ParaLog: Enabling and Accelerating Online Parallel Monitoring of Multithreaded Applications

Evangelos Vlachos,[1] Michelle L. Goodstein,[1] Michael A. Kozuch,[2] Shimin Chen,[2]
Babak Falsafi,[3] Phillip B. Gibbons,[2] and Todd C. Mowry[1]
[1]Carnegie Mellon University [2]Intel Labs Pittsburgh [3]École Polytechnique Fédérale de Lausanne

evlachos@ece.cmu.edu, {mgoodste, tcm}@cs.cmu.edu,
{michael.a.kozuch, shimin.chen, phillip.b.gibbons}@intel.com, babak.falsafi@epfl.ch

Abstract

Instruction-grain lifeguards monitor the events of a running application at the level of individual instructions in order to identify and help mitigate application bugs and security exploits. Because such lifeguards impose a 10–100X slowdown on existing platforms, previous studies have proposed hardware designs to accelerate lifeguard processing. However, these accelerators are either tailored to a specific class of lifeguards or suitable only for monitoring single-threaded programs.

We present ParaLog, the first design of a system enabling fast online parallel monitoring of multithreaded parallel applications. ParaLog supports a broad class of software-defined lifeguards. We show how three existing accelerators can be enhanced to support online multithreaded monitoring, dramatically reducing lifeguard overheads. We identify and solve several challenges in monitoring parallel applications and/or parallelizing these accelerators, including (i) enforcing inter-thread data dependences, (ii) dealing with inter-thread effects that are not reflected in coherence traffic, (iii) dealing with unmonitored operating system activity, and (iv) ensuring lifeguards can access shared metadata with negligible synchronization overheads. We present our system design for both Sequentially Consistent and Total Store Ordering processors. We implement and evaluate our design on a 16 core simulated CMP, using benchmarks from SPLASH-2 and PARSEC and two lifeguards: a data-flow tracking lifeguard and a memory-access checker lifeguard. Our results show that (i) our parallel accelerators improve performance by 2–9X and 1.13–3.4X for our two lifeguards, respectively, (ii) we are 5–126X faster than the time-slicing approach required by existing techniques, and (iii) our average overheads for applications with eight threads are 51% and 28% for the two lifeguards, respectively.

Categories and Subject Descriptors C.0 [*General*]: [System Architectures]; D.2.5 [*Software Engineering*]: Testing and Debugging—Monitors

General Terms Design, Performance, Reliability, Security

Keywords Online Parallel Monitoring, Hardware Support for Debugging, Instruction-grain Lifeguards

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

1. Introduction

Given the industry-wide shift to chip multiprocessors (CMPs), the path to high performance in the immediate and foreseeable future is through parallel processing. Perhaps the key limitation to exploiting this raw performance potential is the ability of programmers to successfully write parallel software. Unfortunately, parallel programming is a notoriously difficult task: history has taught us that as difficult as it is to avoid software bugs in single-threaded[1] code, bugs are even more problematic in multithreaded (parallel) code.

There is a large body of work on tools that help diagnose and fix software bugs, including tools that run *before* [3, 11, 13], *during* [2, 12, 21, 31, 37], and *after* [27, 45, 46] software execution. Many of these tools are complementary, and each has its own strengths and weaknesses. One particularly promising class of tools is *instruction-grain lifeguards* [29,31,32,35,37]: these tools run dynamically (online) with the application, performing sophisticated instruction-by-instruction analysis in software to identify bugs and sometimes even repair them (or limit their damage). Compared with static tools that analyze code before it executes [3,11,13], lifeguards typically report fewer false-positives, because they can directly observe the monitored application's dynamic behavior (e.g., pointers, control flow, run-time inputs, etc.). Compared with post-mortem tools that analyze code after it crashes [27, 45, 46], lifeguards may be able to capture software bugs earlier (i.e. before a crash) and more accurately (based upon instruction-grain dynamic behavior from the start of execution, and not just from a recent window of activity).

While instruction-grain lifeguards offer compelling advantages, their main disadvantage is run-time overhead. Fortunately, we recently demonstrated [5, 6] that in the case of monitoring *single-threaded* applications, a set of hardware-based lifeguard accelerators can dramatically improve performance to the point where the remaining overheads are relatively modest. Namely, the overhead drops from 3–5X to 1.02–1.5X for a number of diverse lifeguards. In this study, we explore how this accelerator framework can be extended to efficient parallel monitoring of *multithreaded* (parallel) applications.

1.1 Challenges in Online Parallel Monitoring of Multithreaded Applications

Today, the only practical way to use instruction-grain lifeguards to monitor a parallel application is to time-slice the multiple application threads onto a single processor and analyze the resulting interleaved instruction stream sequentially. As we observe later in our experimental results, the performance of this state-of-the-art approach is unacceptably poor because neither the application nor

[1] In this paper, we use "single-threaded" as a shorthand for any execution with at most one active application thread at a time.

the lifeguard can enjoy parallel speedups. To enable both the application and the lifeguard to operate in parallel with high performance, we must overcome three key challenges: (i) adapting the three hardware accelerators from our earlier study [5, 6] to work for multithreaded applications, (ii) capturing inter-thread data dependences between the application threads and reproducing their effects appropriately in the lifeguard processing, and (iii) ensuring the multiple lifeguard threads atomically access their shared view (called *metadata*) of the application's state at negligible cost.

Parallelizing Lifeguard Hardware Accelerators. The three lifeguard accelerators described in our earlier study [5, 6] are *inheritance tracking*, *idempotency filters*, and *metadata-TLBs*. These accelerators, which are applicable to stream-based lifeguard frameworks such as LBA [4] and DISE [8], are a key enabler for online, instruction-grain monitoring. Each showed significant performance gains, and in combination they improved lifeguard performance by a factor of 2–3X when monitoring single-threaded applications. However, because of *remote conflicts* by other threads, the three accelerators require significant modifications to monitor multithreaded applications correctly.

Capturing and Enforcing Inter-Thread Data Dependences. A second major challenge in using lifeguards to monitor multithreaded software is capturing the interactions between application threads and translating them into appropriate synchronization amongst the parallel lifeguard threads. While there has been a significant body of work on capturing inter-thread dependences for the sake of deterministic replay and post-mortem analysis [14,18,22,26,27,45,46], our requirements for lifeguard analysis are somewhat different. For example, our key metric is *end-to-end performance* of the online lifeguard analysis (as opposed to storage size of the compressed log). In addition, as highlighted later in the paper by our accelerator modifications, we must also solve the problem of *logical races*—inter-thread effects that are not reflected in coherence traffic (e.g., a race between a memory access and a `free()`).

Ensuring Metadata Access Atomicity. The third major challenge in using parallel lifeguards to monitor multithreaded software is that the lifeguard threads maintain shared metadata corresponding to the application's state, and these metadata must be atomically read and updated by the parallel threads. Because nearly every application instruction may dictate that a lifeguard read and/or update these metadata, the cost for this atomicity must be nearly free.

1.2 Related Work

While there have been a number of hardware proposals for accelerating specific classes of lifeguard analysis algorithms [9, 10, 39, 41,42,47], our goal is to provide a flexible framework that supports *parallel* execution for a diverse range of lifeguards.

Chung *et al.* [7] enhances DBI with transactional memory for monitoring multithreaded applications. It shows that software transactional memory (STM) incurs 41% more overhead than the DBI overhead. In other words, given typical DBI slowdowns of 10-100X, STM incurs another 4.1-41X slowdowns. Nagarajan and Gupta [25] proposes enhancing DBI with hardware modifications to cache coherence so that an application operation and its corresponding lifeguard handler may be executed atomically. Similar to Chung *et al.* [7], this work suffers from the high overhead of DBI, reporting 5-10X overhead.

Regarding capturing and enforcing inter-thread dependences, while our approach builds upon the central insight of previous designs [14, 18, 20, 22, 26, 27, 34, 45, 46] that coherence traffic can help us track inter-thread dependences, our final design is optimized for the requirements of lifeguards, as described later in this paper, including the need to handle *logical races*. Our solution to the logical race problem uses a mechanism similar to one used in Capo [23] for a completely different purpose (i.e., avoiding monitoring during system calls).

Concurrent with this work, Goodstein *et al.* [16] explored an alternative approach to parallel monitoring that requires little or no hardware support, but which can result in some number of false positives.

Finally, there has also been work on parallelizing a lifeguard that is monitoring a single-threaded application [33, 36]; these approaches are orthogonal and complementary to our focus on monitoring multithreaded applications.

1.3 Contributions

This paper makes the following research contributions:

- We present ParaLog, the first design of a system enabling fast online parallel monitoring of multithreaded parallel applications. Our design, which builds upon LBA [4], enables software-defined lifeguards to be ported from single-threaded to multithreaded monitoring with minimal effort.
- We demonstrate how the three hardware accelerators that we proposed earlier [5,6] as key enablers of online single-threaded monitoring can be modified to support online multithreaded monitoring. We present our system design (including its accelerators) for both Sequentially Consistent and Total Store Ordering processors.
- We identify the problems of *logical races*, *remote conflicts*, and *metadata access atomicity*, and present novel solutions based on techniques we call *conflict alerts*, *delayed advertising*, and *synchronization-free fast paths*.
- We implement and evaluate our design on a 16 core simulated CMP, using benchmarks from SPLASH-2 and PARSEC and two lifeguards: a data-flow tracking lifeguard and a memory-access checker lifeguard. Our results show that (i) our parallel accelerators improve performance by 2–9X and 1.13–3.4X for our two lifeguards, respectively, (ii) we are 5–126X faster than the time-slicing approach required by existing techniques, and (iii) our average overheads for applications with eight threads are 51% and 28% for the two lifeguards, respectively.

