Log-Based Architectures: Using Multicore to Help Software Behave Correctly

Shimin Chen* Phillip B. Gibbons* Michael Kozuch* Todd C. Mowry*,†
*Intel Labs Pittsburgh †Carnegie Mellon University

ABSTRACT

While application performance and power-efficiency are both important, application correctness is even more important. In other words, if the application is misbehaving, it is little consolation that it is doing so quickly or power-efficiently. In the Log-Based Architectures (LBA) project, we are focusing on a challenging source of application misbehavior: software bugs, including obscure bugs that only cause problems during security attacks. To help detect and fix software bugs, we have been exploring techniques for accelerating dynamic program monitoring tools, which we call "lifeguards". Lifeguards are typically written today using dynamic binary instrumentation frameworks such as Valgrind or Pin. Due to the overheads of binary instrumentation, lifeguards that require instruction-grain information typically experience 30X-100X slowdowns, and hence it is only practical to use them during explicit debug cycles. The goal in the LBA project is to reduce these overheads to the point where lifeguards can run continuously on deployed code. To accomplish this, we propose hardware mechanisms to create a dynamic log of instruction-level events in the monitored application and stream this information to one or more software lifeguards running on separate cores on the same multicore processor. In this paper, we highlight techniques and features of LBA that reduce the slowdown to just 2%–51% for sequential programs and 28%–51% for parallel programs.

Categories and Subject Descriptors

C.0 [**General**]: System Architectures; D.2.5 [**Software Engineering**]: Testing and Debugging–Monitors

General Terms

Algorithms, Design, Experimentation, Reliability, Security

Keywords

Program monitoring, software bugs, lifeguards, log-based architectures, parallel monitoring

1. INTRODUCTION

The Log-Based Architectures (LBA) project is a joint research effort involving researchers at Intel Labs Pittsburgh and Carnegie Mellon University. The inspiration for the project was the observation that with the industry-wide shift to multicore processing, parallel programming is the path to high performance in the immediate and forseeable future, but parallel programming is also notoriously error-prone. History has taught us that as difficult as it is to avoid software bugs in sequential code, bugs are even more problematic in parallel code.

In addition to the obvious frustration that software bugs cause for end users when the system misbehaves, we are concerned about the impact of bugs for two other reasons. First, even obscure bugs that would rarely cause problems under normal circumstances may be exploited by hackers as *security vulnerabilities*. Second, although large data centers can successfully use redundancy to make themselves fairly resilient to *hardware* failures, the replicated nature of *software* means that *a single bug can potentially take down an entire data center all at once*.

The traditional approach to identifying the root cause of a software bug is that once the application crashes, the programmer then attempts to re-run the software—this time with debugging tools turned on—and reproduce the same conditions that led to the problem. Unfortunately, for parallel software (and especially in an interactive networked environment), it may be extremely difficult (and perhaps even impossible) to reproduce the exact timing and sequence of events that triggered the bug. Although specialized software testing platforms might be built to carefully control the timing and interleaving of system events, a number of important bugs do not reveal themselves until the software is run at full-scale on a live production system. Hence our goal is to diagnose the root cause of a problem the *first time* that it occurs on a live production system.

To accomplish this goal, we have developed a new approach for executing *instruction-grain lifeguards* [22, 24, 25, 27, 30], which are software tools that run dynamically (i.e. online) with the application, performing sophisticated instruction-by-instruction analysis of the running application to identify bugs and sometimes even repair them (or limit their damage). Compared with static tools that analyze code before it executes [4, 11, 12], 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 [20, 38, 39], 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*. These tools are typically implemented today using a dynamic binary instrumentation (DBI) framework [2, 18, 24], and the corresponding slowdowns for the monitored applications often range from 30X to 100X or more [24, 31].

We aim to dramatically reduce the run-time overhead of instruction-grain lifeguards to the point where it would be practical to run them

on live production systems to find software bugs. We propose a *log-driven approach* whereby the hardware captures a per-instruction log from a monitored application and streams it to another core on the same chip, where it can be processed on-the-fly by any software lifeguard. Efficient log streaming is achieved by writing and reading compressed log records into a circular memory-mapped buffer that resides in on-chip cache memory. Through a combination of novel hardware and software support, we have been able to reduce the overhead of instruction-grain lifeguards by more than an order of magnitude to a 1.02X-1.51X slowdown for sequential applications and a 1.28X-1.51X slowdown for parallel applications.

While there have been a number of hardware proposals for accelerating a specific class of lifeguards (e.g., lifeguards for memory-access monitoring [32,36,41], data-race detection [40], or information-flow tracking with simple metadata [9,10,33,35]), these mechanisms are useful only for the narrow class of lifeguards that they support. LBA, in contrast, provides a framework for accelerating arbitrary lifeguards.

The remainder of this paper is organized as follows. Section 2 provides background on the structure and requirements of instruction-grain lifeguards. Section 3 describes our log-driven architecture. Sections 4 and 5 describe our techniques for improving the performance of lifeguards and enabling them to efficiently monitor parallel software, respectively. Finally, Section 6 presents conclusions.

