] = v$, and reads, $R_p[a] = v$, are specific to a processor p, address a, and value read or written v. But this model does *not* deal with self-modifying code or misaligned or mixed-size accesses. For our purposes we need to further model operations on memory areas straddling cache lines.

Assume there exists a cache line size, B, (the size of read and write events in x86-TSO) and that all reads and writes to an address are equivalent to one at the start of its aligned cache line, i.e. $W_p[a] = W_p[a - (a \bmod B)]$. Assume also that there exists a word size, $W \leq B$, such that W contiguous bytes may be modified atomically. Then words with byte addresses that straddle cache lines are modeled as two write events bracketed by lock/unlock events: $L_p; W_p[a]; W_p[a + B]; U_p$.

We further must model the instruction fetch process. We assume all instructions are encodable at some size $I \in [I_{min}, I_{max}]$ where $I_{min} \leq I_{max} < B$. We don't directly account for architectural state or microarchitectural details, rather we assume that processors fetch instructions by reads, R^I. As with writes, reading a straddling instruction requires two separate reads to consecutive cache lines. However, instruction fetch follows a *weaker* memory model than normal reads. First, lock/unlock instructions are ignored, as specified in the SDM. Second, μop caches prevent some reads to code memory from being issued. We capture this weaker model as follows:

- Each processor logically issues an *instruction read*, $R^I_p[PC] = v$, of the B-sized cache line containing PC, in a bounded time window before each time it executes the instruction at PC (i.e. changes register state).
- Instruction read events R^I_p can be reordered past lock and unlock events L_p/U_p.
- An adversarial "μop cache", marks a *subset* of read events as *elided*. An elided event, $E(R^I_p[PC])$, is still placed in the event graph, but the value v returned is the value of *last preceding non-elided read*, $R^I_p[PC] = v$ provided that there is a previous read in the cache.
- There exists an upper bound T_{max}, such that a read $R^I_p[a]$ cannot be elided if the time since last read $R^I_p[a]$ is greater than T_{max}.
- Times such as T_{max} are measured as real, continuous time, which has a monotonic relationship with number of instructions executed on each processor p, and where we assume an upper bound on the real time between any two instructions on the same processor.

Thus, while this models an *incoherent* instruction cache, the upper bound T_{max}, provides a form of *eventual consistency*, or more precisely, a *bounded staleness* property. The algorithms described in Section 5 are safe given the above model, and perform better given smaller T_{max}. The optimized call-site patching in Section 6 can operate *even if* T_{max} does not exist, and example profiling application we describe in Section 9 work even in the extreme case of fully inconsistent, never-invalidated μop caches. Thus this paper presents a sliding spectrum of solutions that improve with the strength of architectural guarantees. And in all cases, our proposals are more efficient than "stop the world" probing.

In the future, processors could take a number of different paths to improve their support for cross-modification. They could respect one or more designated atomic instructions in instruction fetch. They could publish an upper bound on visibility, which could be reported per-processor with a special instruction (similar to `cpuid`).

4. Programming Interface Overview

Now, with our memory model in mind, we describe the API we provide for cross-modifying instructions in memory, starting at the low-level and working up to a complete notion of dynamic probes. Figure 3 illustrates the low level and higher level APIs in use.

Synchronous Word Patching The basic `patch` operation in Figure 3 must replace a single word atomically, such that concurrent threads will only execute the code before or after the patch. It is blocking but may return failure under contention—our expectation is that at least one concurrent `patch` operation will succeed, thus the `patch` should be livelock free. The `patch` interface requires three additional constraints for safe use.

```
// -- Wordpatch API --
bool patch(void *address, uint64_t value);
bool start_patch(void *address, uint64_t value);
bool finish_patch(void* address);

// Callpatch API
bool activate_call(void* address, uint32_t offset);
bool deactivate_call(void* addr);

// Fasinst API
struct ProbeMetaData {
  ProbeId probe_id;
  string function_name;
  enum ProbeType { ENTRY, EXIT};
}

void probe_discovery_callback(ProbeMetaData pmd);
void register_callback(void* callback);
bool activate(ProbeId probe_id, void* probe_ptr);
bool deactivate(ProbeId probe_id);
```

Figure 3. Lower-level patching APIs and fastinst probe API

- Writable code: The `patch` address must be writable. The client may do this eagerly or lazily as part of a signal handler.
- Layout Equivalence: The `patch` operation must not modify the set of valid PC addresses in the program. Furthermore, the `patch` value may only modify bytes corresponding to a single PC, i.e., instruction.[1]
- Disjoint update: no addresses a_1, a_2 may be concurrently modified if $0 < |a_1 - a_2| < 8$. This is because a locking implementation may map locations a_1 and a_2 to different locks.

Asynchronous Word Patching `patch` provides a basic building block that is sufficient for the full probing library we want. However, as we will demonstrate in Section 5, it may be a high latency operation on platforms with a high T_{max}. Thus we also implement an *asynchronous* variant, that separates the act of initiating a patch from finishing the patch. The `start_patch` operation in Figure 3 has the same interface constraints as `patch` but can return before the patch is complete. The `finish_patch` operation must be called to complete each patch operation, and will return `true` when the patch completes successfully.

We implement both the synchronous and asynchronous word patching interface in the `libwordpatch` library.