2. Background: Online Sequential Monitoring

This section presents background material on instruction-grain lifeguards, general-purpose (sequential) lifeguard platforms, and hardware accelerators for sequential monitoring.

Instruction-Grain Lifeguards. Instruction-grain lifeguards often have **common structures**, as observed by previous works [5, 6, 30, 31]. First, a lifeguard often maintains a piece of state (called *metadata* or *shadow state*) for every memory location (and/or register) in the application. Second, if executed application instructions are regarded as *events*, then the lifeguard is composed of a set of *event handlers* that are triggered by certain application events. Third, lifeguard event handlers usually center around the metadata: updating the metadata as a result of an event, or reading the metadata for invariant checking.

For example, the TAINTCHECK lifeguard detects memory-overwrite-based security exploits (e.g., buffer overflow attacks) by monitoring suspect data in the application's address space [32]. It maintains for every memory location a 1-bit "tainted" state, which is initialized to untainted. Unverified input data, such as those from the network or from untrusted disk files, are marked as tainted. TAINTCHECK tracks the propagation of tainted data: For each executed application instruction, TAINTCHECK computes the tainted state of the destination of the instruction by performing a logical OR operation on the tainted states of all the source operands.

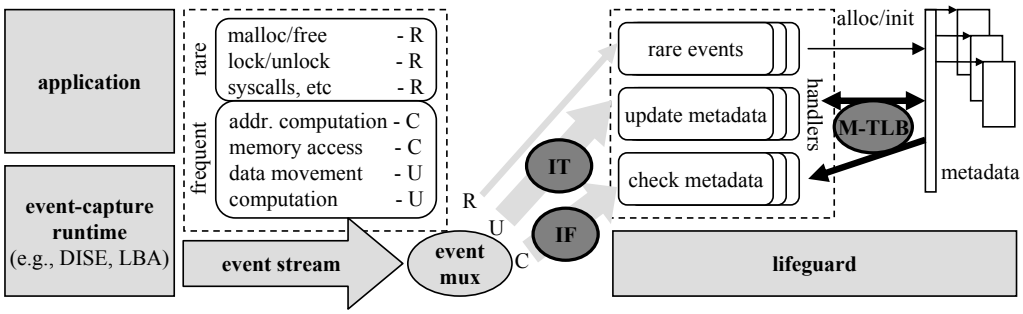


Figure 1. General-purpose online sequential monitoring platform. The left and right halves may run on different cores (as in LBA).

TAINTCHECK reports a security violation if tainted data are used in critical ways, such as in jump target addresses or in the format strings of printf-like calls.

General-Purpose Sequential Lifeguard Platforms. In software-only solutions, lifeguards are implemented on top of dynamic binary instrumentation (DBI) systems [2, 21, 31]. However, these *software-only solutions often incur 10–100X slowdowns* [28, 31, 40]. The large slowdowns result from two orthogonal types of overheads: (i) the DBI cost, such as extracting and delivering instruction events; and (ii) the overhead of lifeguard event handlers, which are invoked on (nearly) *every* application instruction. Because software-only solutions are far too slow, several recent studies exploit hardware designs that address these overheads. In this paper, we are mainly interested in general-purpose hardware solutions, which efficiently support a wide range of lifeguards, as opposed to hardware proposals that focus only on specific lifeguards [9, 10, 38, 42, 47, 48]. We believe that both flexibility and efficiency are important for making an impact on future processors.

Figure 1 depicts a general-purpose lifeguard platform for monitoring sequential applications [6]. On the left, as the monitored application executes, the event-capture hardware extracts information of dynamic instructions and creates a stream of *event records*. The dashed box illustrates examples of application events. On the right, the event delivery hardware retrieves records from the event stream, and issues lifeguard *event handlers* that are registered by the lifeguard software for the corresponding event types. These hardware components significantly reduce overhead (i) above. However, the remaining overhead is still significant (e.g., 3–5X slowdowns [4]).

Three Hardware Accelerators for Sequential Monitoring. To reduce overhead (ii) above, in an earlier study [5, 6] we identified three common sources of lifeguard event handlers' overhead by studying a wide range of lifeguards, and proposed three hardware accelerators for addressing them, as shown in Figure 1. First, propagation tracking (e.g., in TAINTCHECK), which often requires frequent metadata updates, is accelerated by *Inheritance Tracking (IT)*. IT tracks the inherits-from information for application registers in hardware, significantly reducing the number of delivered metadata update events. Second, invariant checks using metadata can be very frequent. However, if the metadata have not been changed since a previous check, a later check on the same metadata can be idempotent. *Idempotent Filters (IF)* cache a small number of recently seen checks to filter out idempotent (thus redundant) checks. Third, metadata address computation is performed in almost all handlers, and may cost more than half of the total instructions in a simple handler. *Metadata-TLB (M-TLB)* provides a fast lookup table to speed up this operation. These three accelerators are useful for a wide variety of lifeguards [6], including those detecting memory-access violations [29,31], data races [37], and security exploits [32, 35].

In our earlier work [5, 6], we implemented the three hardware accelerators in a Log Based Architectures (LBA) lifeguard platform. LBA augments every core in a chip multiprocessor with lifeguard monitoring hardware, including event capture, event delivery, and hardware accelerators, as shown in Figure 1. Given users' relative preferences of performance, power, and correctness, lifeguard monitoring can be dynamically enabled. A monitored application, which must be single-threaded, runs on one core, and the lifeguard runs on another core. The event stream is instantiated as a log buffer (e.g., 64KB) in the last level on-chip cache. Compression techniques can successfully reduce the average size of an event record to less than 1 byte [4]. If the log buffer is full, then the application core stalls; if the log buffer is empty, then the lifeguard core stalls. LBA supports damage containment at the system call boundary by stalling the application at (specified) system calls until the lifeguard finishes processing/checking the remaining records in the log buffer. Combining the three hardware accelerators, LBA achieves lower than 50% slowdowns for a number of diverse lifeguards.

Naturally, the constraint that the framework described above is unable to handle multithreaded applications presents a serious limitation in modern multicore environments.

3. Online Parallel Monitoring: Design Overview

In this paper, we propose ParaLog, a general-purpose platform for online monitoring of multithreaded parallel applications, as shown in Figure 2. Compared to Figure 1, components with new features for parallel monitoring are highlighted. Our goal is to support correct and efficient parallel monitoring while minimizing the lifeguard developers' efforts to port a single-threaded lifeguard. To achieve this goal, our design addresses the following three major challenges.

Application Event Ordering. Every application thread is monitored by a corresponding lifeguard thread. To reduce delays, each lifeguard thread greedily processes its application thread's event stream asynchronously with respect to other event streams, except where necessary to preserve lifeguard correctness. Specifically, because multiple application threads share the same address space, their data sharing and synchronization activities will result in dependences among instruction events. To maintain the correct view of the application's state, the lifeguard must process application events in an order that reflects these dependences. In TAINTCHECK, for example, if one application thread taints a memory location that is read by another application thread, the corresponding lifeguard threads must process the former event prior to the latter event, in order to ensure lifeguard correctness. We propose two new hardware components that capture event orders at the application side, and enforce event orders at the lifeguard side, respectively.

Our *order capturing component* is inspired by previous work on system-wide deterministic replay in multi-processor systems [18,

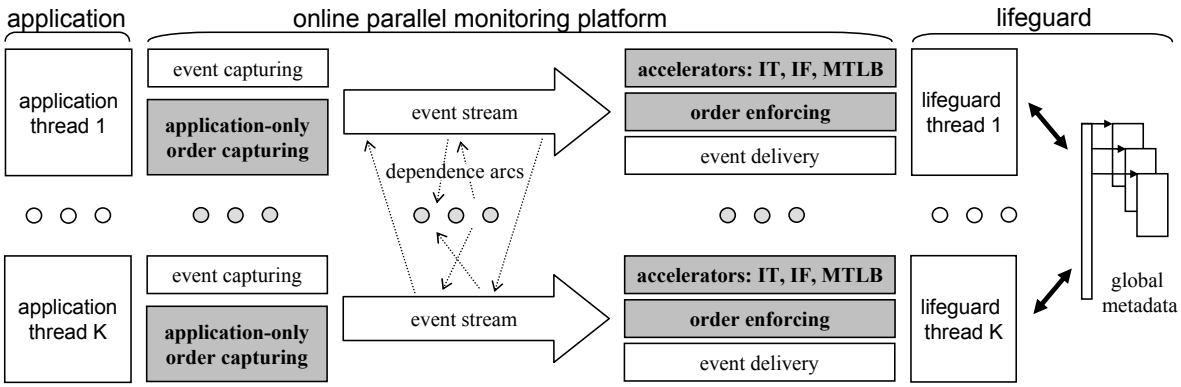


Figure 2. General-purpose online parallel monitoring platform, highlighting components with new features.

22, 26, 27, 45, 46]. It captures event dependences by observing and augmenting cache-coherence messages, and records inter-thread dependence arcs in the event streams, as shown in Figure 2. In particular, the starting point (thread ID, record ID) of a dependence arc is recorded in the event stream associated with the receiving end of the arc. Unlike most previous studies, our design focuses on order capturing at the application level within a monitored application, rather than for the entire system, for two reasons: (i) user-mode lifeguard code should not have visibility into OS kernel state; and (ii) multiple separate applications can be monitored at the same time with little interference. However, it entails that dependence arcs related to OS kernels will be lost. We describe our mechanism to compensate for this in Section 5.4. While we primarily focus on detailed FDR-style [45] dependence capturing, our results show that less detailed dependence capturing can reduce hardware complexity with only a minimal loss in lifeguard performance.

