Parallel LBA: Coherence-based Parallel Monitoring of Multithreaded Applications

Evangelos Vlachos[1] **Michelle Goodstein**[1]
Michael Kozuch[2] **Shimin Chen**[2] **Babak Falsafi**[3]
Phillip B. Gibbons[2] **Todd C. Mowry**[1,2] **Olatunji Ruwase**[1]

March 4, 2009
CMU-CS-09-108

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1]*Carnegie Mellon University* [2]*Intel Research Pittsburgh*
[3]*École Polytechnique Fédérale de Lausanne*

This research is supported by grants from the National Science Foundation and by Intel Research Pittsburgh. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Intel, the NSF or the US government.

Keywords: Dynamic Program Monitoring, Deterministic Replay, Dataflow Analysis, Log-Based Architectures

Abstract

As multicore processors become more prevalent, application performance increasingly depends upon parallel execution through multithreading. However, existing software correctness-checking techniques have focused on single-threaded applications and need to be extended to support the efficient monitoring of multithreaded applications. In particular, software lifeguards (online tools for monitoring software correctness) must be able to reason about the relative ordering of memory accesses and other events issued from different threads. Acquiring such event orderings is not trivial, and prior work has resorted to serializing application threads on a single core to manage concurrency. This paper solves the multithreaded application monitoring problem using a combination of techniques. First, it leverages cache coherence to record the interleaving of memory accesses from different threads. Second, it extends the framework to comprehend "logical races"—events involving at least two threads where the relative order may be important for particular lifeguards, but where not all events are memory accesses (e.g., a race between a malloc() and a memory write). Finally, it uses special thread-switch messages to ensure proper synchronization with cores when application threads are swapped out. Our approach enables software lifeguards not only (i) to reason efficiently about the state of the application, but also (ii) to avoid costly synchronization among lifeguard threads that are maintaining shared shadow state. Existing single-threaded lifeguards may be ported to this framework easily because the semantics of event delivery are preserved. Simulation results for a parallel dynamic information flow lifeguard on a 16 core CMP indicate that our monitoring is at least 5X faster than existing techniques.

1 Introduction

The path to high performance in the immediate and forseeable future is parallel processing. Because parallel programming is a notoriously difficult task, programmers will need all of the help that they can get to debug their software. Hence there has been a recent body of work on how to capture a window of recent inter-thread data dependences prior to a program crash for the sake of *post-mortem debugging* [10, 11, 14, 16, 17, 33, 34].

While post-mortem debugging is useful, another strategy for understanding and coping with software bugs is to use *dynamic monitoring tools* (aka *lifeguards*) that check some aspect of program correctness as the program runs [2, 9, 13, 21, 26]. Compared with post-mortem debugging, lifeguards offer the advantages that they can track program behavior throughout the entire execution (rather than simply trying to work backwards through a finite window of time near the program crash), they can model rich semantic information about the program, and they can potentially catch problems (and hopefully contain their damage) prior to the point where they cause the software to crash.

Lifeguards have traditionally been implemented using dynamic binary instrumentation frameworks (e.g., Valgrind [21], Pin [13], DynamoRio [2]). A recent study demonstrated that with hardware-assisted logging and other mechanisms to eliminate redundancy [4], the run-time overhead of lifeguards can be reduced to something acceptably small for *single-threaded* applications. In this study, we focus on using lifeguards to monitor *multithreaded* applications.

To understand the overall behavior of a multithreaded application, both lifeguards and post-mortem analysis tools combine local information regarding the behavior of individual threads with global information about how those threads interact with each other. While it is relatively straightforward to capture local information within threads, the real challenge is in tracking inter-thread data dependences.

1.1 Tracking Inter-Thread Dependences for Dynamic vs. Post-Mortem Analysis

Because the performance and functionality requirements for *dynamic* versus *post-mortem* analysis tools are quite different, the solutions for how we track inter-thread dependences are also quite different. For post-mortem analysis, the goal is to capture a finite window of activity prior to the point where a program crashes such that an offline tool (e.g., an interactive debugger) can later replay the events within this window to hopefully shed light on the cause of the crash. To maximize the effective size of this replay window given a fixed storage size (thereby increasing the likelihood that it contains the root cause of the problem) and to minimize bandwidth requirements, the key metric for post-morten analysis is *compression*. Hence techniques such as *transitive reduction* [22] have been used to eliminate unnecessary dependence arcs from the log [33]. In contrast, the performance overhead of consuming the dependence log is not a concern, because it occurs offline; in fact, it can even occur sequentially.

For dynamic analysis tools (i.e., lifeguards), the key metric is *end-to-end performance*: not only must the capture of inter-thread dependences occur in real-time (as is the case for post-mortem tools), but the interpretation of these dependences and the execution of the lifeguards themselves must also occur in near real-time. Hence, the lifeguards must execute in parallel to keep up with multithreaded applications, and inter-thread dependence information must be presented to the lifeguards in a ready-to-consume fashion that does not require significant software overhead. The good news, however, is that because lifeguards are continuously consuming inter-thread dependence information as the application executes, we are not concerned about storing the dependence information efficiently (since it will be both produced and consumed on-chip). This has the benefit that we can avoid using transitive reduction techniques [22, 33, 34] that would tend to serialize (artificially) lifeguard execution.

Finally, another key difference is due to the functionality of the analysis. For post-mortem analysis, the goal is to deterministically replay the state of the application within the window of capture. In contrast, lifeguards run alongside the application, performing analogous but different operations on shadow state to model and check some aspect of program correctness (as discussed in more detail later in Section 2). Hence, the ordering requirements for lifeguards are often both more lax and more strict than post-mortem analysis: more lax because some aspects of the application state may be irrelevant to the lifeguard, and more strict because disjoint memory locations may be semantically linked in the lifeguard's model of correctness in a way that would not be captured by address-based dependence tracking (e.g., the state in the header of a heap-allocated block of memory and the data within that block), which could result in a *logical race* (discussed later in Section 2). Fortunately, our solution for logical races solves another challenge for

lifeguards, which is that we would like them to work even if they operate only at a user-level (rather than being forced to have system-level privileges in order to continue their monitoring through system calls).