2. LIFEGUARDS

We focus on the following diverse instruction-grain lifeguards that detect memory violations, security exploits, and data races.

AddrCheck checks whether every memory access is to an allocated region of memory [21]. By intercepting memory allocation routines such as `malloc` and `free`, AddrCheck maintains one-bit of metadata for each byte of the monitored application's address space that indicates whether or not the byte is currently allocated. AddrCheck checks these metadata for every memory access, and raises an error if the accessed byte is not currently allocated.

MemCheck extends AddrCheck to detect also the use of uninitialized values [22,23]. For this purpose, it maintains one "initialized" bit per address byte, in addition to the "allocated" bit. It also maintains state per register indicating which bytes in the register contain "initialized" values. A memory load of an uninitialized value is not an error in itself (for example, copying a partially initialized structure). Rather, MemCheck raises an error only if uninitialized values are dereferenced as pointers, used in conditional tests, or passed into system calls. To achieve this, MemCheck tracks the propagation of uninitialized values in the monitored application: For every executed instruction, the destination becomes uninitialized if at least one of the sources is uninitialized.

TaintCheck detects memory overwrite-related security exploits, such as buffer overflow and format string attacks [25]. The metadata consist of (i) one "tainted" bit per address byte of the monitored application and (ii) state per register indicating which of its bytes are "tainted". TaintCheck marks all unverified program input data, such as data from the network, as suspect, or *tainted*. Subsequently, it carefully tracks the propagation of tainted data through the application: For every executed instruction, the destination becomes tainted if at least one of the sources is tainted. It raises an error if the application uses tainted data in critical ways, such as in jump target addresses, printf-like format strings, or system call arguments. We also study a TaintCheck variant that records a history of the taint propagation, using an 8-byte metadata structure (4-byte "from" address, 4-byte instruction pointer) per 4-byte application word. Upon detection, this lifeguard can reconstruct a taint propagation trail.

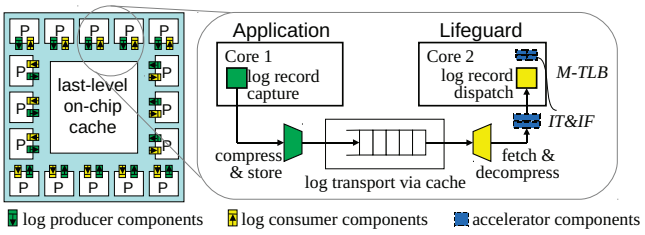

Figure 1: Overview of LBA's logging mechanisms.

LockSet detects data races by checking whether the monitored application follows a consistent locking policy [30]. For each thread t, LockSet maintains the current set S_t of locks held by the thread. For each shared memory location m, it maintains a candidate set S_m of locks. LockSet knows m to be a shared location if a second thread accesses it; at this moment, S_m is initialized with the current lock set of the second thread. Afterwards, whenever a thread t references m, S_m is set to $S_m \cap S_t$. If S_m ever becomes empty, no consistent common lock set protects accesses to m, and LockSet raises an error. A LockSet structure is a list of lock addresses. For every 4-byte word in the monitored application, the metadata are a 32-bit record consisting of a compressed 30-bit pointer to the actual LockSet and a 2-bit state (indicating virgin, exclusive, shared read-only, or shared read-write [30]) for the location.

Common Lifeguard Characteristics. From the above lifeguards and others we have studied, one can extract three common lifeguard characteristics:

C1. A lifeguard maintains a data structure that records state information ("metadata") about the monitored application's address space (e.g., which addresses have been allocated or tainted). There is a 1-1 mapping between application data and lifeguard metadata at some granularity (e.g., each application byte maps 1-1 to a metadata bit).

C2. A lifeguard is interested in observing many, but possibly not all, application execution events. Some of the events are needed solely to maintain the metadata. For other, "interesting" events, the lifeguard checks the current metadata state, and reports an error if an anomaly has occurred.

C3. There is a mapping from monitored application execution events to specific lifeguard functionality ("handlers") based on the event type (load, store, etc.); these handlers are invoked in response to the sequence of execution events.

3. LBA ARCHITECTURE

The general-purpose, hardware-supported logging mechanism that we propose for LBA is well-matched to the lifeguard characteristics described above (**C1**, **C2**, and **C3**). Figure 1 illustrates the main components of this design. The components shown are sufficient to monitor single-threaded applications and will be described in this section. Additional mechanisms are needed to track the ordering of events in a multithreaded application running in parallel on multiple cores; these will be described in Section 5.

The zoom-in portion of Figure 1 depicts that a software lifeguard running on core 2 is monitoring an unmodified application running on core 1. Also shown in the picture are four sets of components: log producer, log transport, log consumer, and accelerator. The optional accelerator components are described in Section 4.

