Logs and Lifeguards: Accelerating Dynamic Program Monitoring

S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, S. W. Schlosser

IRP-TR-06-05

Research at Intel

Logs and Lifeguards: Accelerating Dynamic Program Monitoring

Shimin Chen[1], Babak Falsafi[2], Phillip B. Gibbons[1], Michael Kozuch[1], Todd C. Mowry[1,2], Radu Teodorescu[1,3], Anastassia Ailamaki[2], Limor Fix[1], Gregory R. Ganger[2], Steven W. Schlosser[1]

[1]Intel Research Pittsburgh [2]Carnegie Mellon University [3]UIUC

Abstract

Runtime monitoring tools are invaluable for detecting various types of bugs, in both sequential and multi-threaded programs. However, running one of these tools slows down the monitored program by an order of magnitude or more, thereby limiting the tool's usefulness. Fortunately, the emergence of chip multiprocessors as a dominant computing platform means that resources are available on-chip to assist in monitoring tasks. This paper presents an architecture for exploiting such resources in order to dramatically reduce the overheads for runtime program monitoring. Specifically, we propose adding hardware support for logging a main program's trace and delivering it to another (otherwise idle) processor for inspection. A *lifeguard* program running on this other processor executes the desired monitoring task. Simulation results using three diverse lifeguards (for address checking, for security exploit detection, and for data race detection) demonstrate that our design typically accelerate monitoring tasks by an order of magnitude. Note that such improvements would not be possible if lifeguards were to execute a standard emulation loop of repeatedly fetching a log entry and branching to the appropriate handler. Instead, our design provides a hardware mechanism for lifeguard programs to jump directly to the next appropriate handler and, for common cases, to deliver handler arguments directly to registers. Our design also includes a prediction-based compression scheme that reduces the log-related bandwidth and storage requirements by an order of magnitude, to less than one byte per instruction.

1 Introduction

As the continued scaling of semiconductor technologies has led to phenomenal increases in computational performance over the past few decades, the corresponding increases in both software and hardware complexity have raised concerns that applications and systems are becoming increasingly error-prone. While it has always been difficult to write bug-free code, recent data suggests that bug rates are getting worse over time as software complexity increases [8]. As we move into the era of chip multiprocessors (CMPs), the inherent challenges in writing multithreaded applications may increase bug rates further. In a networked world, even

obscure bugs that cause no harm under normal conditions can leave a system vulnerable to security attacks. Finally, there is also a concern that hardware fault rates may increase significantly in the future [2].

The good news is that there are tools—which we call *lifeguards*—that dynamically monitor an application to diagnose and sometimes even fix problems [24, 21, 17, 11]. These tools are complementary to static program checking tools [7, 12, 10], and have the advantage of observing the actual dynamic state (e.g., program inputs, memory aliasing, etc.). The bad news is that lifeguards are slow: often 25-100 times slower than the original application [21]. Hence they are currently used only during code development; our goal is to reduce their overhead to the point where they can be run continuously on deployed code.

Why Lifeguards are So Slow. There are three key sources of performance overhead in conventional lifeguard implementations. First, because the lifeguard and the monitored program run on the same core, they compete for cycles. Second, they also compete for resources such as registers and space in the cache hierarchy. Finally, the software must recreate hardware state (instruction pointers, effective addresses, etc.) in the monitored program.

Our Approach: Using Logs to Drive Lifeguards. To accelerate lifeguard execution, we will run them on other available cores on a chip multiprocessor. (Scaling projections indicate the ability to incoporate tens of processors into a single chip by 2015 [2].) To enable the lifeguard to efficiently monitor its corresponding application, we propose a hardware-supported logging mechanism that enables the lifeguard to specify what information it needs, and for that information to be shipped to the lifeguard via a *log*. Hence, this paper introduces a novel class of lifeguard support, *Log-Based Architectures* (LBA), that promises to significantly improve the performance of these applications.

Related Work. Both the Flight Data Recorder [32, 33] and BugNet [19] propose forms of logging for the sake of enabling off-line reconstruction of events leading up to a program crash. In contrast, since our goal is to support efficient on-line program monitoring through logging, our log must be delivered in a ready-to-consume form (as opposed to a form that requires significant off-line reconstruction); hence our design is quite different from these earlier works. The work that is closest to ours in terms of its motivation is iWatcher [34], which invokes monitoring code in response to accesses to certain ranges of memory addresses. The iWatcher paper demonstrates significant performance gains for a single lifeguard (ADDRCHECK) relative to its implementation in Valgrind [21]. (These performance gains are comparable to the ones that we report in this paper for ADDRCHECK once you account for sampling rates.) In contrast

to iWatcher, our log-based approach is more general because it enables the monitoring of *any* instructions of interest, including those that do not reference memory. For example, the TAINTCHECK and LOCKSET lifeguards that we explore in this paper cannot be supported by iWatcher, since they require tracking data flow through registers (not just memory accesses).