1.2 Related Work

There has been a significant body of work on capturing inter-thread dependences for the sake of deterministic replay and post-mortem analysis [10, 11, 14, 16, 17, 33, 34]. While our approach builds upon the central insight in some of these designs that coherence traffic can help us track inter-thread dependences, our final design is optimized for the requirements of lifeguards, as described above. There has also been significant work on hardware support for dynamic monitoring tools [4, 6, 36]. Our work builds upon the framework described by Chen *et al.* [4], and to our knowledge is the first framework that correctly handles a broad set of lifeguards operating in parallel to monitor multithreaded applications. Finally, there has also been work on parallelizing a lifeguard that is monitoring a single-threaded application [24, 25]; these approaches are orthogonal and complementary to the focus of this paper (which is on monitoring multithreading applications).

1.3 Contributions

This paper makes the following research contributions:

- we characterize the challenges and requirements for supporting dynamic parallel monitoring, including the problem of *logical races*;
- we present a design for tracking inter-thread dependences that is suitable for driving dynamic parallel monitoring tools (we describe this design for a Sequentially Consistent machine, and also describe how it can be extended to a machine with Total Store Ordering);
- we implement a parallel version of a data-flow tracking lifeguard to evaluate our design, and demonstrate that (i) our scheme scales sufficiently as the number of cores increases, and (ii) it runs significantly faster than existing techniques, e.g., at least 5X faster than a standard thread-multiplexing technique on a 16 core simulated CMP.

2 Requirements for Dynamic Parallel Monitoring

Several interesting software lifeguards belong to the class known as *instruction-grain lifeguards*; these lifeguards perform detailed online monitoring of a target application by inspecting execution at the granularity of individual instructions. Examples include ADDRCHECK [18], which checks for memory allocation errors; MEMCHECK [19,20], a superset of ADDRCHECK that checks for memory initialization errors; LOCKSET [26], which checks for locking protocol violations; and TAINTCHECK [23], which checks for memory overwrite security attacks.

These lifeguards tend to have a similar structure in that they (1) associate fine-grained metadata with the monitored application's address space, (2) check desired invariants by reading the metadata, and (3) maintain the metadata through updates associated with particular events. For example, TAINTCHECK maintains a bit for every byte in the application's address and register space indicating whether the byte is tainted ("suspect") or not. The taint bits are set as suspect data enters the application (such as from a network read), propogated as data moves through the application's address space, and checked when the application performs system calls, indirect jumps, etc.

The challenge with instruction-grain lifeguards is that software-only implementations, effected through dynamic binary instrumentation (DBI), behave very poorly—application slowdown of one to two orders of magnitude is not uncommon [18, 21, 29]. While several studies have proposed hardware support for specific lifeguards [7, 8, 27, 28, 30, 35, 37], only two have suggested *general-purpose* lifeguard support (DISE [6] and LBA [3, 4]), neither of which addressed the challenge of monitoring multithreaded applications. (Previous work [5] addressed multithreaded DBI, but it (a) required transactional memory support and (b) did not address the performance issues that attend DBI.)

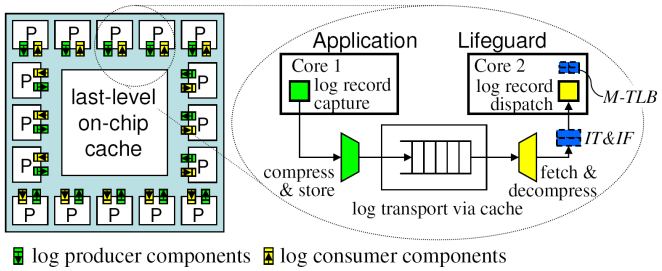

Figure 1: Log-Based Architecture (LBA) design showing symmetric cores with log producer and consumer components for each core. (Image from Chen, *et al.* [4] ©ACM.) The IT&IF component is not relevant to this work. The M-TLB component is used in our experiments (later in Section 5).

2.1 Background: Log-Based Architectures (LBA)

The baseline LBA [4] design provides an interesting platform for single-threaded, instruction-grain lifeguards by augmenting every processor core of a chip multiprocessor (CMP) with a log producer component and a log consumer component (Figure 1). As a monitored application executes, the producer component on the core supporting that application records an execution trace of the application into a log buffer, which resides in the CMP's cache hierarchy. The lifeguard, running on another core in the CMP, monitors the application by asynchronously consuming the execution records in the log.

To avoid the overhead of a software fetch-and-decode loop, the lifeguard is organized as a collection of event handlers—one per log event type. As log records are consumed, the log consumer component extracts the type of the next record and invokes the appropriate lifeguard handler. If the application produces records faster than the lifeguard can consume them, the log buffer fills, and the application core stalls. Similarly, if the lifeguard empties the buffer, its core stalls until additional records arrive.

2.2 Extending LBA for Parallel Application Monitoring

We extend the previous work on LBA to support multithreaded applications by enabling each core supporting an application thread to capture the execution trace for its thread into a separate buffer.[1] The lifeguard associates a thread with each buffer, and the buffers are consumed in parallel. (Recall that this parallel consumption is crucial to the lifeguard's efforts to keep up with the parallel application.) Any lifeguard thread that has a non-empty log buffer can retrieve the next log entry in its buffer and update metadata state accordingly.

A significant advantage of this approach is that the software architecture of a parallel lifeguard is very similar to that of a corresponding single-threaded lifeguard. The most significant difference is related to the metadata that instruction-grain lifeguards maintain; multithreaded lifeguard threads maintain metadata in shared memory.

While this straightforward extension of the original LBA work enables intra-thread operations to be recorded and processed, parallel log processing by multiple threads introduces two significant issues:

1. Log Event Ordering: To maintain the correct view of application state, the lifeguard must process application events in the same order as they were produced by the application (if those events express a dependence), despite the asynchronous log consumption.