3.1 Log Production

As each application instruction retires, the core on which the application is executing (core 1 in Figure 1) captures a *log record* for that instruction event (**C2**). Instruction event records are tuples consisting of the program counter, the instruction type (**C3**), the virtual memory address of any memory operands (**C1**), and an indication of which operands were sources and which were destinations. LBA also includes support for software-inserted *annotation records*. This type of record is injected into the log by wrapped shared library calls, for example, to indicate high-level events of interest to the lifeguard (e.g., malloc library calls).

The stream of log records is compressed using a value predictor based (e.g. [3]) compression engine. The resulting log records may average less than one byte per record [5].

3.2 Log Transport

The mechanism for delivering the stream of compressed log records to the lifeguard core may vary by implementation. For example, a dedicated interconnect could be used to transport the log, if available. In our current design, however, we assume such specialized transport assists are unavailable. Rather, we leverage an existing data channel connecting the application and lifeguard cores, namely the shared on-chip cache.

In a multicore processor, shared on-chip cache structures can be used for the log transport channel. The compressed stream is written to a circular memory-mapped buffer allocated by system software, which fits within an appropriately small fraction of the largest physically-shared on-chip cache. Delivery is effected when the decompression engine on the lifeguard core reads the log data back out of this buffer. Assuming a sufficiently associative write-back cache, this memory-backed design enables easy system management of log buffers (e.g. across context switches) without placing undue pressure on off-chip memory bandwidth. To reduce pollution of L1 caches, an implementation should provide special log buffers to bypass the L1 caches or use cache accesses with explicit replacement hints [17]. To manage producer-consumer coordination of the log, we employ the techniques described by Mukherjee *et al.* [19].

3.3 Log Consumption

After transport, the log is decompressed and delivered to the lifeguard core (core 2 in Figure 1) where the uncompressed event records are placed into a log dispatch buffer. These records are then processed, in-order, by passing each event record to an appropriate handler function in the lifeguard (**C3**). While the lifeguard could fetch each record, decode it, and dispatch it to the appropriate handler, such a software-only fetch-and-decode loop would slow down lifeguard processing.

As an alternative, we propose that the LBA hardware provide a mechanism for event-driven execution. To enable this, the lifeguard populates an *event type configuration table (ETCT)* and registers it with the lifeguard core. This table is indexed by the log event type and contains an event handler entry point for every interested log event type. Every handler ends with a special instruction: `nlba` (next LBA event). This proposed instruction indicates that the previous handler has finished and the core should now jump to the handler address associated with the next event. Certain event values (such as application virtual addresses) that are inputs to the lifeguard handler are automatically placed in registers for ready handler access.

Note that the design shown in Figure 1 enables the application and lifeguard to operate asynchronously. In normal operation, the log buffer contains at least a few unprocessed records; the ETCT design enables the microarchitecture to take advantage of this situation by reading ahead in the log and determining the next handler(s) that will be invoked—thereby avoiding potential mispredicted-branch penalties.

Each invoked lifeguard handler consists of arbitrary code to process an event of a specific type, subject to the above input conventions and ending with `nlba`. By **C1**, the typical handler uses the (application) virtual addresses and registers in the (application) event to compute corresponding (lifeguard) metadata addresses and access the metadata. We use an array to store the metadata for registers, and a two-level table to store the metadata for virtual addresses, both allocated in the lifeguard's address space. The two-level table is an array of pointers to subarrays of metadata values. The memory footprint of the metadata is reduced by allocating only subarrays corresponding to virtual addresses allocated by the application. Moreover, because most lifeguards use only 1 or 2 bits per application byte, the memory footprint and the working sets of the lifeguard are typically smaller by a corresponding factor of 8 or 4. The smaller working sets imply a corresponding decrease in the cache misses and memory bandwidth consumption of the lifeguard relative to the application. Finally, because the application and lifeguard run on separate cores, they do not compete for per-core resources such as L1 caches. All these reasons serve to reduce the lifeguard's processing time and its impact on application performance.

3.4 System Support

We rely on operating system support to manage several aspects of the system such as forming the associations between lifeguards and applications, allocating transport channels for the logs, scheduling the lifeguard and application processes efficiently, spawning child lifeguards to monitor child processes of the application, and providing mechanisms for containing failures in the monitored process. The operating system is also responsible for maintaining appropriate system protections. For example, less-privileged processes typically should not have access to the logs of more-privileged ones. In particular, logging is typically suspended when transitioning into the kernel and managed carefully during context switches.

Typically, when an application process is created, system software will determine if a lifeguard should be associated with it. If so, (i) a lifeguard process is created, (ii) the application's process control block is marked as *logging_enabled* and tagged with the events of interest identified by the lifeguard, and (iii) a log transport channel is allocated, and the channel's identifier is recorded in the process control blocks of the application and the lifeguard. Whenever the application process is subsequently scheduled for execution, prior to transferring control to the application, the logging hardware is enabled to record the specified events into the specified channel. It is advantageous to co-schedule the lifeguard process with the application process, although not strictly required.