Our *order enforcing component* enforces the captured arcs, without modification to the lifeguard-specific code. If there is a dependence arc from event i in event stream t to event i' in event stream t', the order enforcing components collaborate to ensure that only after lifeguard thread t finishes processing event i can event i' be delivered to lifeguard thread t'. Thus, each lifeguard thread can safely process its delivered event stream, without concerns with inter-thread application dependences. The basic idea is as follows, illustrated with the above example. The order enforcing component for any lifeguard thread t publishes globally a progress indicator, $progress_t$, indicating the record ID of the last event record processed by t. Because of the above dependence, event i' has an associated dependence record (t, i). Upon seeing dependence (t, i), the order enforcing component for thread t' reads $progress_t$, and only delivers event i' if $progress_t > i$. If the condition is false, a generic "dependence stall" event is delivered to thread t' instead. We will describe more details in Sections 5.1 and 5.2, including how to efficiently implement the globally advertised progress indicators.

Moreover, we mainly assume the Sequential Consistency (SC) memory model in our discussions. However, our solution can be extended to Total Store Ordering (TSO), as will be detailed in Section 5.5.

Accurate Asynchronous Analysis. As mentioned above, each lifeguard processes asynchronously its application thread's event stream, with respect to other lifeguards, but also with respect to the execution of the application. While the event transport mechanism introduces a delay between fault exercise and error detection, we contain the impact of any faults or vulnerabilities in the application space by enhancing the OS to suspend the application threads on system call boundaries and permitting execution to continue only after the lifeguard determines that the system call inputs are safe, similar to [6].

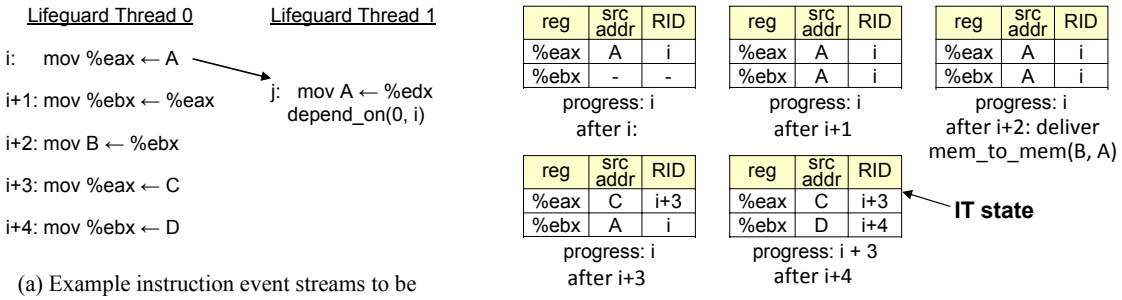
Metadata Access Atomicity. As discussed in Section 2, lifeguards typically maintain metadata state for every memory location in the application's address space. In parallel monitoring, this means that all lifeguard threads share a global data structure containing metadata, as shown in Figure 2. Therefore, all metadata accesses must be properly synchronized. Note that metadata accesses are performed in nearly all lifeguard event handlers. A naive approach is to apply synchronization primitives to all metadata accesses. However, frequent lifeguard event handler code paths are typically composed of only a few instructions (e.g., less than ten instructions) [6], while an atomic x86 instruction (e.g., `xchg`) locks the off-chip memory bus, often requiring over a hundred cycles. Thus, such a naive approach can lead to prohibitively high overhead. Fortunately, our design for capturing and enforcing dependence arcs provides the key ingredients for avoiding synchronization overheads. This is detailed in Section 5.3, where we show that for a wide range of lifeguards (as characterized by properties we define), no explicit synchronization is required to preserve metadata access atomicity in an event handler's frequent "fast path" (*synchronization-free fast paths*).

Local Hardware Accelerators with Possible Remote Conflicts. The hardware accelerators all maintain monitoring states: Inheritance Tracking (IT) keeps inherits-from information, Idempotent Filters (IF) cache recently seen check events, and Metadata TLB (M-TLB) caches frequently used metadata page mappings. Note that some events may conflict with (i.e., invalidate) an accelerator's state. For example, a malloc/free event may significantly change metadata, thus conflicting with an accelerator's state. Moreover, if A is an inherits-from address recorded in IT (implying that a register's metadata is inherited from A's current metadata), then any instruction that overwrites A is a conflicting event (because it may change A's metadata) [6]. In the sequential setting, an accelerator is able to observe all the events and detect all the conflicts. It flushes or invalidate its state upon seeing a conflicting event.

However, detecting conflicts is a major challenge in the parallel setting. As shown in Figure 2, an accelerator can only observe events local to a lifeguard thread, while a remote event may be conflicting. In Section 4, we propose two mechanisms (*delayed advertising* and *conflict alerts*) for dealing with remote conflicts by exploiting the existing dependence arcs and employing a broadcasting mechanism, respectively.

4. Accelerating Online Parallel Monitoring

Hardware accelerators are essential to achieve low overhead lifeguard monitoring, as described in Section 2. The three hardware ac-



(a) Example instruction event streams to be consumed by two lifeguard threads

(b) IT state and event delivery at thread 0 (RID: Record ID)

Figure 3. Handling instruction-level remote conflicts for IT, by setting *progress* to be the minimum RID in the table.

celerators (IT, IF, M-TLB) were shown to improve the performance of a number of interesting lifeguards [5, 6]. However, these techniques do not immediately translate to the parallel world. In Section 4.1, we analyze the hardware accelerators to illustrate the major challenge: efficiently detecting and handling remote conflicts. We describe two mechanisms that target instruction-level remote conflicting events and high-level remote conflicting events in Section 4.2 and 4.3, respectively. In Section 4.4, we describe how to support IT, IF, and M-TLB with these mechanisms.

4.1 Key Challenge in Accelerating Parallel Monitoring: Remote Conflicts

The three hardware accelerators all maintain monitoring states for accelerating lifeguard operations. For correctness, all or parts of monitoring states must be flushed or invalidated upon certain conflicting events; otherwise the monitoring states may be corrupted. In a sequential setting, all the events are seen by an accelerator and therefore it is able to detect conflicts locally. However, in parallel monitoring, an accelerator can only see events local to a lifeguard thread, thus posing the challenge of efficiently handling instruction-level and high-level remote conflicts.

IT (Inheritance Tracking). Figure 3 depicts a scenario where monitoring correctness of TAINTCHECK is violated because of an instruction-level remote conflict for IT. As shown in Figure 3(a), two event streams are being consumed by two lifeguard threads. Figure 3(b) shows the monitoring states maintained in hardware at thread 0. We begin by describing how IT works. Let us focus on event stream 0 and ignore the RID-related fields. As described in Section 2, TAINTCHECK maintains 1 tainted bit per memory location and tainted state for every register of the application. It tracks the propagation of tainted states and detects critical uses of tainted locations as security violations. Therefore, for every event in event stream 0, TAINTCHECK must perform a corresponding metadata operation. For example, for event i, metadata of A must be copied to metadata of $\%eax$. Similarly, for event $i+2$, metadata of $\%ebx$ must be copied to metadata of B. It is easy to see that events i to $i+2$ actually propagate tainted state from A to B. Without IT, the event delivery component invokes a TAINTCHECK event handler for every event, performing the above metadata operations. IT aims to reduce the number of delivered events by tracking inherits-from addresses for all the application registers. There is one row per register in the hardware IT table. For simplicity, Figure 3(b) shows only the relevant IT table rows. After i, IT records A in row $\%eax$, without delivering the actual event to TAINTCHECK. After $i+1$, the IT row $\%eax$ is copied to row $\%ebx$, also without event delivery. After $i+2$, IT finds out that A is copied to B, and delivers to TAINTCHECK a single memory to memory copy event, `mem_to_mem(B,A)`. In this way, IT delivers a smaller number of events instead of one for each record, significantly reducing the amount of work for the lifeguard. However, because the read of metadata associated with A is delayed, the system must compensate for potential conflicting events that intervene.

As shown in Figure 3(a), j represents a conflicting event: it overwrites an inherits-from location recorded in the IT table. If event j were delivered before $i+2$, then TAINTCHECK would overwrite the metadata state for A. By the time that `mem_to_mem(B,A)` were delivered, the original metadata state for A would have been lost and TAINTCHECK would copy A's new state to B's metadata. This is a wrong operation! In a sequential setting, this problem is solved by checking every store address against all the recorded inherits-from locations for conflicts [6]. Upon detecting a conflict, the affected IT rows are flushed: delivered to the lifeguard then cleared. For example, in the above conflicting scenario, event i and $i+1$ will be delivered before j, leading to correct TAINTCHECK operations. However, this proposed solution is incomplete for parallel monitoring because IT at thread 0 does not see event j at thread 1.

Besides instruction-level conflicts, a lifeguard may require flushing the IT table upon high-level events (such as `malloc/free`). An example is MEMCHECK [31], a memory checking lifeguard that tracks the propagation of the initialized states of memory locations [31]. Therefore, IT requires handling of both instruction-level and high-level remote conflicts.

IF (Idempotent Filters). IF caches recently seen check events, whose types are configurable by lifeguards. If an incoming event hits in the IF cache, the event is regarded redundant and not delivered to the lifeguard. IF cache entries are invalidated upon certain events that are also configurable. For example, ADDRCHECK [28] is a lifeguard that checks whether the memory location in every read/write has been allocated. Two checks to the same address are idempotent if there is no `malloc/free` in between. Therefore, IF can be configured to cache memory events, and invalidated upon high-level `malloc/free` events. Moreover, in general, it is possible that instruction-level events that change metadata states may also trigger IF cache invalidation. Therefore, IF also requires handling of both instruction-level and high-level remote conflicts.