2. Metadata Access Atomicity: Because metadata state is shared between threads, the lifeguard must take care that it is accessed atomically.

The ordering requirement is particularly important because insufficient regulation between lifeguard threads would mean that a thread could consume an event out-of-order with respect to another thread's consumption of a different

[1] The alternative, merging multiple logs into a single log, is prohibitively expensive due to the bandwidth associated with the application thread logs.

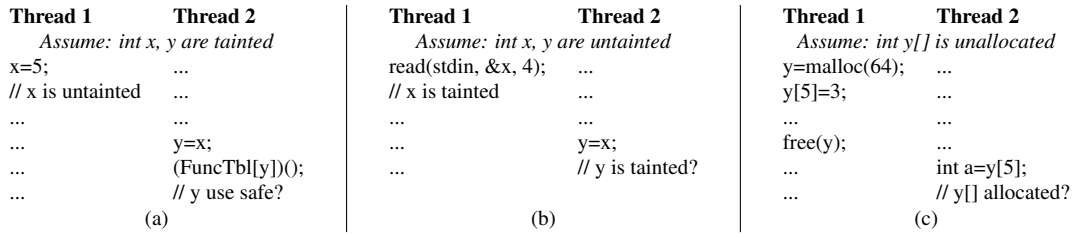

Figure 2: Examples of (a) a data race, (b) a data race involving a system call, and (c) a logical race between two threads. In each case, two unsynchronized threads execute in parallel, and a lifeguard will be unable to determine the state of the application after execution without resolving the thread interleaving.

event. Using TAINTCHECK as an example, if thread A in the application fills a buffer from the network, and thread B then performs a system call that involves the buffer, the system must ensure that the lifeguard thread for A's events reaches the buffer fill operation (recording a "tainted" status) before the lifeguard thread for B's events reaches the system call, regardless of the relative speed of the two lifeguard threads.

Event ordering associated with explicit synchronization can be handled easily by the LBA framework through its *annotation* mechanism [4], which enables system software to record marks for interesting calls like `lock()` and `unlock()` in the log. A lifeguard thread observing one of these marks in its log can then stall (if needed) until conditions needed to proceed are met. However, a complementary mechanism must be added to handle dependences not protected by synchronization (i.e., *data races*).

Section 3 describes our approach for recording inter-thread ordering information, which resolves data races by leveraging cache-coherence observations to record inter-thread dependence arcs in the logs. While this approach was inspired by several recent post-mortem debugging efforts [11, 14, 16, 17, 33, 34], our design is optimized for lifeguard requirements, not post-mortem deterministic replay. For example, because user-mode lifeguard code should not have visibility into privileged state, lifeguards are restricted to recording ordering information at the application level. Consequently, some dependence arcs will be lost when the application is not executing, such as during a system call. A mechanism is then required to compensate for the missing arcs (Section 3.2).

By re-using this compensation mechanism, the system can also enable lifeguards to resolve the outcome of *logical races*. We define a *logical race* to be two unsynchronized memory operations from different threads involving the same address, where at least one is a *logical access*; logical accesses are operations that involve, but do not access, memory addresses, such as allocation and de-allocation. To correctly maintain metadata, lifeguards need to resolve the order of logical races. Figure 2 depicts the three types of races.

Perhaps the most significant difference between post-mortem debugging tools and this work is our requirement for online consumption and enforcement of the ordering information. Here, when a lifeguard fetches a dependence record from the log, the lifeguard must stall until the dependence is resolved. As will be described in Section 3.1, the stalling may either be implemented in hardware or software.

In either case, because the dependence arcs reflect the cache-coherence state of the pertinent memory locations during application execution, the stalling mechanism will ensure that at most one lifeguard thread will consider itself to have exclusive access to the corresponding metadata. Consequently, the stalling mechanism provides our Metadata Access Atomicity requirement for a broad class of lifeguards.

3 Hardware Support for Dynamic Parallel Monitoring

In this section, we present our hardware solution for enabling dynamic parallel monitoring of multithreaded applications, described as extensions to the LBA platform (Section 2.1). In Section 3.1, we begin by describing our baseline mechanisms for capturing application event dependences at the log producers and enforcing these dependences at the log consumers, assuming no system calls and no logical races. Then, in Section 3.2, we study the dependences because of system calls and logical races in detail, and propose hardware mechanisms to address the challenges. In Section 3.3, we propose a lightweight lifeguard metadata synchronization scheme that leverages the dependence-enforcing hard-

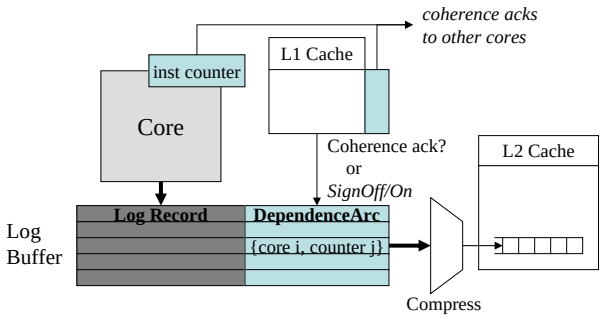

Figure 3: Components for producing dependence arcs. As coherence messages arrive, they may generate Dependence Arcs, which are associated with pending Log Records. As instructions retire, their associated records are committed to the log.

ware support. While our descriptions in Section 3.1-3.3 assume Sequential Consistency (SC), in Section 3.4, we extend our solution to support Total Store Ordering (TSO).

3.1 Capturing and Enforcing Application Event Dependences

Our design takes advantage of cache coherence messages for recording fine-grain dependences among application events, which is inspired by Xu et al.'s work on the Flight Data Recorder (FDR) [33] for post-mortem deterministic replay. However, as discussed earlier, dynamic parallel monitorning has very different requirements from deterministic replay, posing new research challenges. In the following, we study two challenges: (i) per-application dependence tracking, as opposed to system-wide tracking; and (ii) dependence enforcing mechanisms for supporting a diverse range of lifeguard operations.