An important aspect of our design is that the log records are typically buffered by the transport mechanism, enabling the lifeguard to make progress when the application has encountered a long latency event or vice-versa. One effect of this buffering is that the lifeguard (application) will stall if the finite buffer becomes empty (full). Another effect is that lifeguards may detect an event some non-trivial time after the event has occurred. This detection-delay may be acceptable for *detection-only* lifeguards that seek to merely provide the user with a warning that the violation of some invariant has been observed. For security lifeguards, however, the delay between violation and detection presents a problem; consequently, the system software executes security-sensitive applications in *containment* mode. In this mode, system software contains any damage that may be done by a corrupted application, by stalling each system call issued by the application until the lifeguard affirms that no corruption has been detected. Similar system call-based containment approaches can be found in the literature [13–15, 26].

3.5 Performance Gains from LBA's Logging Mechanism

The hardware-supported logging mechanism described in this section accelerates a broad class of instruction-grain lifeguards, by focusing on the characteristics common to many lifeguards (**C1–C3**). It reduces or eliminates most of the key overheads of software-only (DBI) solutions such as (i) overheads of running the application and the lifeguard functionality on the same core (e.g., stealing CPU cycles, saving/restoring registers, polluting L1 caches) and (ii) overheads of mimicing hardware state in software (e.g., program counter tracking, effective address calculation). Overall, our simulation study with CPU-intensive benchmark programs shows that LBA's logging mechanism reduces the performance overhead of instruction-grain lifeguards from 30X–100X to 3X–5X [5]. In the next section, we discuss optimizations that reduce this overhead even further.

4. OPTIMIZING PERFORMANCE FOR INSTRUCTION-GRAIN LIFEGUARDS

We have developed three approaches to reduce lifeguard overhead below the 3X–5X slowdowns achieved by LBA's logging mechanism. These approaches—(i) hardware accelerators [6, 7] (Section 4.1), (ii) compiler optimizations [28] (Section 4.2), and (iii) parallelizing the lifeguards [29] (Section 4.3)—optimize the performance of instruction-grain lifeguards, and are applicable beyond just the LBA framework (e.g., hardware accelerators are useful for DISE [8] and compiler optimizations for Valgrind [24]).

4.1 Hardware Accelerators

We identify three main sources of overhead for a wide range of instruction-grain lifeguards, and propose three flexible hardware accelerators for reducing the overhead, as depicted in Figure 2:

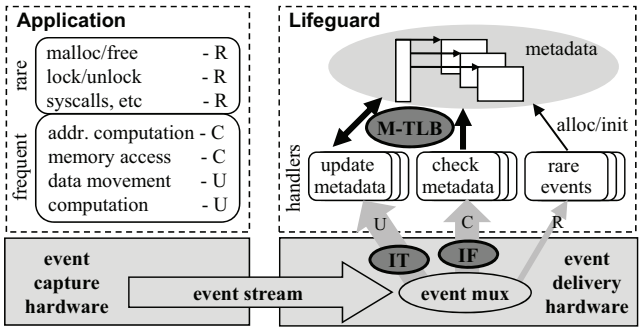

Figure 2: Accelerating lifeguards with IT, IF, and M-TLB.

- *Inheritance Tracking (IT) for Propagation-Style Metadata Updates*. For dynamic information tracking (DIFT) lifeguards (e.g., TaintCheck and MemCheck), one key source of overhead is propagation tracking [33]: Given an executed application instruction I, the metadata of I's destination location is computed as a combination of the metadata of all I's source locations. The challenge is to support a wide range of lifeguard metadata organizations. Note that propagating metadata values in hardware would be lifeguard-specific because the hardware would need to understand the specific metadata organization. Metadata propagation, on the other hand, follows the dataflow structure of the monitored program, which is lifeguard-independent. Moreover, we find that a more restricted version of propagation, unary propagation, can support many useful lifeguards (including TaintCheck and MemCheck). Based on these insights, we propose *Inheritance Tracking (IT)* that tracks unary data inheritance with a hardware shadow register table for effectively filtering out propagation events associated with the flow of data through registers.

- *Idempotent Filters (IF) for Redundant Metadata Checks*. For many lifeguards (e.g., AddrCheck, MemCheck, and LockSet), checking is performed on every memory reference and/or on every address computation. We observe that if the underlying metadata are not changed, metadata checks for "redundant" events will have the same result. Therefore, we propose *Idempotent Filters (IF)* to reduce lifeguard checking overhead. The idea is to introduce an IF cache of recently observed checking events. If an incoming event hits in the cache, it is discarded (filtered). An event is delivered to the lifeguard only upon a miss in the IF cache. To deal with metadata changes, lifeguards can configure the IF cache to be fully or partially invalidated upon certain events (such as `malloc` and `free`).