M-TLB (Metadata TLB). M-TLB uses a fast hardware lookup table to maintain the most frequently used mappings between application virtual page addresses and metadata virtual page addresses. Given an application data address in an event, the metadata address can be efficiently computed using M-TLB. Typically, a lifeguard dynamically allocates metadata pages when the corresponding data addresses are first used (e.g., by `malloc`). Simple lifeguard implementations do not need to worry about de-allocating metadata pages; they simply mark the metadata states to be invalid. In such cases, M-TLB entries will always be valid. However, more sophisticated lifeguard implementations may de-allocate metadata pages for saving system space. De-allocations can only happen after high-

level application events, such as `free`. In this situation, remote high-level events may conflict with M-TLB states.

4.2 Delayed Advertising for Instruction-Level Remote Conflicts

We make the following observations regarding the relationship of dependence arcs and instruction-level remote conflicts. If an accelerator keeps monitoring state related to event i and a remote event j conflicts with this state, event i and event j must access the same application address and one of them must be a write. Therefore, event i and event j must be ordered by dependence arc(s), as illustrated by Figure 3. However, event j may not be ordered relative to the later event (e.g., $i + 2$) that manifests the remote conflict. To make matters worse, dependence arcs are recorded at the receiving ends because of the nature of the order capturing mechanism. As a result, an accelerator may never be aware of the dependence event that causes the conflict.

Our solution exploits the enforced order for the dependence arc(s) from i to j. As described in Section 3, the order enforcing component for a lifeguard thread t publishes globally $progress_t$, indicating the record ID of the last event processed by t. In the example of Figure 3, thread 1 checks the progress of thread 0 by reading $progress_0$, and delivers event j only if $progress_0 > i$. Therefore, we can control thread 1's progress by delaying the advertising of $progress_0$. Intuitively, if the monitoring state related to event i is kept in an accelerator, we can regard event i as still being processed. Only after the state of i is no longer kept by accelerators do we report globally that all lifeguard processing related to event i completes.

Our solution is illustrated in Figure 3(b). We add an RID field for every IT table row to record the RID associated with the inherits-from address. The field is copied along with other fields for a register-to-register copy operation (e.g., $i + 1$). The progress indicator $progress_0$ is computed as the minimum of all the recorded RIDs. In this way, thread 1 will deliver j only after $i + 4$, thus successfully avoiding the remote conflict from causing any problem.

Once an RID is recorded in an IT row of register R, it stays until the row is overwritten or flushed under one of the following situations. First, register R is overwritten in a later event. Second, because we continue to monitor local conflicts as in the sequential setting, if a local event conflicts with the row of R, then we flush the row by delivering an event to the lifeguard. Third, in case of a dependence stall (waiting for a remote thread), we flush all the IT entries, and publish an accurate *progress*. In this way, we guarantee that there is no deadlock resulting from delayed advertising. Finally, we support an optional threshold. If $progress_t$ for a lifeguard thread t is smaller than the record ID of the last event processed by t by more than the threshold, we forcefully flush IT entries to refresh $progress_t$.

Regarding the computation of the minimum of the RIDs in the IT table, this operation is not in the critical path. Because our purpose is to delay the delivery of event j, it is always correct to report an older (smaller) minimum as *progress*. Therefore, we just need a reasonably fast mechanism for this computation. Moreover, we can reduce the frequency of the computation by checking to see whether or not the IT table row containing the previous minimum RID has been overwritten or flushed.

4.3 *ConflictAlert* Messages for High-Level Remote Conflicts

High-level remote conflicts have different characteristics from instruction-level remote conflicts. Consider the case of a `free` high-level event F and a memory read event M that accesses a location in the freed address range. If an accelerator maintains monitoring states related to M, then F may be a conflict (depending on the lifeguard). However, unlike instruction-level remote conflicts, F may not be ordered relative to M by any dependence arc. The `free` library call often creates free block information close to the boundaries of the address range, while M may access a location far away from the boundaries. Therefore, there may be no cache coherence messages linking the two events although F and M are logically related, creating a *logical race*. In such cases, the solution in Section 4.2 does not apply. On the other hand, high-level events are much less frequent than instruction-level events, opening new opportunities for solving the problem.

We propose to employ a mechanism that broadcasts alert messages for specified high-level events, called *ConflictAlert* messages. The basic idea is as follows. The event capturing component at the application side provides the mechanism for sending *ConflictAlert* messages. Consequently, a customized wrapper library can intercept important high-level library calls and broadcast *ConflictAlert* messages. Upon receiving such a broadcast message, the event capturing component associated with each executing thread creates a record in its event stream. At the lifeguard side, such a *ConflictAlert* record triggers invalidation or flushing of accelerator states. Our design enables the lifeguard to configure the set of interesting high-level events at lifeguard initialization time. The details of the *ConflictAlert* mechanism will be described in Section 5.4.

4.4 Supporting Three Hardware Accelerators

As discussed in Section 4.1, in general, IT and IF require handling both instruction-level and high-level remote conflicts, while M-TLB requires handling only high-level remote conflicts. For IT and IF, we employ Delayed Advertising for avoiding instruction-level remote conflicts from corrupting metadata states. For IT, IF, and M-TLB, we employ *ConflictAlert* messages for high-level remote conflicts. Upon seeing a *ConflictAlert* record, IT may flush the IT table according to the configuration specified by the lifeguard. Likewise, IF may invalidate its cache, and M-TLB may flush its fast lookup entries.

5. Capturing and Enforcing Event Ordering

To handle fine-grain dependence events, ParaLog makes use of cache-coherence messages. However, it differs from FDR [45] and related work in that (i) dependences are tracked per-application rather than system-wide, (ii) the sequence of dependence events are gathered per application-thread rather than centralized, and (iii) the dependence information is consumed *online* (ideally on other cores from the same die as the application cores). While recent work [23] investigated property (i) in the context of special hardware, we assume a conventional cache-coherent multicore processor.

5.1 Capturing Fine-Grain Application Dependence Events

The event capturing components on which the monitored application runs (Figure 2) collect information on every dynamic *event* in the monitored application, including retired instructions, system calls, and important library calls (such as `malloc/free`) as in [5, 6]. We extend this design with an *order capturing component* to record happened-before dependences exposed by cache coherence activities, as depicted in Figure 4(a).

Similar to FDR, we augment each processor core with a counter that is incremented by one when an instruction/μop retires. (This counter is used as the aforementioned record ID for the corresponding event.) We also augment every L1 cache block with a field to record the counter value when the cache block was last accessed. The counter value is piggy-backed onto cache coherence messages. In our system, however, these counter values are $(thread\_id, counter)$ tuples, and the processor maintained tuple is saved and restored by the OS during context-switch events. Further,

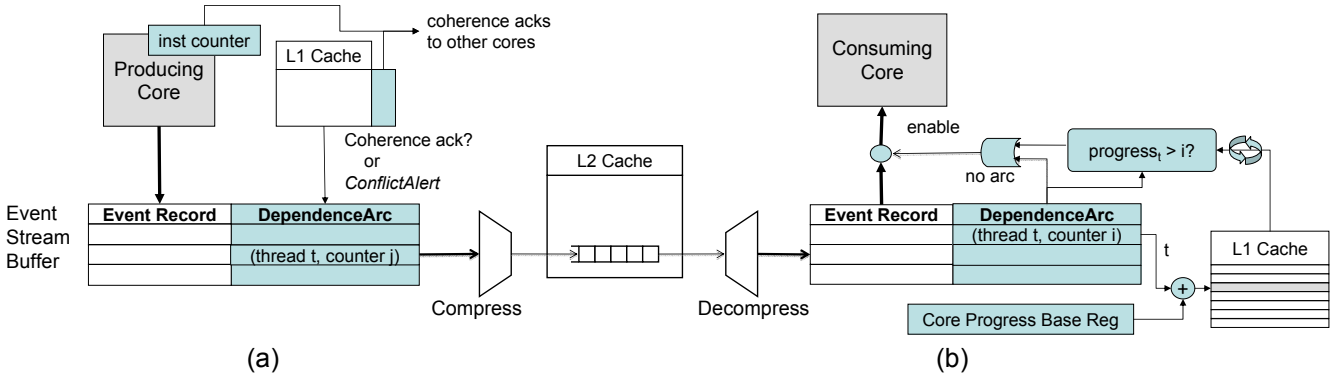


Figure 4. (a) Components for producing dependence arcs. As coherence messages arrive, they may generate dependence arcs, which are associated with pending events. As instructions retire, their associated records are committed to the stream. (b) Components for coordinating consumption of dependence arcs. As events are decompressed from the stream, each is checked if it has an associated dependence arc. If not, it may be dispatched for consumption; otherwise, a check determines if the dependence has been resolved.

the dependence arcs generated by coherence messages are stored into the event stream of the thread running on the core that caused the coherence event. Logically then, at least, the event stream contains both per-thread application operations (e.g. instructions) and per-thread dependence events.

We also consider an alternative design that reduces hardware complexity by not tagging cache blocks; the current per-core counter is (conservatively) sent in coherence messages instead of the more accurate per-cache block counter value of the preceding scheme. Because the per-core counter is often larger than the per-block counter, this may introduce artificial delays in lifeguard processing. However, our results in Section 7 show that the loss in lifeguard performance may be minimal.