[1] This constraint can be relaxed. Given the implementation in Section 5 multiple PCs can be updated simultaneously with the same effect as a sequence of independent updates to each instruction, with the added constraint that the patches occur atomically. Furthermore it should be safe to allow additional PCs to be introduced during patching, though this complicates formalization greatly.

Call Toggling While word patching allows the code to transition between any two valid sequences, our scalable probing client works within a more restricted space—merely toggling a five-byte CALL instruction on and off. We define the specific `activate_call`/`deactivate_call` interface for this operation, where the `offset` argument to the `activate_call` instruction is the appropriate four-byte position-independent offset to either the target function or procedure-link-table entry for this call site.

As with the word patching interface, call sites must be writable. Call patching will maintain layout equivalence internally, and still requires disjoint updates. In addition the client must know the correct four-byte offset values and thus each call patch site may require additional initialization.

We implement the call patch interface in the `libcallpatch` library.

Scalable Probes Irrespective of which patching variant we use, we must build up from one-word patches to full insertion of dynamic probes. At that level, our goal is to dynamically attach an indirect call to a function pointer. Providing this kind of API is the goal of the `fastinst` interface in Figure 3 and corresponding library, `libfastinst` (Section 7). Here we've abstracted from raw patch address to `probe_id` locators, because a *probe provider* will have its its own means of identifying and enumerating valid probe sites. Because our approach relies on *in-place* modification, probe sites are not *arbitrary* code locations; they must start out either:

- *activated*: containing a CALL/JMP instruction, or
- *deactivated*: containing any relocatable code sequence of at least five bytes (often a NOOP)

Notice that on x86_64 the function pointer `&myProbe` is a 64-bit virtual address. A call to this address would require more than five bytes. Thus, even with the prerequisites above, there is not room at the patch site to dynamically generate the full code for the call. Rather, we use the standard technique of inserting a short relative jump which calls out-of-line code to: do metadata lookup, execute displaced instructions from the patch site (if needed), and execute the full call sequence to the user-specified function pointer.

The client of the `fastinst` API also needs to register a callback for probe *discovery*. Whenever the probe provider discovers a new probe—either at startup or on first invocation—it will invoke this callback with a probe metadata structure that includes: the function in which it resides, if it is an entry or exit probe for that function, etc. The client can cache the probe ids during discovery and use them in subsequent probe API operations.

Finally, the APIs above are designed for process *self-instrumentation*, rather than the traditional approach (used by DTrace, PIN, LTTng, SystemTap, Dyninst, etc) of a separate process that conducts instrumentation and receives events. We make this choice in order to support transparent and lightweight deployment inside existing applications (with a single `LD_PRELOAD`), and in this respect we use a similar design to DynamoRio [6].

```
// -- Write an interrupt byte into the address,
//    returns true if the byte was already an interrupt --
void int3_lock(address)
  return (atomic_swap_byte(address, INT3) != INT3);

bool patch(address, value)
  if (not is_straddler(address))
    *address = value;
    return true;
  else if (int3_lock(address))
    wait();
    write_back(address, value);
    wait();
    write_front(address, value);
    return true;
  else return false;
```

Figure 4. Synchronous word-patching algorithm

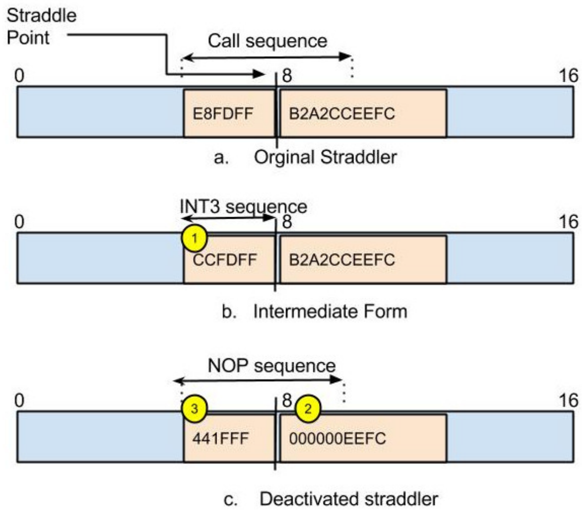
5. Word Patching

In this section we present an algorithm for applying patches for words that may straddle cache line boundaries. The algorithm is based on the processor model presented in Section 3 and uses wait times that exceed T_{max} to ensure that no partially written patches are fetched for execution. By locking patch-sites, the patching algorithm is also safe in situations where there is more than one thread concurrently applying patches, with the aforementioned assumption of no partially overlapping patch-sites.

The implementation of the patching algorithm, `patch`, is outlined in Figure 4. Patches that do not straddle a cache line are applied as a single store operation. Given our formal model, this operation can be done consistently without extra synchronization, and is guaranteed to be seen by concurrent processors within time T_{max}.

The patches that do straddle a cache line boundary, however, are applied as independent new_{fr} and new_{bk} parts, before and after the cache line boundary, respectively. The contents of memory before application of the patch will be referred to as the old_{fr} and old_{bk}. When patching a straddler, `patch` performs the following operations:

- **Try to lock the patch site**: The `int3_lock` operation uses a trap instruction to try to lock the patch site. The trap instruction also protects the patch site from threads arriving at the address after patching has been initiated. These threads will go into a signal handler and spin until the patch has been completely applied. This ensures consistency.
- **Wait**: A wait of T_{max} prevents views of old_{fr} combined with new_{bk}.
- **Apply back part of patch**: write the new_{bk}.
- **Wait**: This wait of T_{max} prevents views of new_{fr} and old_{bk}.



a. Orginal Straddler

b. Intermediate Form

c. Deactivated straddler

Figure 5. Patching straddlers

```
bool start_patch(address, value)
  if (not is_straddler(address))
    *address = value;
    return true;
  else if (int3_lock(address))
    timestamp = get_current_time();
    add_patch(address, value, timestamp);
    return true;
  else return false;
```

Figure 6. Algorithm to starting an asynchronous patch.