- *Metadata-TLB (M-TLB) for Metadata Mapping*. The third key source of overhead is the mapping from an application address to the corresponding metadata location(s). We observe that in many frequently invoked lifeguard handlers, the mapping often takes half of the executed instructions. We propose a hardware translation look-aside buffer mechanism, *Metadata-TLB (M-TLB)*, for solving this performance problem. M-TLB translates application-space virtual addresses to lifeguard-space (metadata) virtual addresses. Lifeguards can use a proposed instruction, `lma` (load metadata address), to look up an M-TLB entry associated with a given application-space virtual address. M-TLB resides in user space and is managed by lifeguards, minimizing the need for OS support: Upon an M-TLB miss, a configured software handler is called to insert new entries into the M-TLB.

We implemented these three accelerators within LBA. Our simulation study with CPU-intensive benchmark programs shows that the hardware accelerators can effectively reduce the performance overhead of lifeguards down to 2–51% for a wide range of lifeguards. Further details can be found in Chen *et al.* [7].

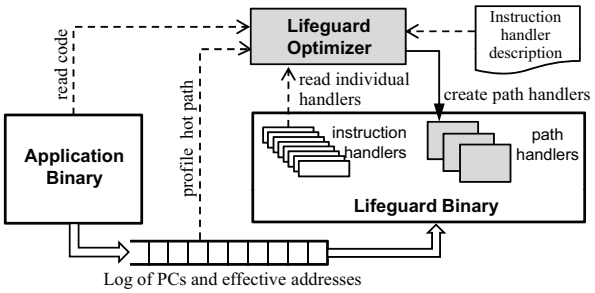

Figure 3: Integrating a JIT-style lifeguard optimizer.

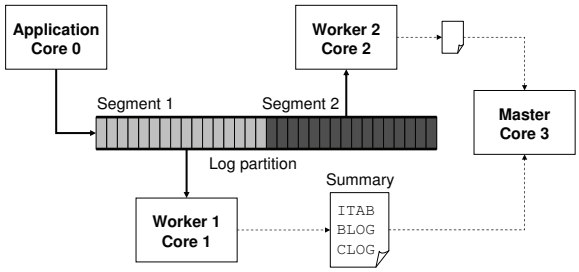

Figure 4: Parallelizing a DIFT Lifeguard.

4.2 Compiler Optimizations

We investigate an alternative approach to the above hardware techniques: recognizing and eliminating the redundancy in lifeguard checking code through compiler optimizations in *software*.

For decades, optimizing compilers have successfully improved software performance by reducing redundant computations along execution paths [1]. For traditional static analysis, a control flow graph is typically used to summarize the possible execution paths; for more recent JIT-style dynamic code analysis, the optimizations are typically applied to an observed set of hot paths.

We observe that a *hot path* (potentially spanning large numbers of basic blocks) in the monitored application triggers a frequently executed, fixed sequence of lifeguard event handlers. Therefore, we bundle all such event handlers into a single *path handler*. By exposing the lifeguard analysis associated with an entire hot path to our optimizer, we can find many more opportunities for eliminating redundant work in the lifeguard analysis.

In addition to exploiting the usual forms of redundancy elimination that are utilized by modern optimizing compilers, we explore the common properties of lifeguards and use *domain-specific knowledge about the lifeguard behaviors* to perform more aggressive optimizations. We identify and remove redundant checks on the same metadata (i.e., checks without any metadata changes in between). Because there is a one-to-one mapping from application data to lifeguard metadata, alias analysis for metadata can be effectively performed using the monitored application's data addresses. Moreover, we exploit spatial locality of metadata accesses and share portions of the metadata mapping computations across multiple metadata accesses. Finally, we apply the optimizations across path handlers especially for hot paths in loops.

As shown in Figure 3, we propose to incorporate a JIT-style lifeguard optimizer into the lifeguard supporting framework. The lifeguard optimizer obtains the application's hot control flow profiles from the log. It then reads the hot code paths from the application binary, reads the relevant instruction event handlers from the lifeguard binary, and composes and optimizes the appropriate sequences of event handlers into *path handlers*. The optimizer is designed to execute off the critical path of application-to-lifeguard communication and hence should have minimal adverse impact on application and lifeguard performance. A path handler is invoked whenever the corresponding hot path is at the head of the log; otherwise an appropriate instruction handler is invoked.

We incorporated our approach into both LBA's logging mechanism from Section 3 and a lifeguard platform based on dynamic binary instrumentation (Valgrind) [24]. We did not use the hardware accelerators described in Section 4.1. Perhaps surprisingly, we show that, even starting with well-optimized applications and hand-tuned lifeguard handlers, our optimizations can find and eliminate significant redundancy in both frameworks, achieving improvements of up to 31% on Valgrind and 53% on LBA. Further details can be found in Ruwase *et al.* [28].

4.3 Parallelizing Lifeguards

Given the high overheads for instruction-grain lifeguards, it is natural to consider whether these overheads can be significantly reduced by parallelizing the monitoring functionality. For certain lifeguards such as AddrCheck, the parallelization is straightforward because the monitoring work can be partitioned on a per-address basis and divided among multiple lifeguard cores. Program instructions that allocate or free blocks of memory may need to be broadcast to all the lifeguard cores, but other than these rare events, each lifeguard core can independently check memory accesses that fall within its assigned partition. Similarly, LockSet, although slightly more complicated, can be readily partitioned such that each lifeguard core independently checks accesses that fall within its partition.