5.2 Enforcing Event Ordering During Lifeguard Consumption

To see how dependence events are consumed online, assume that the event stream includes a dependence event indicating that application thread t's instruction i happened before application thread t''s instruction i'. In this case, the lifeguard monitoring thread t' (i.e., *Lifeguard*$_{t'}$) will wait until the lifeguard monitoring thread t (i.e., *Lifeguard*$_t$) has completed its work associated with i before *Lifeguard*$_{t'}$ begins processing i'.

To support the checking of *Lifeguard*$_t$'s progress, we propose the hardware coordination mechanism shown in Figure 4(b) and inspired by CNI [24]. The threads share a memory-mapped table of progress counters, indexed by thread id; *progress*$_t$ contains the counter value corresponding to the last advertised progress made by thread t. Each core maintains a hardware pointer to this table (the *Core Progress Base Register*). When the consumer hardware processes a dependence arc, it extracts the thread id, t, and counter value, i. Using t as an offset into the table, the hardware can determine the current counter value for thread t, *progress*$_t$, and determine whether it is greater than i. If so, the local event is delivered. If not, the consumer spins, re-reading $progress_t$ periodically, until the desired progress value is reached. A "dependence stall" event is delivered to *Lifeguard*$_{t'}$ in the meantime. Note that each core's counter will be maintained on a separate cache line to avoid excessive coherence traffic.

5.3 Enforcing Metadata Access Atomicity

While the mechanism described above ensures that lifeguards will process individual instructions in an order consistent with the monitored application, there is one other synchronization requirement for proper lifeguard execution: we must ensure that concurrently-executing lifeguards do not inadvertently corrupt their metadata due to a lack of mutual exclusion. If we were running the lifeguards on a machine that supported transactional memory [17], this requirement would be trivially satisfied if each lifeguard event handler was executed as a separate transaction. Because hardware transactional memory support is not widely available today (and was not assumed in our experiments), we also need a solution that works on conventional machines.

The good news is that in the simple case where application reads and writes translate only into corresponding reads and writes of the associated metadata by the lifeguard, the event ordering mechanism described in Section 5.2 is sufficient to satisfy all lifeguard synchronization concerns, since the read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependences for the metadata accesses will already be handled properly by the dependence orders captured from the application. In practice, however, lifeguards may perform not only a read but also a *write* or a *read-modify-write* of metadata in response to an application read. Because cache coherence does not dictate an ordering between application reads on separate processors, their corresponding lifeguard handlers may run concurrently, possibly resulting in simultaneous read-modify-write attempts to the same metadata locations. To prevent these unsafe scenarios from corrupting metadata, we need additional synchronization in certain cases, as discussed below.

Note that another characteristic of metadata can potentially cause a surprising data corruption problem: when the metadata is much smaller than its corresponding application data (e.g., a single bit of metadata tracks a byte of application data), writing to a metadata location may require a read-modify-write to update individual bits within a byte. Hence the seemingly benign case of performing concurrent metadata writes in response to concurrent application writes to separate memory locations (A and B) may cause a data hazard if *metadata*(A) and *metadata*(B) fall within the same byte (or minimal granularity of data access). We refer to this problem as a *bit-manipulation data race*. Fortunately, we are able to avoid this problem for the lifeguards and architecture we study by mapping the metadata such that whenever *metadata*(A) and *metadata*(B) fall within the same *byte*, A and B fall within the same *cache line* (and hence any write conflicts across application threads are already captured by the mechanism described in Section 5.2).

In summary, the order-enforcing mechanism described in Section 5.2 automatically guarantees metadata access atomicity with-

out any lifeguard modifications provided that the following three conditions are met:

1. there is a one-to-one mapping from application data accesses to lifeguard metadata accesses (e.g., a handler for a write to A accesses only $metadata(A)$);
2. application reads translate only into metadata reads; and
3. metadata bit-manipulation races are prevented as described above.

Note that conditions 1 and 3 are true for typical lifeguards. Condition 3 is typically satisfied because even if just a single metadata bit is used to track a 4-byte application word, with cache line sizes of 32 bytes or larger, accesses to different cache lines typically result in accesses to different metadata bytes.

Condition 2 is true for a large number of lifeguards, including TAINTCHECK and ADDRCHECK. However, some lifeguards (e.g., the data race detector LOCKSET [37]) violate condition 2: i.e., the lifeguards may perform metadata writes in response to application reads. To avoid potential data races to metadata, the lifeguard developers must use additional software synchronization (e.g., locks) to protect the metadata writes in read handlers. For example, one approach is to divide read handlers into a fast, frequent code segment and a slow, infrequent code segment, where the fast segment performs read-only metadata accesses, while the slow segment may perform read-write metadata accesses. If the slow segment acquired a lock and only performed a single write to metadata, then two read handlers (denoted H_a and H_b) would be properly synchronized, as follows. If both H_a and H_b perform writes in the slow segment, then the lock serializes H_a and H_b. If H_a performs reads in the fast segment but H_b performs a single metadata write to the same location in the slow segment, then H_a may see the value either before or after it is written by H_b, but both values are acceptable because the corresponding application reads are not ordered. Thus, we can use this *synchronization-free fast paths* approach to obtain good performance without sacrificing atomicity.

In practice, we observed that a wide range of lifeguards [29, 30, 32]—including TAINTCHECK—can be easily ported to satisfy these conditions, thereby preserving metadata access atomicity while also achieving reasonably high performance.

5.4 Implementing *ConflictAlert* to Effectively Handle System Calls and Logical Races

We extend the baseline mechanisms described in Section 5.1 to handle cases where dependence arcs are missing: (i) system calls because we do not capture OS kernel activity; and (ii) logical races because such races are not reflected in cache coherence traffic (as described in Section 4.3). Because these high-level application events are infrequent, we propose to implement a message broadcasting mechanism, called *ConflictAlert*, to handle these cases.

The order capturing components provide support for sending and receiving *ConflictAlert* messages. The send action is controlled by software. Note that previous work (e.g., LBA) already provides a way to instrument system calls and important library calls (via a wrapper library) for the purpose of obtaining event records for these high level events. Here, we extend this mechanism to issue requests for sending *ConflictAlert* messages. A *ConflictAlert* message contains information about the related high-level event, including event type and optionally event parameters (e.g., the affected memory range as discussed below). Upon receiving a *ConflictAlert* message, an order capturing component creates a *ConflictAlert* record that contains all the information in the message. A broadcasting message can be sent before (after) a system/library call; we call such a message a *CA-Begin* (*CA-End*) message. Note that these messages act as a serializing event for the issuing thread; it will not proceed past the message send until it receives an acknowledgement from the order-capturing components associated with all the other executing threads.

Handling Logical Races. For high-level events, such as `malloc` and `free`, lifeguards often only care about the begin or the end of the events but not both. For example, lifeguards mainly care about the end of `malloc` and the begin of `free`. Our solution enables lifeguards to specify which types of high-level events they care about and whether they care about the beginning or ending of such events. The lifeguards can also specify whether a *CA-Begin* or *CA-End* record with a particular event type should invalidate or flush IT, IF, and/or M-TLB. As described in Section 4.3, this will ensure the accelerators correctly handle high-level remote conflicts.

Handling System Calls. For a system call, we typically issue both *CA-Begin* and *CA-End* messages. Because these messages appear in the event streams of all executing threads, lifeguards are able to determine precisely whether a memory access in thread t occurs before, during, or after a system call issued by thread t'. Such determinations are important in many lifeguards; for example, ordering memory accesses with respect to `read` system calls is crucial in TAINTCHECK. Suppose that thread t issues a memory load from an address that corresponds to a `read` buffer. If the access occurs after the `read`, the destination will be tainted; if it occurs before, it may not be. If it is concurrent to the system call, the lifeguard will probably conservatively consider the destination to be tainted and possibly warn of the race condition. To enable the detection of such races, the wrapper library includes memory range information in the *ConflictAlert* messages for appropriate system calls, as discussed next.

Memory Range Parameters. The optional memory ranges associated with *ConflictAlert* messages typically may be calculated from the calling parameters of system/library calls. Given such information, for logical races, accelerators' states can be selectively flushed or invalidated. For system calls, we can use these ranges to detect racing conditions between application threads and the OS kernel.

Race detection can be performed in hardware or software. For hardware-based detection, we add a hardware range table per thread, which is leveraged at the lifeguard side. Upon seeing a *CA-Begin* (*CA-End*) record, the recorded memory range is inserted into (removed from) the table. The table has one entry per core in the system, and always maintains the information for the actively running application threads. Information for threads that are scheduled off can be flushed to memory. Given such a table, the order enforcing component can detect race conditions by checking event records against the table.

5.5 Supporting Total Store Ordering (TSO)

In a non-SC memory model, using cache coherence activities to infer the ordering of concurrent events may result in cycles of dependences, thus leading to deadlocks in deterministic replay and when delivering lifeguard events. In this subsection, we describe how we support TSO, overcoming these challenges.

Previous work on deterministic replay provided an extension to handle TSO that records data values when a processor observes a SC-violating behavior [46]. For every pair of $R \rightarrow W$ conflicting instructions where the load violates SC, the corresponding coherence message is not recorded. Instead, the value of the load instruction is saved in the log. This ensures that replay remains deterministic, as the load instruction acquires the correct value from the log. Unfortunately, recording data values is insufficient for parallel monitoring. In TAINTCHECK, for example, the data value alone fails to record the propagation behavior, and hence is insufficient for determining taint status. However, *as long as we ensure that the correct metadata is always available to non-SC reads, our lifeguards remain accurate*.