- **Apply front part of patch**: This completes the patch and also unlocks the patch-site by overwriting the trap instruction.

In Figure 5 we show what the memory contents will be during the process outlined above. The example shown straddles a cache line boundary after the third byte and we exchange a call instruction with a 5 byte NOOP. In Section Section 8.1 we estimate the T_{max} required between the writes in the algorithm by testing.

5.1 Asynchronous Protocol

Since the `patch` algorithm for patching straddlers involves periods of waiting, it is natural to hide this latency with an asynchronous implementation. The API for asynchronous patching consists of two functions, `start_patch` (Figure 6) and `finish_patch` (Figure 7).

The `start_patch` operates on non-straddling addresses through the same synchronous mechanism used in `patch`. For straddlers, it adds a meta-data object, a `Patch` object, to a global table. The `Patch` object is created in the `add_patch` function and contains: a timestamp, patch value to apply, a lock, and a `State` indicating at what stage of application this patch is in. The states are:

```
bool finish_patch(address)
  if (not is_straddler(address))
    return true;
  Patch p = find_patch(address);
  /* synchronize finish */
  if (not trylock(p.lock))
    return false;
  else if (p.state == FINISHED)
    unlock(p.lock);
    return true;
  else if (not t_max_elapsed(p.timestamp))
    unlock(p.lock);
    return false;
  else if (p.state == FIRST_WAIT)
    write_back(address, p.value);
    p.timestamp = get_current_time();
    p.state = SECOND_WAIT;
    unlock(p.lock);
    return false;
  else if (p.state == SECOND_WAIT)
    write_front(address, p.value);
    p.timestamp = 0;
    p.state = FINISHED;
    unlock(p.lock);
    return true;
  /* unreachable */
```

Figure 7. Algorithm to further the asynchronous patch.

- `FIRST_WAIT`: The trap instruction has been written at the patch address and we are waiting for T_{max} time to elapse before writing the back part of the patch.
- `SECOND_WAIT`: The back part has been written and we are waiting for enough time to pass before writing the front.
- `FINISHED`: The patch has been completely applied.

The `finish_patch` function tries to move an active patch location to a new state, returning true if the patch is `FINISHED`. It applies the same basic protocol as the synchronous `patch` algorithm, and can only transition state if enough time has passed since the patch transitioned into the current state.

Both the synchronous and asynchronous protocols are fast and scalable for non-straddling locations. Patching straddlers is potentially *slow* but remains *scalable*. The actual straddler patch latency depends on the system-specific T_{max} and can be on the order of several thousand cycles, which can also impact the read path given the INT3 lock, however there are no global barriers for readers and patch sites are independent. Furthermore separate threads can patch disjoint locations without interfering. The asynchronous version goes further and allows higher patching throughput by allowing many outstanding, concurrent patches initiated by the same thread. In Section 8 we thoroughly evaluate both protocols.

6. Restricted Call Toggling

Given the restricted interface provided for call toggling defined in Figure 3, the observation that the `patch` operation in Figure 4 simply requires a single non-blocking store operation, and the specific encoding of PC-relative CALL instructions in x86_64, we can construct an algorithm to handle all

```
void __cyg_profile_func_enter(void* func_addr,
                              void* call_site_addr);
void __cyg_profile_func_exit(void* func_addr,
                             void* call_site_addr);
```

Figure 8. `-finstrument-functions` added functions.

PC-relative CALLs *without* depending on `WAIT` and T_{max}. We consider each possible straddle point within the CALL instruction separately, and find a (de)activation solution that requires changing only single cache line—either *front* or *back*, but not both. As with word patching, call patching depends on the *eventual visibility* of modified instructions, i.e., that there exists a T_{max}, but the *value* of T_{max} does not bound the toggle performance. The guarantee to clients, in turn, is eventual delivery of the modified behavior, which is useful for instrumentation applications that are statistical in nature to start with. Even with an unbounded T_{max} it would still be possible to use the callpatch API with the assumption that only the current core will see the changes, but that nothing will *go wrong* in other cores.

As a short PC-relative CALL requires five bytes, an `0xE8` prefix, and a four-byte little endian offset. The instruction will push the PC plus 5 bytes onto the stack as the return value, add the immediate offset to the current PC and jump to that new location. We examine the four possible interior straddling points, transforming them from straddler patches to non-straddling operations.

- **1|4 split**: In this case the straddle point occurs right after the CALL instruction opcode (`0xE8`). To deactivate the call we change the back 4 bytes to the relative address of a degenerate trampoline (a `ret` instruction). In our current implementation we generate such a trampoline lying within a 2^{32}-byte offset and then cache the trampoline address so that it can be used in subsequent deactivations of *any* patch sites within reach of the trampoline[2]. Patching in the original 4 byte offset reactivates the call.
- **2|3 split**: In this case we deactivate the call by rewriting the two bytes in front of the split into a two-byte relative jump to the that skips over the back of the instruction. Restoring the original two bytes reactivates it.
- **3|2 and 4|1 splits**: These cases can be reduced to the 2|3 case where only the first two bytes of the front cache line are modified.