Far more challenging to parallelize, however, are DIFT lifeguards (e.g., TaintCheck and MemCheck). These have strong serial dependencies among lifeguard event handlers because the event handlers perform metadata propagations that follow the data flow in the monitored application. To address this challenge, we partition the log into segments and then track the metadata propagation within a segment *symbolically* whenever the values are unable to be determined explicitly (because they are being computed concurrently by another lifeguard core).

Figure 4 depicts an overview picture of our parallel DIFT algorithm. The lifeguard is composed of a master thread and a number of worker threads, each running on a separate core. The log is conceptually divided into disjoint and equal-sized partitions. Each partition is further divided into k disjoint segments if there are k workers, where the ith worker is assigned the ith segment. All workers read their assigned segments in parallel (we assume that a lifeguard thread can selectively read portions of the log), and generate a data structure that symbolically summarizes propagation information in the segment. After processing a log segment, a worker passes the generated summary to the master thread, and waits until the next assigned log segment in the next log partition is available. The master thread combines the summaries according to the log order. By plugging in the explicit metadata values at the start of a segment, the master computes the final metadata values for every segment, updates lifeguard metadata, and performs the actual checks. As an

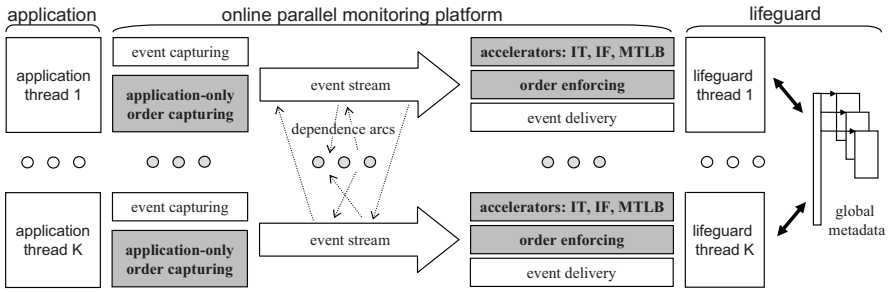

Figure 5: The ParaLog parallel monitoring platform, highlighting components with new features.

opimization, the master can also be assigned a segment at the head of the log to process in parallel with the workers.

We implemented a parallel taint analysis algorithm in LBA's logging mechanism (i.e., techniques from Section 3 but not Sections 4.1 nor 4.2) using this approach. Our simulation results on SPEC benchmarks and a video player show that, on a 16 core processor, our parallel algorithm reduces the lifeguard performance overhead to as low as 1.2X using 9 monitoring cores. Further details can be found in Ruwase *et al.* [29].

5. ENABLING AND ACCELERATING ONLINE PARALLEL MONITORING

Thus far, we have discussed techniques and mechanisms suitable for monitoring applications executing on a single core, either single-threaded applications or multithreaded applications that are time-sliced on a single core. In this section, we describe techniques and hardware mechanisms for monitoring *parallel* applications, i.e., multithreaded applications concurrently executing on multiple cores.

Prior to our work, the only practical way to monitor a parallel application using lifeguards was to time-slice the multiple application threads onto a single processor and analyze the resulting interleaved instruction stream sequentially. This resulted in unacceptably poor performance because neither the application nor the lifeguard could enjoy parallel speedups. We present two approaches enabling both the application and the lifeguard to operate in parallel with high performance. A key challenge addressed by these approaches is how to ensure that the lifeguard processing properly accounts for any inter-thread data dependences among application threads. The approaches differ fundamentally in how they solve this challenge: ParaLog (Section 5.1) uses an efficient hardware mechanism to log inter-thread data dependences, while Butterfly Analysis (Section 5.2) is a software-based approach adapting dataflow analysis techniques. Both can be viewed as extensions of LBA's solutions for monitoring single core applications, although the techniques apply more broadly.

5.1 ParaLog

ParaLog [37] is a general-purpose platform for online monitoring of parallel applications, as shown in Figure 5. Compared to Figures 1 and 2, there are $K > 1$ application threads running on K (application) cores, each producing an event stream (log) that is processed by a distinct lifeguard thread running on a distinct (lifeguard) core. 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: (i) capturing inter-thread data dependences between the application threads and reproducing their effects appropriately in the lifeguard processing, (ii) ensuring the multiple lifeguard threads atomically access the global metadata at negligible cost, and (iii) adapting the three hardware accelerators from Section 4.1 to work for parallel applications. We will discuss ParaLog's solution to each of these challenges in turn.