Mechanisms for Capturing Fine-Grain Event Dependences for a Monitored Application. The LBA producer components (Figure 1) collect information of every dynamic *event* in the monitored application, including retired instructions, system calls, and important library calls (such as `malloc/free`). We extend LBA producer components to record happened-before dependences exposed by cache coherence activities, as depicted in Figure 3.

Similar to FDR, we augment each processor core with a counter that is incremented by one when an instruction retires (generating a log record). 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 with cache coherence messages. Our mechanisms record dependence arcs at the arc head core. For example, if an arc represents that core k's instruction I_k happened before core j's instruction I_j, then the arc is recorded at core j. Note that LBA producer components already enhance each core with a log record ID counter. Besides retired instructions, it is also incremented for annotation records injected by software (such as for system calls and important library calls). Since it is only required that the FDR counter monotonically increases, we reuse the log record ID counter rather than including a separate counter.

However, rather than tracking system-wide dependences in an always-on mechanism such as FDR, we track dependences for monitored applications and generate log entries only when the monitored applications are running. Logging application events only is important for two reasons. First, for security reasons, user-mode lifeguards that monitor an application should not see the activities of the operating system (OS) kernel or other unrelated processes. Second, system-wide logging consumes larger amounts of space and requires more work at the log consumers than per-application logging.

To realize per-application dependence tracking, we make the per-core counter value to be part of the thread state maintained by the OS. By saving and restoring this value at context switch time, the OS ensures that the value reflects the total number of log records generated since application launch.

Moreover, we need to guarantee that the counter field associated with each cache block correctly reflects the per-application last-visit time. This invariant is maintained provided that the monitored application process does not share main memory with other processes. However, processes may share memory under two situations. First, during

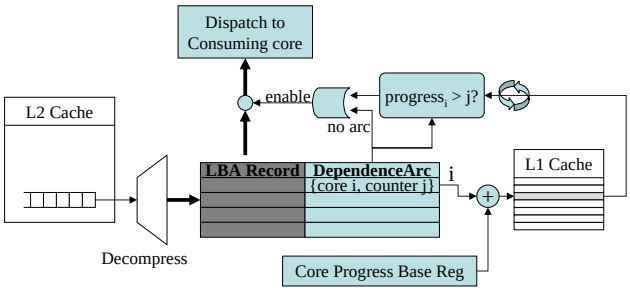

Figure 4: Components for coordinating consumption of dependence arcs. As Log Records are decompressed from the log, each is checked if it has an associated Dependence Arc. If not, it may be dispatched for consumption. If so, a check must be made to determine if the dependence has been resolved.

paging, application A's main memory pages may be swapped out and reused by application B. In this case, the operating system will perform a TLB shoot-down operation to ensure that the old page mapping does not exist, and we piggyback a "clear all counters" event on such operations to maintain the counters correctly. In the second case, application A shares memory with application B intentionally, such as by using System V shared memory. In this situation, lifeguards must monitor A and B as a unit as they now share address space. Therefore, A's threads and B's threads are grouped together and monitored as a single logical process.

Enforcing Dependences at Log Consumers for Lifeguard Operations. In our system, each running application thread is paired with a lifeguard thread, and the logs connecting such pairs include dependence records reflecting the ordering observed between the application threads. Concerted delivery of log records at multiple cores is required so that the lifeguard threads observe application events in the correct order.

Upon seeing a dependence arc (e.g., representing that core k's instruction I_k happened before core j's instruction I_j), a log consumer performs the following actions: (i) checks whether the log record corresponding to I_k has already been delivered; and (ii) if not, stalls until I_k is delivered.

To support the actions, log consumers need to "advertise" the progress on log record delivery in such a way that any log consumer can check the progress of the record delivery of all other consumers with low overhead. A software implementation would maintain a progress table in the lifeguard process's state, with one progress entry per core. By observing records of scheduling events, a lifeguard thread knows on which core (e.g., core k) the events being processed have occurred. Every time it processes a log entry, it updates entry k in the progress table. A lifeguard event handler can be triggered for every dependence arc, which checks the progress table. If the dependence is not satisfied, the lifeguard thread can poll the progress table until it is. Given that at any point in time there can only be one writer of a specific progress entry with monotonically increasing counts, lifeguard threads do not need to protect the table with locks. However, the table is accessed very frequently (for each log record). If a lifeguard thread is stalling, the progress entry will be bounced back and forth between the waiting thread and the thread updating the progress entry, incurring significant coherence traffic.

To remove the software overhead of maintaining this table, we propose the hardware coordination mechanism shown in Figure 4 and inspired by CNI [15]. The cores share a memory-mapped table of progress counters; counter k contains the counter value corresponding to the last delivered log event that has occurred on application core k. Each core maintains a hardware pointer to this table (the *Core Progress Base Register*). As the log consumer processes Dependence Arcs, it extracts the core id, i and counter value, j. Using i as an offset into the table, the hardware can determine the current counter value for core i, $progress_i$, and determine whether it is greater than j. If so, the local event may be delivered; if not, the consumer spins until the desired progress value is reached. Note that each core's counter will be maintained on a separate cache line to avoid excessive coherence traffic.