7. Full Dynamic-Probe Implementations

The main concern of call toggling and word patching is to safely apply the specified patch. A full probing implementation requires a mechanism for enumerating these patch sites and calculating the byte sequences to be patched in for (de)activating the probes, potentially caching them per probe site for efficiency reasons. We provide this at the high-level `libfastinst` library that dynamically attaches calls to full 64-bit function pointers.

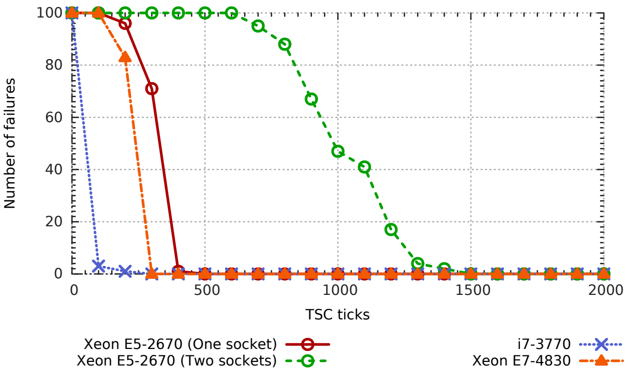


Figure 9. Experimentally determining T_{max} for a selection of microarchitectures.

Conventions for probe starting state and location discovery are tightly coupled. Many different compiler conventions are possible for automatically inserting and recording the locations of probes. Here we focus on the widely available `-finstrument-functions` probe provider, where the basic idea is to have the compiler arrange for call instructions to be already present, but at unknown locations in the application. In this provider, we use the `-finstrument-functions` compiler option to systematically create call instructions to known destinations. This flag instructs the C/C++ compiler to add the calls in Figure 8 to profiling enter/exit symbols at function entry and exits.

Our probe provider in turn implements these `cyg_*` functions and we link them using `LD_PRELOAD` at program start. Thus probes start *on* and are deactivated when called (or in the background by a daemon thread).

The `cyg_*` functions act as trampolines containing a call to the user's function pointer. The function pointer is held as part of per-probe-site metadata. Thus a transition *between* active states with different function pointers requires only modifying the metadata entry to point to a different function. The modification is done as a regular atomic without any involvement of `wordpatch`—mutating program *data* rather than code. In contrast, an $active \rightarrow inactive$ transition is performed by disabling the call instruction, i.e., we disable the call to `cyg_*` function using call toggling or word patching to a 5 byte NOOP sequence. To reactivate the probe-site, the call is toggled or the NOOP sequence is swapped again with the original byte sequence. We cache these byte sequences in probe metadata so that they do not need to recalculated at each probe toggle.

Optimized argument passing The first invocation sets up the probe-specific metadata in a global data structure and

[2] A number of alternatives exist. If the target is a PLT entry—common since the probe libraries are often dynamically linked—we may use the unused linkage bytes there. It is also possible to use a return instruction lying inside the function being patched as the trampoline. This is also a good fall back in the unlikely case that we cannot find space for a trampoline due to the virtual memory being dense in that region.

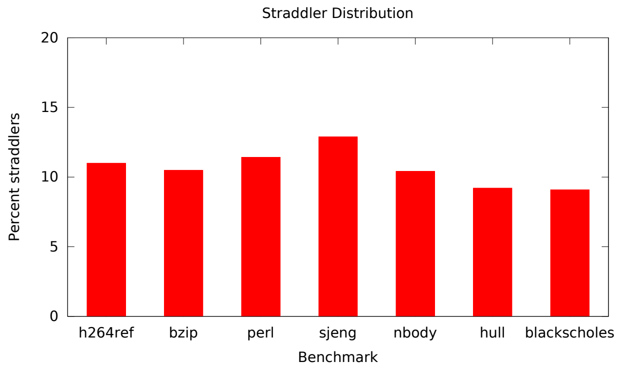


Figure 10. Straddler Distribution

Benchmark	Initialization Cost (%)
h264ref	0.8
bzip	0.01
sjeng	0.1
perl	0.07
nbody	0.05
hull	0.02
blackscholes	0.001

Table 1. Probe initialization cost as % of process runtime for programs in the SPEC CPU 2006 suite.

invokes the discovery callback. Existing `cyg_*` call sites *could* be left as-is, because these calls *already* pass the `call_site_addr` that uniquely identifies the probe. But this calling convention can be improved, so the initialization routine also optionally sets up a fast path for future invocations by injecting the newly generated *probe id* as an argument to the `cyg_*` function in place of the `func_addr`. This modification mutates the call site of the `cyg_*` function (and must use `libwordpatch` to do so). Subsequent invocations use the probe id to do an efficient array lookup—instead of a hash lookup on an address—for retrieving probe metadata, since the generated probe ids are dense. The initialization cost here is incremental since uninvoked probes do not get initialized.

8. Evaluation

In this section we determine T_{max} on a number of x86_64 implementations and then analyze the cost and scalability of individual probe actions for both our library and a couple of competitors. For probe costs we profile microbenchmarks and applications from the SPEC CPU 2006 suite. All parallel applications were run with 16 threads. We used a machine running a Linux 3.19.0-28 kernel on two Xeon E5-2670 CPUs with hyperthreading disabled for the benchmarks unless otherwise stated.