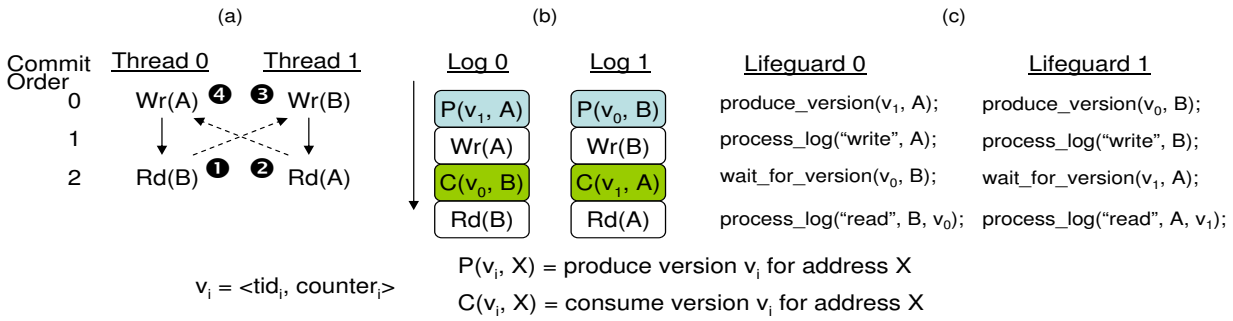


Figure 5. Supporting TSO. (a) An example of accesses that generate a cycle of dependences when coherence activity is used to infer ordering. (b) The contents of the event stream logs with entries for application instructions and annotations. (c) Actions taken by the lifeguards when processing their respective logs.

Table 1. Experimental Setup

Simulator description	
Simulator	Virtutech Simics 3.0.22
Extensions	Log capture and dispatch
Target OS	Fedora Core 5 for x86
Cache simulation	g-cache module

Simulation parameters	
Cores	2, 4, 8, 16 cores, 1 GHz clock cycle, in-order scalar, 65nm
Private L1-I	64KB, 64B line, 4-way assoc, 1-cycle access lat., LRU
Private L1-D	64KB, 64B line, 4-way assoc, 2-cycle access lat., LRU
Shared L2	2MB, 4MB, 8MB, 64B line, 8-way, 6-cycle access lat., 4 banks
Main Memory	90-cycle latency
Log buffer	64KB, assuming 1B per compressed record [4]

Benchmarks	Input
barnes	16K bodies
ocean	Grid size: 258 x 258
lu	Matrix size: 1024 x 1024
fmm	32768 particles
radiosity	Base problem: -room
blackscholes	simlarge
fluidanimate	simlarge
swaptions	simlarge
Compiled for x86, pthread synchronization primitives	

Our TSO solution enables us to reverse non-SC $R \rightarrow W$ arcs to become SC $W \rightarrow R$ arcs, enabling forward progress. We identify all the pairs of $R \rightarrow W$ conflicting instructions where the load violates SC, as proposed in [46]. The idea is that for each such arc from thread t to thread t', we require the lifeguard thread t' (for the writer) to create a copy of the previous metadata to be accessed by lifeguard thread t. When lifeguard thread t sees such an arc, it requests a copy of the correct metadata before analyzing the read event. In other words, we require lifeguard threads to create temporary versioned metadata for dealing with non-SC behaviors.

To implement this, at the application side, we do not record the $R \rightarrow W$ dependence. We create a version for the access by combining thread IDs with the current event record ID at thread t. Thread t records a "consume version" record in its event stream. The version number is piggy-backed with cache coherence messages (e.g., with the ack to a cache invalidation request). Upon seeing the version number, thread t' records a "produce version" record in its event stream.

Figure 5 shows the scheme as it affects (a) the application, (b) the event stream (log) and (c) the lifeguard, all under TSO. In Figure 5(a) we observe a memory ordering of ❶❷❸❹, a non-SC cycle. Specifically, thread 0 reads address B first. Later, thread 0 receives an invalidation for address B. Hardware at the core where thread 0 runs identifies the potential dependence cycle, and includes in the coherence reply message a request for generating a version (<0,2>) of metadata for address B. Additionally, the event record for Rd(B) is annotated with the version (<0,2>) of metadata the lifeguard should read before processing. The order capturing hardware at the core where thread 1 runs receives the version number in the coherence message and discards the dependence. Instead, it inserts an annotation before the event record for the Wr(B) instruction so the lifeguard generates the versioned metadata for address B before overwriting metadata. The event streams for threads 0 and 1 appear in Figure 5(b). Figure 5(c) shows the lifeguards processing their event streams. Lifeguard 0 generates versioned metadata for address A, processes Wr(A), waits for the versioned metadata for address B to be generated, and then processes Rd(B). The $R \rightarrow W$ dependence has logically been inverted, but the two lifeguard threads remain correct.

Hardware Accelerators Revisited. Some of the accelerators described in Section 4 require slight modifications to work in a TSO environment. Specifically, both IT and IF are unable to differentiate among different versions of the same memory location. We solve this problem by always delivering both the event that accesses versioned metadata, as well as any pending state that inherits/caches status of the same address. Given that the number of pairs of accesses from different threads that violate SC semantics is low [15], we expect this solution to be the most simple and efficient.

6. Experimental Setup

Simulation Setup. We use the Simics [43] full-system simulator to model different configurations of sequentially consistent shared-memory CMP systems. We extend Simics with log record capture and event dispatch support. We assume a dependence tracking mechanism based on RTR [46], modified as described in Section 5.1. Table 1 shows the simulation parameters, where all are modeled under the same technology (65nm). Every benchmark is run with 2, 4 and 8 application threads on a 4-core, 8-core, 16-core machine, respectively, devoting half of the cores to the application and half to the lifeguards. We simulate a two-level cache hierarchy with private, split L1 caches and a shared, unified, and inclusive L2. L1 parameters are maintained constant across configurations, while for L2 we alter only the size as the number of cores increases in the system. Associativity, access latencies, line size and num-

ber of banks utilized are provided by CACTI [19], as an optimum configuration for the specific cache size.

Benchmarks. We choose a diverse set of CPU-intensive parallel benchmarks to "stress test" instruction-grain monitoring. We include benchmarks from the SPLASH-2 [44] benchmark suite, as well as from the recently released PARSEC [1] suite. Specifically for PARSEC, we chose the benchmarks for which (i) the exact number of generated threads can be controlled by an input parameter and (ii) the runtime is not prohibitively long. Although our proposed technique is not limited by the number of threads running on the system, it becomes complicated to identify bottlenecks on the system, and report performance, when there are more application and lifeguard threads than available cores. Table 1 shows the benchmarks used to evaluate our system, along with their corresponding input parameters.

Performance Measurements. For every CMP configuration, we run every benchmark alone in the system, with monitoring turned off. We perform functional simulation of the whole system and its cache hierarchy, taking periodic checkpoints of the system state. In parallel with the functional simulation, a log of the application is produced for every interval between two checkpoints, and saved in the local disk. A native version of the lifeguard we want to simulate consumes each log in order to produce the metadata state at the beginning of every interval. To take performance measurements, we focus on the checkpoints that include the parallel phase of the application, and we report results only for this phase. We first load a checkpoint and instantiate the lifeguard we want to simulate. We initialize the metadata state and start functional simulation of the system in order to warm up the lifeguards' L1 data caches. L1 data caches for the cores that run application threads are already warm due to the functional simulation of the first step. After the warming up window, we turn on detailed performance simulation and run for a given timing window.

Although functional warming of caches is not always safe, we are confident that lifeguards' L1 data cache states are warmed up during performance measurements for two reasons: (i) the ratio of metadata size per application byte is small (in our case 1 or 2 bits per byte), resulting in much smaller working sets; and (ii) the measured lifeguard miss rates are consistently 10 to 100 times lower than application miss rates. Even if our approach were not accurate enough, this would only result in having a slow lifeguard, which would make our performance results pessimistic.

Lifeguard Implementation. To evaluate the performance of our approach, we implemented a parallel version of TAINTCHECK [32] and a parallel version of ADDRCHECK [28]. Both of the lifeguards require instruction-grain monitoring, while utilizing the full set of accelerators studied. For performance reasons, we modified TAINTCHECK to associate 2 metadata bits per application byte, so that frequent cases (application reads/writes 1 word/double) can be handled efficiently (lifeguard reads/writes 1 byte/word), while ADDRCHECK associates 1 metadata bit per application byte. The metadata for both lifeguards are organized in a two-level data structure. The first level is a pointer array, pointing to chunks of metadata in the second level. The higher part of the application effective address is used to index the first level table, while the second part indexes the metadata chunk. This organization saves space, because a chunk is allocated only when the corresponding virtual space is used by the application. TAINTCHECK requires the ordering (outcome) of all the application data races, as well as correct ordering for all the high level events. Because TAINTCHECK maps application reads to metadata reads and applications writes to metadata writes, no special synchronization mechanism is required by lifeguard threads apart from the ordering provided by the dependence arcs that are already included in the log. ADDRCHECK requires only the correct ordering of all the high-level events of the allocation library with respect to the rest of the application activity. Because ADDRCHECK maps application reads and writes to metadata reads, no other synchronization mechanism is required, apart from the ordering provided by the *ConflictAlert* messages already included in the log.

7. Experimental Evaluation

We begin our evaluation of online parallel monitoring by examining its impact on overall execution time. Figure 6 shows three different cases: (i) the application without any monitoring (NO MONITORING), (ii) the state-of-the-art approach of timeslicing the application threads onto one processor and performing the lifeguard analysis on another processor (TIMESLICED MONITORING), and (iii) our technique (PARALLEL MONITORING). The x-axis indicates the number of application threads for each benchmark, ranging from one (i.e. sequential execution) to eight. Note that for k application threads, the NO MONITORING, TIMESLICED MONITORING, and PARALLEL MONITORING cases run on $2k$, 2, and $2k$ cores, respectively. The y-axis shows execution time normalized to the application running sequentially without monitoring. (Recall that the application stalls whenever the circular log buffer is full, and hence the execution times for the application and lifeguard are the same.) Both the TIMESLICED and PARALLEL monitoring schemes use lifeguard accelerators (the former requires only the sequential versions [5, 6]). ADDRCHECK utilizes the Metadata TLB and the Idempotent Filters accelerators, while TAINTCHECK the Metadata TLB and the Inheritance Tracking ones.