3.2 Efficiently Handling Dependences for System Calls and Logical Races

The baseline mechanisms described in Section 3.1 assume no system calls and no logical races. Here, we extend our mechanisms to handle both. Note that because we do not generate logs during system calls, any happened-before arcs

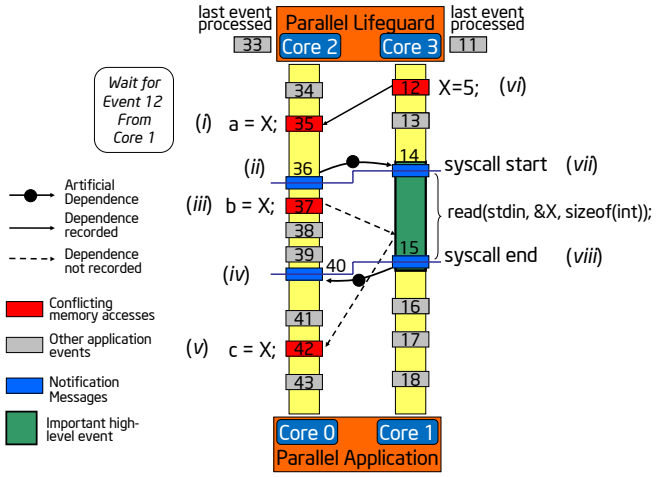

Figure 5: In a four-core CMP system, two application threads are monitored by two lifeguard threads. There is a log for every pair of application/lifeguard threads. Rectangle boxes (with numbers) represent application events (with the recorded counter values). Solid arcs are recorded but dotted arcs are missing. The blue boxes show the proposed mechanism for addressing the problem.

between system call activities and monitored application activities are missing. Moreover, there are no dependence arcs that track logical races because such races are not reflected in cache coherence messages.

Figure 5 illustrates this problem with an example in a four-core system. Two application threads are being monitored by two lifeguard threads. The application thread running on core 1 executes a `read` system call, which overwrites variable X in the operating system kernel. The thread running on core 0 loads from variable X before the system call (event i), concurrently with the system call (event iii), and after the system call (event v). However, since operating system activities are not logged, the dependence information is missing from the logs, which could lead to out-of-order event delivery at log consumers.

Generating Artificial Dependences through *SignOff* and *SignOn* Messages. To address this problem, we propose to generate artificial dependence arcs as shown in Figure 5. Note that LBA already provides a way to instrument system calls and important library calls (via wrapper library) for the purpose of inserting annotation log records for these high level events. Here, we extend this mechanism. At the start (end) of the system call S, the log producer core broadcasts a *SignOff* (*SignOn*) message to all the other cores, indicating that it is about to leave (return to) normal processing. The message contains the current local counter value and information about the call S (such as the memory range that may be accessed by S). A receiver of the message (1) enqueues all the pending log entries for retired instructions (e.g., by draining the store buffer), (2) inserts a *SignOff* (*SignOn*) log record with all the fields in the message, and (3) sends an acknowledgment message back to the sender. The acknowledgment message contains the receiver's local counter value. Meanwhile, the sender blocks until it gets the acknowledgments from all the other cores, then logs all the acknowledgments into its log.

Note that the above procedure of sending and logging messages is the same for both *SignOff* and *SignOn*. The difference lies in the actions taken at the *consuming* core. *SignOff* is regarded as dependence arcs from receivers to the sender, while *SignOn* is treated as dependence arcs from the sender to all receivers. As shown in Figure 5, *SignOff* adds the arc (ii to vii), while *SignOn* adds the arc (viii to iv). In this way, events that happened before the system call (event i) or after the system call (event v) are correctly ordered.

However, what about the dependent events (e.g., event iii) concurrent to the system call? Note that these dependent events must be true data races in the multithreaded application. We call them *racing events*. A properly synchronized application would synchronize these events with the system call invocation; such synchronization would lead to cache coherence messages between user-level threads and dependence arcs recorded by our scheme. For the purpose of dynamic monitoring, we are interested in identifying and reporting racing events to lifeguards, and allowing lifeguards

to decide how to handle them (e.g., stopping monitoring, flagging errors, assuming conservative outcomes, or ignoring them).

Detecting Racing Events. A *SignOff* (*SignOn*) message carries the memory range information for the corresponding call. For example, a `read` system call specifies a memory range as input buffer to receive inputs. This information is used for detecting racing events at log consumers. One design is to add a hardware range table at each log consumer. Upon seeing a *SignOff* (*SignOn*) record, the recorded memory range is inserted into (removed from) the table. Then a log consumer checks memory addresses in log records against the range table before delivering the records. If an address hits in a range, then the consumer invokes a lifeguard event handler for dealing with the data race. However, this design requires a table entry per application thread. Because of limited hardware resources, the number of supported application threads is limited. An alternative design maintains the detailed range table in main memory, while the hardware range table maintains a summary of the detailed table, storing a limited number of aggregate super-ranges that cover all the existing ranges. The hardware table serves as a filter: If an address does not fall in any super-range, then it is definitely not a racing event; Otherwise, we may have false positives and need to look up the detailed table. In our experiments, for simplicity, we limit the number of application threads to be no more than the number of cores, and use the former simplistic design. We leave an investigation of the latter more sophisticated design to future work.

More Details. Logical races are handled in the same way as system calls. Here, we instrument important calls to extract the memeory range information required to monitor potential logical races (e.g., `malloc/free`). For dependence tracking purpose, the "start" of the function call is treated as the point where the memory range information is available. Therefore, the memory range can be included in the *SignOff* message.

Application threads may have been scheduled off when a *SignOff* (*SignOn*) message is broadcasted. To deal with this situation, messages that cannot be delivered are recorded in a main memory buffer. When the application threads are scheduled on, we insert the missing *SignOff* (*SignOn*) messages that are still relevant into their logs.