8.1 Validating the Model

The model of Section 3 and patching algorithms of Section 5 require an upper bound, T_{max}, on the duration of wait needed to ensure writes are visible to instruction fetch on other cores. We have developed a stress test to empirically determine T_{max}. The results are shown in Figure 9, and the test uses the following algorithm:

- A *patcher* thread repeatedly activates and deactivates a cache-line-straddling call-site. The call instruction can straddle the cache line boundary in 4 different ways depending on where within the instruction the cache line boundary occurs. We let $S \in \{1..4\}$ indicate the straddling position.
- N *executer* threads to repeatedly execute the patched instruction sequence in a tight loop. We vary N in the range $\{2..6\}$.
- Each variant as defined by N and S is run 5 times for a total of 100 tests.

The test is finished once the patcher thread has toggled the call on and off 50 million times. A test is considered a failure if the program crashes as the result of executing an illegal instruction or a segmentation fault, which result from mixed front and back portions of the instruction.

We evaluate on a selection of Intel x86 microarchitectures: Nahalem (Xeon E7-4830), Sandy Bridge (Xeon E5-2670) and Ivy Bridge (Core I7-3770). We vary wait time, measured in `rdtsc` ticks, over the interval 0 - 2400 in increments of 100. In all single-socket configurations no failures occurred with a wait of 600 ticks (as reported by `rdtsc`) or higher on any of the test systems. Some systems required less wait to stop showing failures, such as the I7-3770 and the E5-4830 that are both failure free from a wait of 400 ticks and upwards.

The dual-socket system shows failures at higher waits, but also hits zero failures and stays there. We use this dual-socket system for all our remaining benchmarks. And, after adding a safety margin, we use a wait time of 3000 on this platform. To determine these parameters, deploying the `libwordpatch` (but not `libcallpatch`) library requires an installation-time benchmark of the system to determine this platform-specific number. If hardware vendors in the future publish more detailed cross-modification specs, that would obsolete this step.

8.2 Probe initialization costs

Probes require a one-time initialization, and these costs must be quantified in order to validate our principle of minimal startup overhead in the critical path. We measured average initialization cost of a probe with a synthetic benchmark consisting of 20,000 probes. On Sandy Bridge the initialization cost is ∼18,000 cycles on average. Next we ran an application benchmark in which every probe site is initialized upon first invocation, but then permanently deactivated. We report probe initialization cost as a percentage of application run-

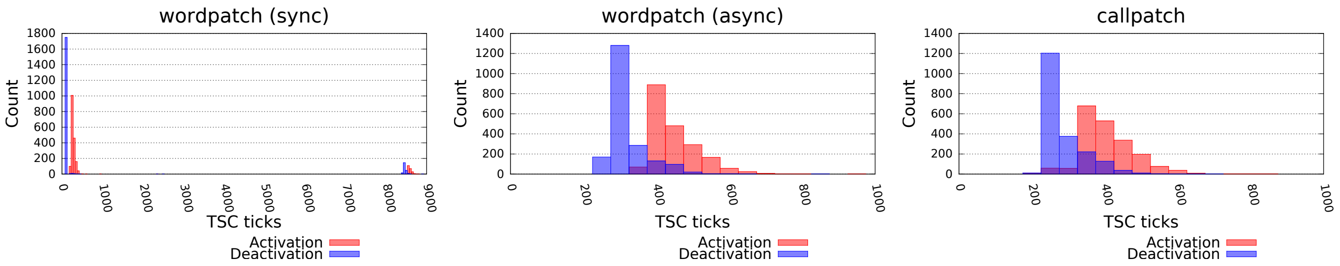


Figure 11. Distribution of probe activation and deactivation latencies for wordpatch and callpatch ∼10% straddlers.

time for SPEC benchmark applications. Table 1 shows that this cost doesn't exceed 1% across all applications, with a geomean of 0.15%. Additionally, due to the on-demand nature of the `-finstrument-functions` probe provider, this cost is not paid up front at once during program initialization time, so the effect on program initialization is minimal.

8.3 Probe activation and deactivation costs

Next we measured the probe toggling cost at the `libfastinst` layer. The cost includes probe metadata lookup in addition to the underlying patching costs of `libwordpatch` and `libcallpatch`, with wait setting of 3000 cycles for wait-based protocols. We generated a synthetic application with large number of instrumented functions (20,000 probes) so that some of the probes would be in straddling positions. Then we ran a probe deactivation pass (all probes starts in active state by default) measuring time for each deactivation call. Next we ran a reactivation pass measuring activation costs. The histograms in Figure 11 summarize individual probe toggle latencies for each of those probe operations. Synchronous probe deactivation displays a bimodal behavior with the expensive mode corresponding to straddler deactivations (with multiple waits in the critical path). Asynchronous patching and callpatch show unimodal, normal cost distributions. The cost of asynchronous patching is low since it pushes the wait costs to a different thread than the calling thread. The probe activation cost also follows a very similar pattern.

The average cost of probe activation/deactivation/invocation are the vital statistics for any probing implementation. We count *invocation cost* as the overhead to invoke an empty probe function at the probe site. In our `-finstrument-functions` probe provider this is the cost of multiple jumps via `cyg_*` trampoline calls as well probe meta data lookup costs within `cyg_*`. Table 2 compares the costs of several dynamic instrumentation methods executing on a single core. We used Java OpenJDK 1.8 with a `VolatileCallSite` for `invokedynamic` results.