Application Event Ordering. Each application thread is monitored by a corresponding lifeguard thread. To reduce delays, each lifeguard thread greedily processes its application thread's log asynchronously with respect to other logs, 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 assume the application and lifeguard are running on a processor supporting the *sequential consistency (SC)* memory model, although our approach can be extended to the *total store order (TSO)* memory model. ParaLog piggybacks off cache coherence messages to capture inter-thread dependences. Unlike FDR [38] and related work on capturing inter-thread dependences, (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*. In our *application-only order capturing component* (see Figure 5), the starting point (thread ID, record ID) of a dependence arc is recorded in the log associated with the receiving end of the arc. Our *order enforcing component* enforces the captured arcs, without modification to the lifeguard-specific code, by (a) publishing the record ID of the most recent event record processed by each lifeguard thread, and (b) stalling a lifeguard thread at a dependence (t, i) until the published record ID for t is at least i.

There are two sources of dependences that are not captured by the above approach: (i) system calls because we do not capture (privileged) OS kernel activity, and (ii) races not reflected in cache coherence traffic (such as between freeing a block of memory and a read to an address within the block). Both sources are handled via a message broadcasting mechanism (called *conflict alerts*) that alerts all the other lifeguard threads at system call boundaries and potentially problematic events.

Metadata Access Atomicity. Lifeguard threads reading and writing the global metadata must be properly synchronized. Using locks or atomic instructions for each such access would be prohibitively expensive. Fortunately, for a wide range of lifeguards (basically, lifeguards such as TAINTCHECK and ADDRCHECK with a 1-1 mapping from data to metadata such that application reads translate only into metadata reads), no explicit synchronization is required to preserve metadata access atomicity in an event handler's frequent "fast path". This is because races between metadata accesses correspond to inter-thread application dependences, and hence our event ordering mechanisms already prevent the race. In some cases, a lock must be acquired on a handler's infrequent "slow path"; however, we show that correctness is ensured even though the fast paths acquire no locks.

Local Hardware Accelerators with Possible Remote Conflicts. The hardware accelerators discussed in Section 4.1 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 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. However, detecting conflicts is a challenge when monitoring parallel applications because an event at one application core may conflict with the accelerator state at other lifeguard cores. To properly handle such remote conflicts, we use (i) the *conflict alert* mechanism discussed above for high-level conflicts such as those involving malloc or free, and (ii) a lighter-weight *delayed advertising* approach for the more frequent instruction-level conflicts involving a single application write (e.g., that updates a taint status). In delayed advertising, the published record ID for a lifeguard thread t is not updated to i until record i's effect has been flushed from t's accelerator state, thereby ensuring accelerator correctness.

Our simulation study with CPU-intensive benchmark programs running on a 16-core processor (8 application threads, 8 lifeguard threads) shows that ParaLog reduces the performance overhead of ADDRCHECK and TAINTCHECK down to 1.28X and 1.51X, respectively, on average. Further details can be found in Vlachos *et al.* [37].

5.2 Butterfly Analysis

Butterfly Analysis [16] is a complementary approach to enabling online monitoring of parallel applications. Compared to ParaLog, it requires neither hardware support for tracking inter-thread data dependences nor processors with strong memory consistency models (SC, TSO), but it can result in some false positives in lifeguard analysis. (It never misses errors, but it can occasionally report a possible error when none exists.) Instead of relying on detailed inter-thread dependence tracking and SC/TSO, butterfly analysis relies only on the fact that in modern processors, any memory access in the distant past (1000s of instructions ago) has become visible to all the threads. It makes no assumptions on (and avoids the overheads of tracking) the relative ordering of more recent memory accesses by different threads. Instead, the analysis must account for this bounded window of uncertainty in the orderings of recent events by concurrent threads.

Butterfly analysis is a new framework for performing lifeguard analysis that automatically reasons about bounded windows of uncertainty using an approach inspired by *interval analysis* [34]. Unlike traditional dataflow analysis, which performs static analysis on control flow graphs, our approach analyzes *dynamic* traces of

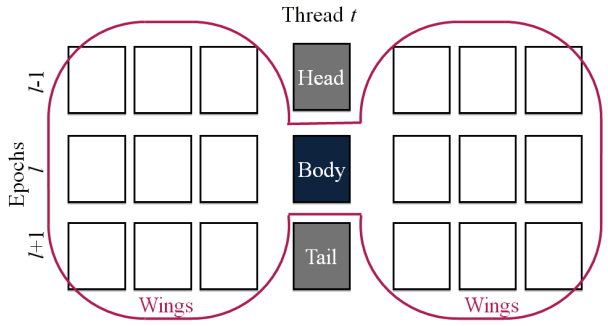

Figure 6: Potential concurrency modeled in butterfly analysis: instructions in the *Body* interleave with instructions in the *Wings*, the *Head* has already executed, and the *Tail* has not yet executed.

instructions on different threads. We adapt two techniques from static dataflow analysis, reaching definitions and reaching expressions, to this new domain of online parallel monitoring, in order to overcome the potential state explosion of considering all the possible orderings among events in each window of uncertainty. Significant modifications to these techniques are required to ensure the correctness and efficiency of the approach.