3.3 Lightweight Synchronization for Lifeguard Metadata Accesses

In our setting, multithreaded applications are monitored by multithreaded lifeguards. Since metadata (e.g., tainted bit per application byte) are global states, lifeguard metadata accesses must be properly synchronized. A naive approach is to use locks to protect metadata accesses—obtaining a lock at the entrance of an event handler, and releasing the lock at the exit of the event handler. However, frequent lifeguard event handler code paths are typically composed of only a few instructions (e.g., less than ten instructions) [4]. Therefore, the locking overhead of the naive approach is unacceptable. In this paper, we propose a lightweight metadata synchronization approach based on the following observations for a wide range of lifeguards [19, 20, 23, 26]:

- There is a one-to-one mapping from application data accesses to lifeguard metadata accesses. For example, TAINTCHECK performs a propagation operation based on an application instruction. It accesses the metadata corresponding to the source and destination operands of the instruction.
- Dependent lifeguard metadata accesses correspond to dependent application data accesses. In general, this may not be true because the data granularity is often different from metadata granularity (e.g., ranging from 8-to-1 mapping in TAINTCHECK to 1-to-2 mapping [4]). Therefore, the accesses to a single 4-byte metadata word may result from accesses to different 4-byte application words. However, the observation is true with our dependence tracking design, which is based on cache coherence messages. This is because event dependences are tracked at cache line granularity (e.g., 64 bytes). Therefore, accesses to two different lines will result in accesses to different metadata words.
- Frequent lifeguard handler paths only perform metadata loads for data loads, and can perform metadata loads or stores for data stores. For example, this is clearly the case for TAINTCHECK.

Given these observations, frequent event handler paths will be automatically protected by our dependence enforcing mechanism, which invokes lifeguard event handlers in a proper order according to application data dependences. Therefore, in our lightweight metadata synchronization, we only need to protect the slow paths and infrequent event handlers with locks, while relying on our proposed hardware mechanism for protecting the frequent handler paths.

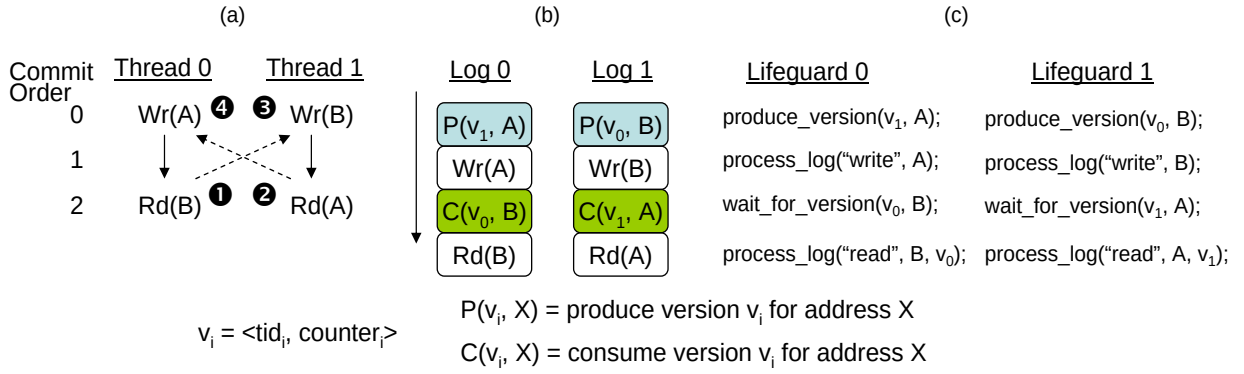

Figure 6: Supporting TSO. (a) An example of accesses that generate a cycle of dependences when coherence activity is used to infer ordering. (b) The contents of the log with entries for application instructions and annotations. (c) Actions taken by the lifeguards when processing the log.

3.4 Supporting Total Store Ordering (TSO)

Using coherence activity to infer the ordering of concurrent events in the system has the limitation of generating cycles of dependences in non-SC memory models. Previous work on deterministic replay provided an extension to handle TSO that entailed recording data values every time a processor is observed violating SC semantics by other processors in the system [34]. Under this scenario, 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 insufficent to calculate correct metadata. In TAINTCHECK, for example, the data value alone fails to record the source of the data, 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*.

Our TSO solution allows us to reverse non-SC $R \rightarrow W$ arcs to become SC $W \rightarrow R$ arcs, allowing forward progress. We identify all the pairs of $R \rightarrow W$ conflicting instructions where the load violates SC, as proposed in [34]. For each arc from thread i to thread j, we require the writer to create a copy of the previous metadata accessible by thread i. We require that the reader, thread i, request a copy of the correct metadata before analyzing the read. To implement this, we first do not record the $R \rightarrow W$ dependence. Next, we use version numbers formed by combining thread IDs with the current counter. We symbolically create a $W \rightarrow R$ arc by inserting a "produce version" annotation before the write in thread j's log, and inserting a "consume version" annotation before the read in thread i's log.

Figure 6 shows the scheme as it affects (a) the application, (b) the log and (c) the lifeguard, all under TSO. In Figure 6(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 log entry for Rd(b) is annotated with the version (<0,2>) of metadata the lifeguard should read before processing.

The log producing 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 log entry for the Wr(b) instruction so the lifeguard generates the versioned metadata for address B before overwritting metadata. The logs for threads 0 and 1 appear in Figure 6(b). Figure 6(c) shows the lifeguards processing their logs. 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.

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	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 [3]

Table 1: Simulation Setup

4 Experimental Setup

Simulation Setup We use the Simics [31] full-system simulator to model different configurations of sequentially consistent shared-memory CMP systems. We implement parallel LBA by extending Simics with log record capture and event dispatch support. We assume a perfect dependence tracking mechanism based on RTR [34], modified as described in Section 3.1. Table 4 shows the simultation 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 number of banks utilized are provided by CACTI [12], 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 SPLASH-2 [32] benchmark suite, as well as from the recently released PARSEC [1] suite. Specifically for PARSEC, we chose the benchmarks, for which the exact number of generated threads can be controled by an input parameter, and their runtime is not prohibitevly 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 4 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 every 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, in our case we are confident that lifeguards' L1 data cache states are warmed up during performance measurements for two reasons: first, the ratio of metadata size per application byte is small (in our case 2 bits per byte), resulting in much smaller working sets and second, the measured lifeguard miss rates are consistently 10 to 100 times lower than application miss rates, across benchmarks. Even if our approach is not accurate enough, this will only result in having a slow lifeguard, which will only make the performance results to be pesimistic.