One pattern we see is that many solutions—like DynInst and DTrace, but also Intel Pin and others—support efficient invocation once instrumentation is complete, but not rapid activation/deactivation. In order to support applications that rely on high-frequency toggling—like the profiler in Section 9—first, `libfastinst` needs to demonstrate lower constant factors for activation/deactivation, which it does in Table 2. Second, it is necessary to scale to many threads executing probe sites as well as toggling probes. Thus, in the next section, we turn to scalability.

Method	Activation	Deactivation	Invocation
fastinst (sync):			
non straddlers	300	120	35
straddlers	8850	8435	35
fastinst (callpatch)	432	315	35
JVM VolatileCallSite	995		55
JVM MutableCallSite	1432		83
Dyninst	1,929,244	995,447	320
DTrace	∼70,771		1176

Table 2. Average cost, in TSC ticks, of probe operations with several dynamic instrumentation methods. Note that Java doesn't have an explicit notion of *deactivation*, rather deactivation is accomplished by setting the target to a a NOOP-function. DTrace (1.12.1) numbers come from a different machine—a MacBook Pro, mid 2014, because DTrace is still not well supported in Linux—and represent the time it takes to enable an already compiled probe when using `libdtrace`, ordinarily DTrace probes are enabled on the command line, in about a second.

8.4 Scalability

In order measure the effect of probe toggling on hot code we again, as in Section 8.1, use one thread that patches a hot probe site while multiple threads execute. We varied the number of executor threads and toggling frequency while observing the throughput of total function calls through the probe site.

We tested two probe toggling modes with `libfastinst` running on `libcallpatch`. Activate/activate toggling mode replaces an existing function pointer with another, while activate/deactivate mutates the code to remove the call site. As outlined in Figure 12 the throughput remains high and unaffected by toggling through 100K toggles per second. Figure 14 illustrates the parallel scalability of activate/activate and activate/deactive toggling.

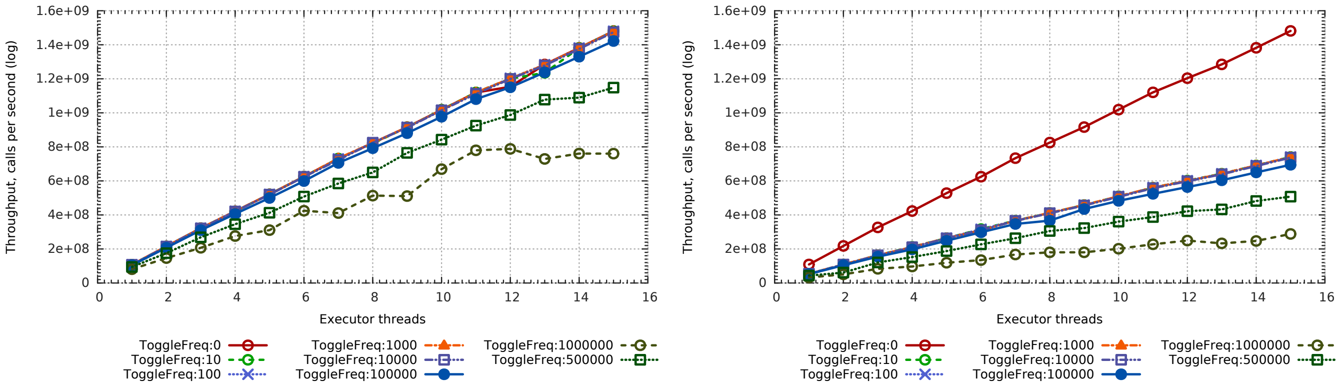


Figure 14. FastInst activate/activate throughput (left) and activate/deactivate throughput (right). Here we see that throughput is affected by toggle rate, but scalability is not. Adding more threads executing the same probe site increases throughput linearly.

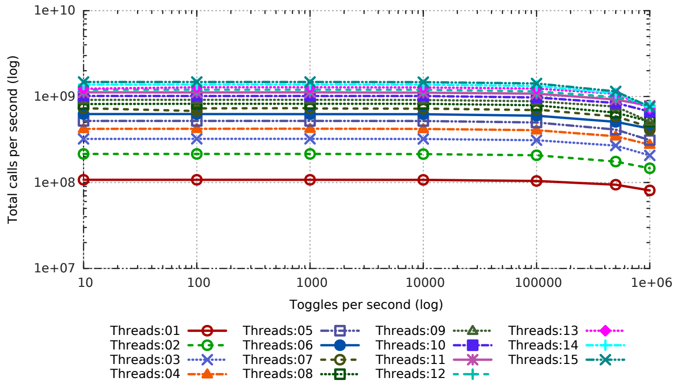


Figure 12. libfastinst: Probe throughput as toggling frequency increases to a million toggles per sec (using `callpatch`). Throughput is unaffected through 100Khz toggle frequency.