We organize our evaluation following this approach (k application threads on $2k$ cores), as a way to isolate the slowdown induced by monitoring, while maintaining the number of application threads constant (*Constant-Application-Size*). However, this assumes there are always cores available for lifeguard threads to run on. A *Constant-Resource* comparison, where k lifeguard and application threads *in total* run on k cores, would be appropriate to show the opportunity cost of using cores for monitoring, although it would make it hard to identify sources of overhead. We present and discuss our results in the rest of the section under the *Constant-Application-Size* approach. Data for the *Constant-Resource* approach can be found in Figure 6, by comparing the $2k$-threaded NO MONITORING case against the k-threaded PARALLEL MONITORING case (although this comparison is not discussed here).

As we see in Figure 6, our PARALLEL MONITORING approach is dramatically faster than today's TIMESLICED approach, enabling TAINTCHECK to achieve speedups ranging from 1.5X–4.1X with two application threads (roughly twofold on average), and speedups ranging from 5.3X–85X with eight application threads (roughly 24X on average). ADDRCHECK also achieves similar speedups ranging from 1.4X–3.1X for two threads (2.2X on average), and 5.7X–126X for eight threads (36X on average). This result is not surprising: our PARALLEL approach achieves these speedups relative to TIMESLICED because it can take advantage of the parallel hardware threads on the CMP (as well as additional cache space, which causes some superlinear effects) to accelerate both the application and the lifeguard. Clearly this is a large improvement over the state-of-the art.

Understanding Overheads. Another interesting question is how PARALLEL MONITORING compares with NO MONITORING. As we see in Figure 6 and in Figure 7, the relative slowdown when we add PARALLEL MONITORING (given a fixed number of application threads) is less than 1.5X in the majority of cases for TAINTCHECK. (In LU and OCEAN, it is less than 1.15X.) These numbers are consistent with the relative slowdowns observed in our earlier study [5, 6] for TAINTCHECK running on sequential applications.

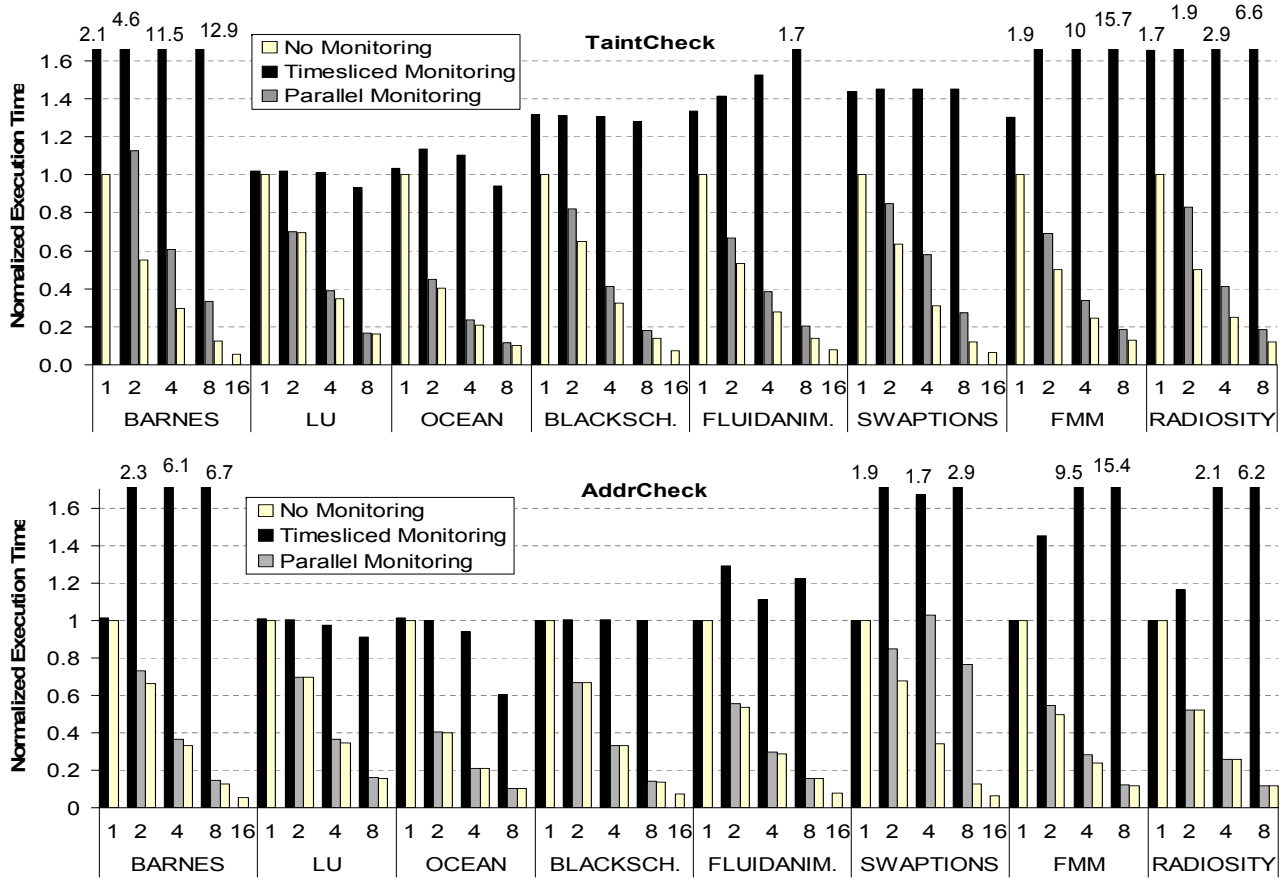


Figure 6. Execution time for PARALLEL and TIMESLICED monitoring schemes for 1, 2, 4 and 8 application threads. The execution time for the application alone is also shown (for 2, 4, 8 and 16 threads).

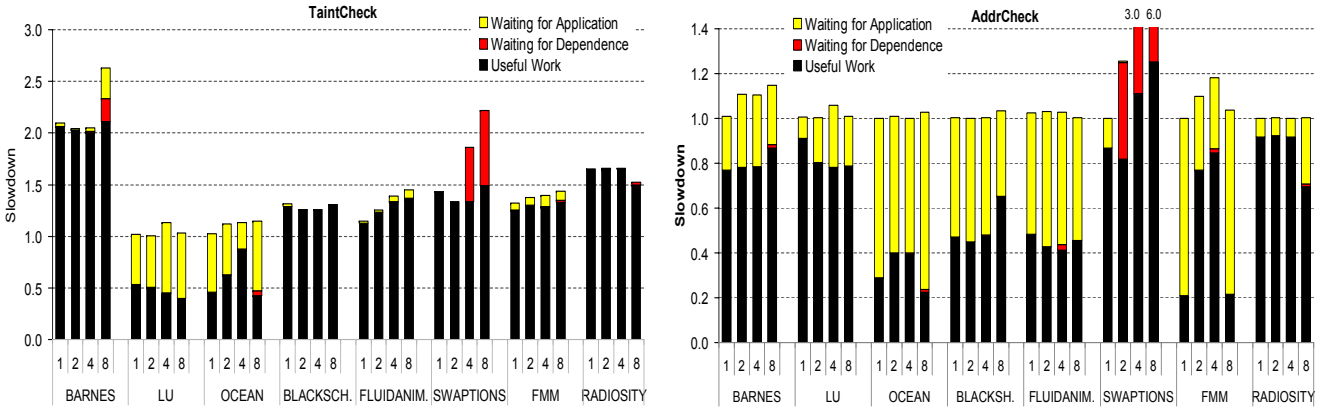


Figure 7. Performance slowdown for 1, 2, 4 and 8 application threads when PARALLEL monitoring is enabled on the system. Slowdown for every configuration is normalized with respect to the application with the same number of threads running alone on the system. The case with 1 application thread is identical with the one from figure 6.

Furthermore, ADDRCHECK does not incur any practical overhead in the majority of the cases ($< 10\%$), and almost in all the cases the overhead is less than 20%, with the exception of SWAPTIONS that we will discuss shortly.

To provide more insight on these slowdowns, Figure 7 breaks down the time into three components: time spent doing *useful work*, time spent *stalled waiting for data dependences* from other lifeguard threads, and time spent *stalled waiting for the application* to produce events in the log. The latter category indicates that there are bursts of time when the lifeguard catches up with the application. The WAITING FOR DEPENDENCE category indicates that event ordering across the lifeguards is having an impact on their performance; we observe this with 4 and 8 threads in SWAPTIONS, and (to a lesser extent) with 8 threads in BARNES.