Lifeguard Implementation. To evaluate the performance of our approach, we implemented a parallel version of TaintCheck [23]. For performance reasons, we associate 2 metadata bits per application byte, so that frequent cases

Benchmarks	Input
barnes	16K bodies
ocean	Grid size: 258 x 258
lu	Matrix size: 1024 x 1024
blackscholes	simlarge
fluidanimate	simlarge
swaptions	simlarge

Table 2: Benchmarks used and their input sets.

(application reads/writes 1 word/double) can be handled efficiently (lifeguard reads/writes 1 byte/word). The metadata 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. Due to TaintCheck's property to reflect application read events to metadata read and application write events to metadata write, 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.

5 Experimental Evaluation

We now evaluate the performance of our parallel dynamic monitoring scheme. Figure 7 shows the total execution time for three different schemes for a system that monitors parallel applications with 2, 4 and 8 threads: TIMESLICED, PARALLEL, and NONE. The TIMESLICED scheme models the state-of-the-art approach, which is to time-slice all of the monitored application's threads onto the same processor in order to aquire the application's inter-thread dependence orderings. In the TIMESLICED case, the application is monitored by a single lifeguard thread that runs on a separate processor. The PARALLEL scheme is our parallel monitoring approach (which also includes the LMA optimization from Chen *et al.* [4] to accelerate metadata accesses), where the application runs in parallel on k processors, and the lifeguard also runs in parallel on a separate set of k processors within the same CMP. As a point of comparison, the NONE scheme simply shows the application running in parallel on k processors without any monitoring.

As we see in Figure 7, our PARALLEL monitoring approach runs dramatically faster than today's TIMESLICED approach, achieving speedups of roughly twofold when the application has two parallel threads, and increasing to 5X or more when the application has eight threads. (Note that the relative speedups can be superlinear—e.g., 61X with 8 threads in BARNES—due to cache working set effects.) 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 to accelerate both the application and the lifeguard. Compared with running the application along on the system without monitoring (i.e. the NONE scheme), both monitoring schemes are slower, as expected, due to the significant amount of computation performed by the lifeguards to do their checking.

To better understand the overheads of our PARALLEL monitoring scheme, Figure 8 shows a breakdown of the execution time of the lifeguards into the following 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. As we see in Figure 8, the lifeguards spend the bulk of their time doing useful work: 85% or more in all cases except LU (which we will revisit shortly). The time spent stalled waiting for data dependences from other lifeguards is small: less than 10% in all cases. This suggests that our scheme for synchronizing the lifeguards is relatively efficient. Finally, the fraction of time when the lifeguard's log buffer is empty is very small in most cases with the exception of LU, where it is significant. The issue in LU is that the application threads stall for signficant amounts of time at barrier synchronization, and when they stall, there are no events in the log buffer for the corresponding lifeguard thread to process. This latter stall time is due to the synchronization characteristics of the monitored application, and is beyond the scope of our inter-thread dependence tracking mechanism for the lifeguards.

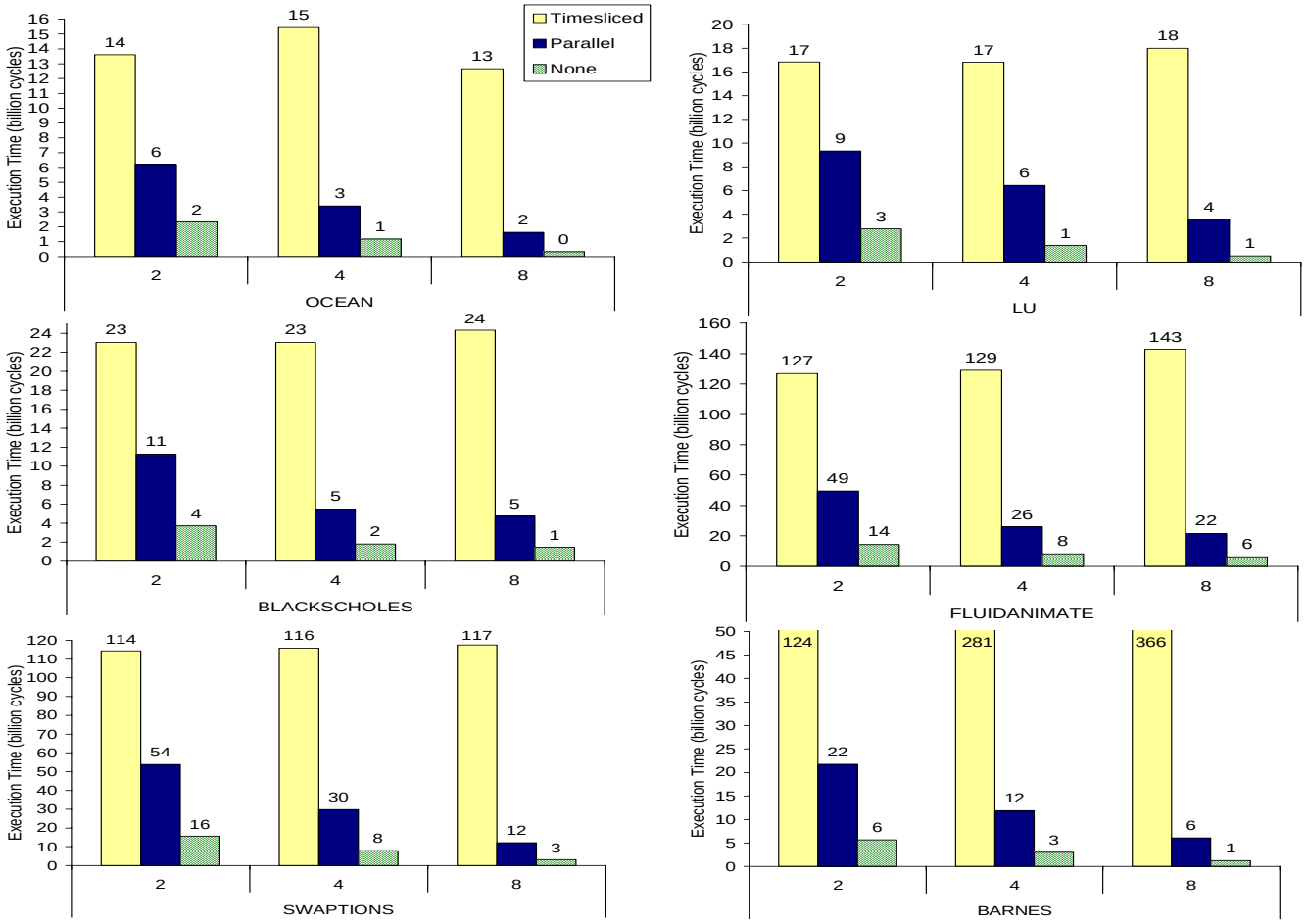

Figure 7: Execution time for three different monitoring schemes for 2, 4 and 8 application threads.