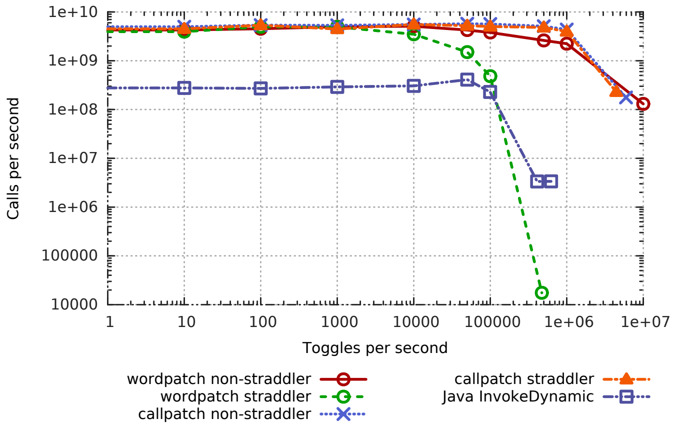


Figure 13. Java `invokedynamic` with a `VolatileCallSite` and our raw `patch` calls exhibit a similar pattern of throughput as frequency of mutation of code memory increases. This shows the deleterious effects of mutating hot code, but this stress test shows toggle rates that are unrealistic even in aggressive instrumentation applications.

Next, in Figure 13, we compared raw `patch` toggling between two calls, compared against Java `VolatileCallSite`, which is a version of `MutableCallSite` with similar semantics to `patch`. All variants show degradation in throughput at sufficiently high frequencies, but wordpatch-straddler and Java fare much worse. Java's `VolatileCallSite` API would appear to be the *only* solution, other than `libfastinst`, we're aware of that is designed to support rapid toggling. The documentation in JDK 8 states that it *"sees updates to its call site target immediately, even if the update occurs in another thread"*. In our experiments, however, the behavior was not consistent with this. Even though throughput, as in Figure 13, is reasonable, the *balance*—how many calls to one probe state or the other—was chaotic. For example, over all the runs in Figure 13, the geometric mean imbalance factor (ratio of the more frequently to less frequently observed state) for Java was 38.7× versus 2.5× for wordpatch and 2.7× for callpatch.

8.5 Latent costs

A deactivated probe with `wordpatch` is a 5 byte NOOP. However a probe deactivated using `callpatch` might contain a relative CALL/JMP instruction as a part of the inactive state if the probe site is a straddler. We compared the increased cost of running a deactivated probe in a loop, compared to an empty loop. The cost on our test platform was 1-2 cycles for straddle points that allow a short relative jump, and 4 cycles for the "1|4" straddle point that requires calling and returning from a trampoline. This is one reason that compilers should avoid straddling probe sites, if possible.

9. Case Study : Sampling Profiler

In order to measure probe overheads "in the wild" in real applications, we developed a custom *latency profiler*. Unlike typical statistical profilers, which sample instants in time, this profiler samples *intervals* of time by instrumenting, e.g., the start and end of a function call. The profiler still uses statistical sampling, turning on and off instrumentation dy-

			slow-	toggle
Benchmark	# toggles/sec	# samples/sec	down %	%
h264ref	24,015	115,526	6	0.1
bzip	2231	10,638	0.6	0.01
sjeng	14,616	73,060	3	0.06
perl	42,276	211,346	11	2
nbody	8052	34,888	2	0.02
hull	1185	5636	0.2	0.005
blackscholes	730	3650	1	0.002

Table 3. Slowdown when running in profiler mode, measured in CPU time. The toggle overhead contribution is the percentage of extra CPU time due to probe toggles. The # samples/sec is the number of profiler library invocations per second via probe sites.

namically. The instrumentation measures the duration of each function call in addition to counting how many times each function has been invoked. When a certain sample size threshold is exceeded the instrumentation self-deactivates. The profiler spawns a daemon thread at program startup which wakes up once per each epoch and activates all the probes that self-deactivated since the last epoch check. The sample size was fixed at 10 and epoch period at 10ms. We used the same SPEC benchmarks from Section 8.2.

First, we collected statistics on the occurrence of straddlers at call sites in the applications considered. This gives an indication on the relative effect of straddler handling protocols on the overhead for each application. As shown in Figure 10, the proportion of straddlers to the total number of patch sites is relatively stable around 10% across the applications.

Next we ran the applications with profiling and measured the slowdown. Table 3 shows that the overhead varies widely though never exceeding 11%, and this holds in spite of applications doing as many as 42,276 probe toggles and 213,346 samples per second. Bzip shows very little overhead potentially due to memory bound nature of the benchmark. The majority of functions in hull are long lived thus the effect of instrumentation is not significant.

We measured toggling related overhead as percentage of application runtime. It stays below 2% for our benchmark applications. Table 3 outlines the results. This profiler is a small prototype meant to demonstrate that dynamic probing can scale to large numbers of probe invocations and probe toggles, even spread across 16 application threads, and with a small effect on application throughput.

10. Related Work

Profiling techniques: Bursty Tracing Our profiler implementation was included here only as a benchmark of rapid-toggling. It is worth noting, however, that this idea of a instrumentation-plus-backoff was introduced almost twenty years ago, although it is rarely, if ever, deployed in modern tools. In 1996, [14] described an adaptive profiling framework that combines the instrumentation and sampling approaches—activating and deactivating profiling dynamically to adjust overhead. Arnold Ryder sampling [3] and "bursty tracing" followed a few years later and expanded on this concept [13]. However, this work predated modern multicore architectures, and did not include a cheap and scalable framework for toggling the instrumentation.