Heartbeats, Epochs, and Butterflies. We rely on a regular signal, or *heartbeat*, to be reliably delivered to all application cores and placed in the respective logs. This could be implemented using a software token ring. We do not require instantaneous heartbeat delivery, but do assume a maximum skew time for heartbeats to be delivered. The heartbeats serve to partition the logs into *epochs*. By making sure that the time between heartbeats is always larger than the maximum delay due to a core's reorder and store buffers, memory latency, and skew in heartbeat delivery, we ensure that *non-adjacent* epochs (i.e., epochs that do not share a heartbeat boundary) have strict happens-before relationships. On the other hand, we will consider instructions in *adjacent* epochs, i.e., epochs that share a heartbeat boundary, to be *potentially concurrent* when they are not in the same thread. Figure 6 depicts blocks of instructions for three epochs (rows) across seven threads (columns), summarizing the potential concurrency with instructions by thread t in epoch l (labeled the *Body* of the "butterfly"). (In reality, epoch boundaries will be staggered and blocks will be of somewhat varying sizes.)

Two-pass Lifeguard Analysis. Lifeguard analysis proceeds using a sliding window of three epochs. Because the analysis considers only three epochs at a time, we can introduce state to summarize earlier epochs. Let *strongly ordered state* be the global metadata state resulting from events that are known to have already occurred, i.e., state resulting from instructions *executed at least two epochs prior.* To perform lifeguard analysis on an epoch l, we use a two pass approach. In a first pass over the log entries in each block of epoch l, each lifeguard thread performs our dataflow analysis using locally available state (i.e., ignoring the wings), producing a summary of lifeguard-relevant events. Next, the lifeguard threads compute the *meet* of all the summaries produced in the wings. In a second pass over the blocks, we repeat our dataflow analysis, this time incorporating state from the wings, and performing necessary checks as specified by the lifeguard writer. Finally, the lifeguard threads agree on a summarization of the entire epoch's activity, and update the strongly ordered state accordingly.

The lifeguard writer specifies the events the dataflow analysis will

track, the meet operation, the metadata format, and the checking algorithm. Tolerating uncertainty potentially introduces imprecision into our dataflow analysis. We prove that our approach does not miss any errors, and sacrifices precision only due to the lack of a relative ordering among recent events.

Further details on butterfly analysis, including its formalization and performance, can be found in Goodstein *et al.* [16].

6. CONCLUSION

Log-Based Architectures (LBA) is a general-purpose monitoring framework that exploits multicore processors to help detect and fix software bugs. This paper discussed our solutions for accelerating dynamic program monitoring tools (a.k.a. lifeguards), and for supporting parallel applications. Overall, LBA reduces the slowdown of dynamic monitoring from 30X-100X to just 2%–51% for sequential benchmark programs and 28%–51% for parallel benchmark programs.

This paper mainly focuses on monitoring user-level applications. We are currently investigating how to protect operating system kernels against device driver crashes and rootkit exploits. Several unique challenges arise in kernel monitoring: for example, how to effectively manage log generation and processing for potentially a large number of threads that execute in different contexts, such as system calls or interrupts? How to contain bugs in the kernel? How to improve and adapt existing user-level lifeguards for monitoring kernels? We are actively working on addressing these challenges.

Acknowledgement. This work was supported in part by a National Science Foundation grant. The authors thank Babak Falsafi, Michelle Goodstein, Olatunji Ruwase and Evangelos Vlachos for their collaboration on the LBA project. We also thank Anastassia Ailamaki, Limor Fix, Greg Ganger, Bin Lin, Vijaya Ramachandran, Michael Ryan, Steve Schlosser, Vivek Seshadri, Theodore Strigkos and Radu Teodorescu for their contributions to LBA.

7. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

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

[3] M. Burtscher. VPC3: A fast and effective trace-compression algorithm. In *SIGMETRICS/PERFORMANCE*, 2004.

[4] 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.

[5] 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.

[6] 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), 2009. *Top Picks from the 2008 Computer Architecture Conferences*.

[7] 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.

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

[13] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2003.

[14] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An extensibility system for commodity operating systems. In *USENIX ATC*, 1998.

[15] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *USENIX Security*, 1996.

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

[17] J. Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. In *MICRO*, 2005.

[18] 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.

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

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

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

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

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

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

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

[26] N. Provos. Improving host security with system call policies. In *USENIX Security*, 2003.

[27] 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.

[28] O. Ruwase, S. Chen, P. B. Gibbons, and T. C. Mowry. Decoupled lifeguards: Enabling path optimizations for dynamic correctness checking tools. In *PLDI*, 2010.

[29] 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.

[30] 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.

[31] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *WBIA*, 2009.

[32] 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.

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

[34] R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3), 1981.

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

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

[37] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry. ParaLog: Enabling and accelerating online parallel monitoring of multithreaded applications. In *ASPLOS*, 2010.

[38] M. Xu, R. Bodik, and M. D. Hill. A 'flight data recorder' for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

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

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

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