6 Conclusions

This paper presented a framework for dynamic parallel monitoring of multithreadead applications. By leveraging cache coherence to track inter-thread dependencies among application threads, and the use of *SignOff* messages to reason about the order of high-level application events, this framework is able to acquire a sufficient ordering of application events, required for correct monitoring.

We implemented a parallel version of TAINTCHECK to demonstrate the applicability of our technique, and to evaluate our system. We show that *parallel monitoring* is a key requirement for efficient monitoring of parallel applications, while previous approaches of serializing application threads on a single core are inherently non-scalable and extremely expensive. Our technique takes advantage of available parallelism and achieves a speedup of 1.19 to 2.05 (from 2 to 4 threads) and 2.28 to 4.5 (from 2 to 8 threads). Relative to the application, the slowdown ranges from 2.6x to 9.4x, remaining at least 5x lower than other state-of-the-art monitoring tools.

References

[1] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

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

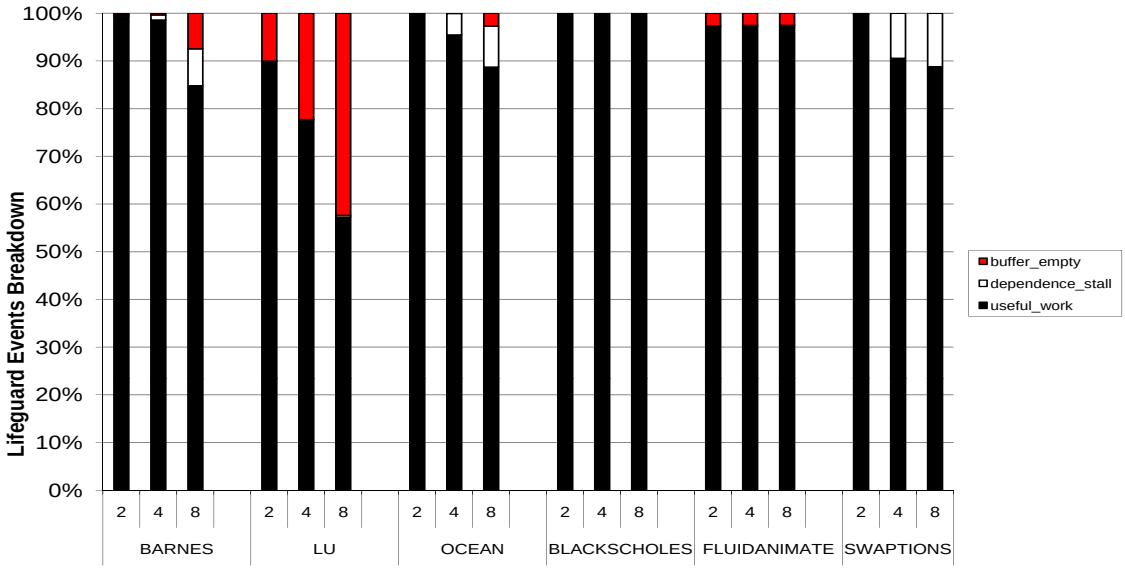

Figure 8: Breakdown of execution times for the PARALLEL monitoring case.

[3] Shimin Chen, Babak Falsafi, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R. Ganger, Bin Lin, and Steven W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *ASID Workshop at ASPLOS*, 2006.

[4] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *ISCA*, 2008.

[5] Jaewoong Chung, Michael Dalton, Hari Kannan, and Christos Kozyrakis. Thread-safe dynamic binary translation using transactional memory. In *HPCA*, 2008.

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

[7] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO-37*, 2004.

[8] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *ISCA*, 2007.

[9] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Engineering*, 27(2), 2001.

[10] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *USENIX ATEC*, 2006.

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

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

[13] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[14] Pablo Montesinos, Luis Ceze, and Josep Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *ISCA*, 2008.

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

[16] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *ASPLOS-XII*, 2006.

[17] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, 2005.

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

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

[20] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *VEE*, 2007.

[21] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[22] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *PADD*, 1993.

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

[24] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *ASPLOS*, 2008.

[25] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing Dynamic Information Flow Tracking. In *SPAA*, 2008.

[26] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multi-threaded programs. *ACM TOCS*, 15(4), 1997.

[27] 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.

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

[29] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. Analyzing dynamic binary instrumentation overhead. In *WBIA Workshop at ASPLOS*, 2006.

[30] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. MemTracker: Efficient and programmable support for memory access monitoring and debugging. In *HPCA-13*, 2007.

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

[32] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

[33] Min Xu, Rastislav Bodik, and Mark D. Hill. A 'Flight Data Recorder' for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[34] Min Xu, Rastislav Bodik, and Mark D. Hill. A regulated transitive reduction (RTR) for longer memory race recording. In *ASPLOS*, 2006.

[35] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *HPCA-13*, 2007.

[36] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iwatcher: Simple, general architectural support for software debugging. *IEEE Micro*, 24(6):50–56, 2004.

[37] Yuanyuan Zhou, Pin Zhou, Feng Qin, Wei Liu, and Josep Torrellas. Efficient and flexible architectural support for dynamic monitoring. *ACM TACO*, 2(1), 2005.