Probe and instrumentation frameworks Throughout this paper, we have mentioned several software systems that can provide dynamic probes, including Intel Pin [12], DynInst [4], DTrace [11], LTTng [8], and SystemTap [23]. Many of these were developed as commercial software, but there have been major academic developments as well, including DynamoRio [6] and the long-running Paradyn/Dyninst project [3]. Linux kernel has supported dynamic probes via kernel level kprobes [23] and user level uprobes [17] for some time. Uprobes uses INT3 breakpoints and requires kernel intervention for enabling probe points. DynInst has evolved substantially over the years and has been used in many contexts. VampirTrace, for example, uses DynInst to inject tracing code that logs program events [18]. Some configurations of DynInst use an in-process agent to accomplish *self-propelled instrumentation* which can follow control flow between processes and even machines [22].

11. Conclusions and Future Work

In this paper we presented algorithms for scalable probe toggling along with an analysis of the low-level performance of toggling that has been absent from previous work in this area. The conclusion is that out-of-place instrumentation is slow to activate, and in-place instrumentation based on traps (and typically involving the kernel) are slow to invoke once activated. We believe that in-place instrumentation with cross-modification is the way forward for scalable probes. We provide an à la carte menu of solutions to achieve this that can work in different software and hardware contexts.

Our work complements existing instrumentation frameworks such as Dyninst that provide general-purpose binary modification. The scalable code-patching and call-patching approach we've described in this paper is a basic building block that could be potentially integrated with these more general approaches. For example, there has already been substantial effort into *deconstructing Dyninst*, into suite of narrowly-focused tools [24]. One prospect in the future would be to equip Dyninst with a highly-restricted version of its binary patching API (a curtailed version of `BPatch`), which could optionally be implemented with the techniques described in this paper. Likewise, probe-focused libraries like DTrace could be further optimized using the technique described in this paper.

[3] Found on the web at: `http://www.dyninst.org/`

Ultimately, we believe better support for toggle-able probes should be provided by ahead-of-time native code compilers, following the lead of the Intel compiler, and in future work we plan to address this gap in one or more major open-source compilers. For example, there is a particular need for instrumentation support in non-C languages such as Rust and Haskell. Where possible, better compiler support can remove the need for probes that start in an active state, or for probes that cross cache line boundaries, but cannot eliminate the need for scalable probe toggling via cross-modification.

References

[1] kpatch: dynamic kernel patching. Technical report.

[2] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 2–14, Seattle, WA, May 1990.

[3] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM.

[4] A. R. Bernat and B. P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 9–16. ACM, 2011.

[5] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the SIGPLAN 2008 Conference on Programming Language Design and Implementation*, Tucson, AZ, June 2008.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 265–275. IEEE, 2003.

[7] B. Daloze, C. Seaton, D. Bonetta, and H. Mössenböck. Techniques and applications for guest-language safepoints. In *Proceedings of the 10th Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems Workshop (ICOOOLPS)*, 2015.

[8] M. Desnoyers and M. R. Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium)*, volume 2006, pages 209–224. Citeseer, 2006.

[9] M. Desnoyers, P. McKenney, A. Stern, M. Dagenais, and J. Walpole. User-Level Implementations of Read-Copy Update. *Parallel and Distributed Systems, IEEE Transactions on*, 23(2):375–382, February 2012.

[10] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing*, 15:399–407, 1992.

[11] B. Gregg and J. Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

[12] K. Hazelwood, G. Lueck, and R. Cohn. Scalable support for multithreaded applications on dynamic binary instrumentation systems. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, pages 20–29, New York, NY, USA, 2009. ACM.

[13] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, pages 117–126, 2001.

[14] J. K. Hollingsworth and B. P. Miller. An adaptive cost system for parallel program instrumentation. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing - Volume I*, Euro-Par '96, pages 88–97, London, UK, UK, 1996. Springer-Verlag.

[15] I. Intel. and ia-32 architectures software developers manual. *Volume 3A: System Programming Guide, Part*, 1, 64.

[16] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. Cmp $ im: A pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, pages 28–36, 2008.

[17] J. Keniston, A. Mavinakayanahalli, P. Panchamukhi, and V. Prasad. Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps. In *Linux Symposium*, page 215, 2007.

[18] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008.

[19] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[20] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[21] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. In *Conference Record of the Thirty-Second ACM Symposium on Principles of Programming Languages*, Long Beach, CA, January 2005.

[22] A. V. Mirgorodskiy and B. P. Miller. Diagnosing distributed systems with self-propelled instrumentation. In *Middleware 2008*, pages 82–103. Springer, 2008.

[23] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and J. Chen. Locating system problems using dynamic instrumentation. In *2005 Ottawa Linux Symposium*, pages 49–64. Citeseer, 2005.

[24] G. Ravipati, A. R. Bernat, N. Rosenblum, B. P. Miller, and J. K. Hollingsworth. Toward the deconstruction of dyninst. Technical report, Technical Report, Computer Sciences Department, University of Wisconsin, Madison (ftp://ftp. cs. wisc. edu/paradyn/papers/Ravipati07Symta bAPI. pdf), 2007.

[25] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro*, (4):65–79, 2010.

[26] T. B. Schardl, B. C. Kuszmaul, I. Lee, W. M. Leiserson, C. E. Leiserson, et al. The cilkprof scalability profiler. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*, pages 89–100. ACM, 2015.

[27] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[28] S. Wallace and N. Bagherzadeh. Modeled and measured instruction fetching performance for superscalar microprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 9(6):570–578, June 1998.