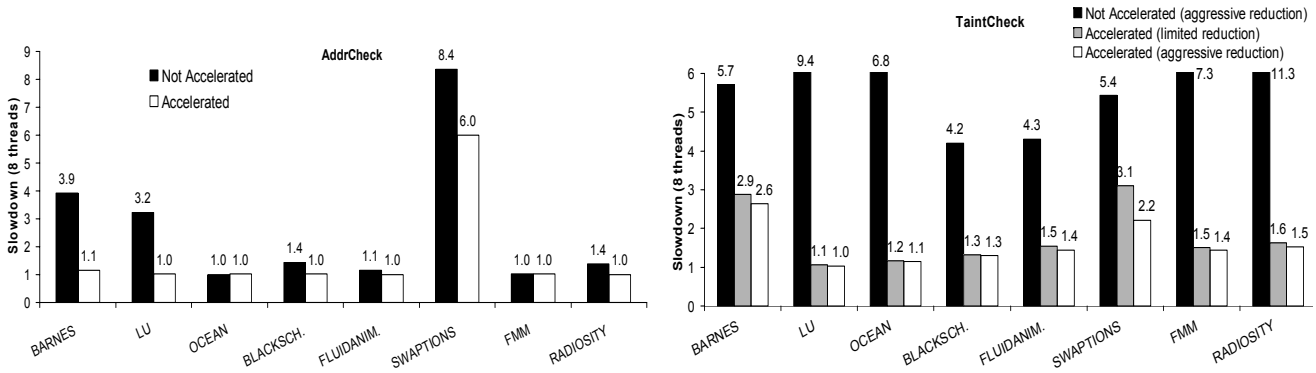


Figure 8. Variations of the PARALLEL scheme design with 8 application threads, normalized to NO MONITORING with 8 threads.

As we see in Figure 7, TAINTCHECK slows down BARNES by a factor of 2X simply because the lifeguards are busy performing useful analysis work. Lifeguards perform different amounts of work for different types of instructions, and the mixture of instructions in BARNES (which does significant pointer chasing) happens to invoke more expensive lifeguard processing than in LU and OCEAN (which are matrix-oriented). SWAPTIONS, on the other hand, has a much higher frequency of remote conflicts than the other applications, and therefore the lifeguard threads spend more of their time stalled waiting for other lifeguard threads to catch up. We observe that the median stall time for one of these lifeguard synchronization events is over 500,000 cycles, which is surprisingly large. Interestingly, this large latency is *not* caused by a chain of stalls (e.g., THREAD A waits for THREAD B, which is in turn waiting for THREAD C, etc.). Instead, the problem is simply load imbalance within the lifeguards due to the combination of point-to-point synchronization and static assignment of work across lifeguard threads. To reduce this source of overhead, we would need a mechanism for *work stealing* across lifeguard threads, which may be possible using techniques proposed by Ruwase *et al.* [36] and Nightingale *et al.* [33].

On the other hand, ADDRCHECK tends to be less costly than TAINTCHECK, mostly because it needs to process a narrower set of instructions (only memory accesses to the heap). This has the immediate result of the lifeguard spending a big part of its execution waiting for the application to produce events in the log. Out of all the benchmarks, SWAPTIONS is again the one penalized the most due to ordering from conflicting accesses in the application and *ConflictAlert* messages from the consecutive allocations and deallocations of memory from the application threads. Specifically, we measured that throughout the execution of its parallel section, SWAPTIONS performs approximately 450K pairs of allocations and frees, for which a pair of *ConflictAlert* messages is generated for every call. However, every pair of *ConflictAlert* messages is translated to a barrier at the lifeguard side and corresponds to a conservative solution for all the lifeguard threads to stall until the metadata state is updated after every `malloc/free`. We also observed that 1/3 of all allocations request at most 64 bytes of memory (1 cache block), 2/3 of all allocations request at most 32 cache blocks and *no* allocations request more than 128 cache blocks. An alternative to the *ConflictAlert* mechanism for small allocations would be to induce dependence arcs by touching the allocated/freed cache blocks within the allocation library. This could enable lifeguard threads not interested in these blocks to proceed without stalling.

Impact of Lifeguard Accelerators. To quantify the performance benefit of our lifeguard accelerators (described earlier in Section 4), Figure 8 shows the slowdown—relative to the application running with 8 threads and no monitoring—of the PARALLEL MONITORING case without and with acceleration, for our two lifeguards. Specifically for ADDRCHECK we observe that accelerators have a large impact on the "heavy" benchmarks (e.g., BARNES) and achieve a speedup of 3.4X–1.13X (no practical speedup for LU and FMM that have less than 1% overhead). For TAINTCHECK, compare the height of the leftmost and rightmost bars for each application in Figure 8. As we see in the figure, lifeguard acceleration has a large impact on performance for all the benchmarks, resulting in speedups ranging from 2X (BARNES) to 10X (LU).

Impact of Aggressive Dependence Reduction. In our experiments so far, we have assumed an aggressive design for eliminating unnecessary dependence arcs, similar to FDR [45]. One interesting question is how much performance would we sacrifice with a less aggressive scheme that requires less hardware. In particular, we consider a design that maintains only one full (64-bit) timestamp per processor (as opposed to one timestamp per cache block, as in FDR [45]). The performance of this less-aggressive case is shown for TAINTCHECK only as the center bar for each application in Figure 8. Comparing these center bars with the rightmost bars, we observe that for most applications, the performance loss of the less-aggressive scheme is relatively small. One reason why this is true is that most of the time when a lifeguard encounters an incoming dependence arc, the dependence has already been satisfied. Hence in many cases, the dependences that are being eliminated under aggressive dependence reduction were not causing expensive stalls in the first place. For a group of dependences between a pair of threads that occur close together in time, it is likely that only one expensive stall will occur for the entire group; hence reducing some of these dependences is of little consequence. The performance impact of the less-aggressive design is more noticeable in BARNES and SWAPTIONS, which are the two applications that do spend a noticeable amount of time stalling on inter-thread dependences. In summary, although aggressive dependence reduction shows benefits in some cases, a less aggressive design also appears to be a viable design option.

8. Conclusions

This paper presents the first design of a general-purpose online monitoring platform that supports fast parallel monitoring of multithreaded parallel applications. To obtain fast monitoring, we successfully parallelize the three hardware accelerators in [6], overcoming a variety of challenges. Our simulation results of a 16-core CMP demonstrate that: (i) our parallel accelerators improve performance by 2–9X and 1.13–3.4X for the TAINTCHECK and ADDRCHECK lifeguards, respectively; (ii) we are 5–126X faster than the time-slicing approach required by existing techniques; and (iii)

our average overheads for applications with eight threads are 51% and 28% for the two lifeguards, respectively.

9. Acknowledgments

We would like to thank Olatunji Ruwase and Michael Ryan for their comments and valuable feedback on this paper. This work is supported in part by a grant from the NSF.

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[2] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004.

[3] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software – Practice and Experience*, 30(7), 2000.

[4] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *ASID Workshop at ASPLOS*, 2006.

[5] S. Chen, M. Kozuch, P. B. Gibbons, M. Ryan, T. Strigkos, T. C. Mowry, O. Ruwase, E. Vlachos, B. Falsafi, and V. Ramachandran. Flexible hardware acceleration for instruction-grain lifeguards. *IEEE Micro*, 29(1):62–72, 2009. *Top Picks from the 2008 Computer Architecture Conferences*.

[6] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *ISCA*, 2008.

[7] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-safe dynamic binary translation using transactional memory. In *HPCA*, 2008.

[8] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A programmable macro engine for customizing applications. In *ISCA*, 2003.

[9] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO*, 2004.

[10] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *ISCA*, 2007.

[11] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Engineering*, 27(2), 2001.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, 2002.

[14] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX ATEC*, 2006.

[15] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *ISCA*, 1999.

[16] M. L. Goodstein, E. Vlachos, S. Chen, P. B. Gibbons, M. Kozuch, and T. C. Mowry. Butterfly analysis: Adapting dataflow analysis to dynamic parallel monitoring. In *ASPLOS*, 2010.

[17] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *HPCA*, 1993.

[18] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ISCA*, 2008.

[19] HP Labs. Cacti 5.1 Technical Report. http://www.hpl.hp.com/research/cacti/.

[20] H. Kannan. Ordering decoupled metadata accesses in multiprocessors. In *MICRO*, 2009.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[22] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *ISCA*, 2008.

[23] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *ASPLOS*, 2009.

[24] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent network interfaces for fine-grain communication. In *ISCA*, 1996.

[25] V. Nagarajan and R. Gupta. Architectural support for shadow memory in multiprocessors. In *VEE*, 2009.

[26] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS*, 2006.

[27] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, 2005.

[28] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, U. Cambridge, 2004. http://valgrind.org.

[29] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[30] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE*, 2007.

[31] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[32] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[33] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *ASPLOS*, 2008.

[34] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A.-R. Adl-Tabatabai. Architecting a chunk-based memory race recorder in modern CMPs. In *MICRO*, 2009.

[35] F. Qin, C.Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, 2006.

[36] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing Dynamic Information Flow Tracking. In *SPAA*, 2008.

[37] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic race detector for multi-threaded programs. *ACM TOCS*, 15(4), 1997.

[38] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. Heapmon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM J. on Research and Development*, 50(2/3), 2006.

[39] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.

[40] G.-R. Uh, R. Cohn, B. Yadavalli, R. Peri, and R. Ayyagari. Analyzing dynamic binary instrumentation overhead. In *WBIA Workshop at ASPLOS*, 2006.

[41] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *HPCA*, 2008.

[42] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. MemTracker: Efficient and programmable support for memory access monitoring and debugging. In *HPCA*, 2007.

[43] Virtutech Simics. http://www.virtutech.com/.

[44] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

[45] M. Xu, R. Bodik, and M. D. Hill. A 'Flight Data Recorder' for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[46] M. Xu, R. Bodik, and M. D. Hill. A regulated transitive reduction (RTR) for longer memory race recording. In *ASPLOS*, 2006.

[47] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *HPCA*, 2007.

[48] Y. Zhou, P. Zhou, F. Qin, W. Liu, and J. Torrellas. Efficient and flexible architectural support for dynamic monitoring. *ACM TACO*, 2(1), 2005.