Contributions. In this paper, we propose a log-based approach to accelerating dynamic program monitoring, and we evaluate this approach using three diverse lifeguards. Our results demonstrate a substantial (roughly an order of magnitude) performance gain over a commonly-used open-source lifeguard infrastructure. We also evaluate compression techniques for minimizing the bandwidth and storage requirements of the log, and observe that this compression appears to work well enough to make logging feasible.

2 Analysis of Existing Lifeguard Overheads

We begin our investigation by studying three typical lifeguards, a software-only approach to lifeguard implementation, and the performance of that approach.

2.1 Example Lifeguards: ADDRCHECK, LOCKSET, and TAINTCHECK

We begin by describing three diverse lifeguards that are the focus of our study.

ADDRCHECK [20] is primarily a fine-grained memory address checker, ensuring that every load or store is to an allocated region of memory. By intercepting memory allocation routines such as malloc() and free(), ADDRCHECK maintains a logical bitmap with one bit for each byte of the target program's address space that indicates whether or not that byte is currently accessible. The bitmap is consulted on load or store operations, and an error is raised if the corresponding bit(s) are not set. ADDRCHECK uses this mechanism to also catch accesses to inappropriate regions of the stack. The tool also checks for a number of other common memory-related errors such as double free()'s and memory leaks.

LOCKSET [24] detects possible data races in multithreaded programs by noting references to shared memory addresses for which no consistent locking policy exists. During execution, the lifeguard maintains a data structure for each shared memory address, m, that indicates the set of locks, S_m, which have consistently protected accesses to m since its allocation. The lifeguard also maintains a similar structure, S_t, for each thread, t, indicating the set of locks held by t. During execution, when t references m, S_m is updated such that $S_m := S_m \cap S_t$. In this way, S_m always reflects the set of locks that have consistently protected m.

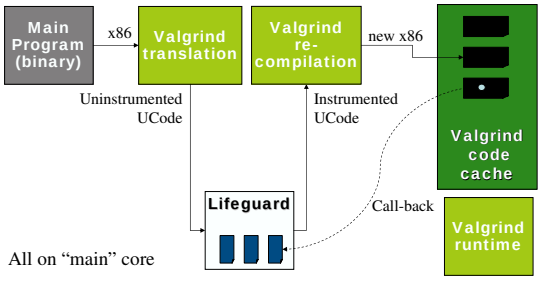


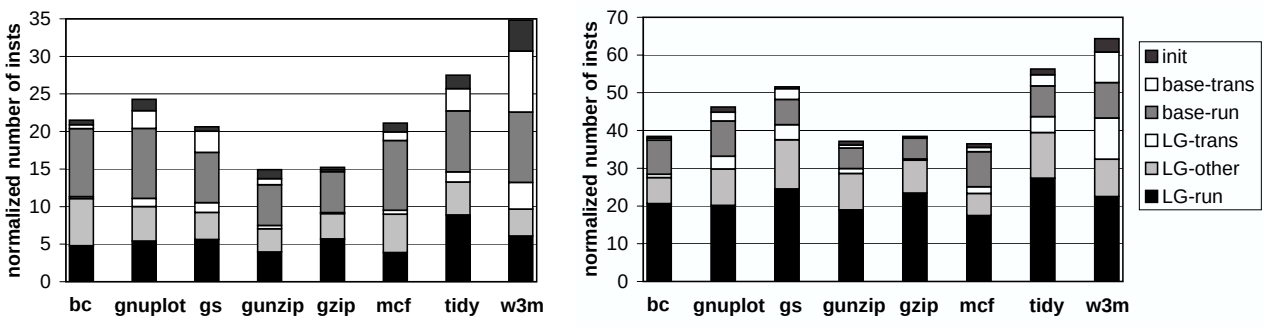
Figure 1: High-level components of the Valgrind engine.

If S_m ever becomes empty, the algorithm has detected that no consistent locking policy protects accesses to m, and an error is raised (additional details can be found in [24]).

TAINTCHECK [22] was designed to catch overwrite-related security exploits such as those due to buffer overruns and format string vulnerabilities. During runtime, all unverified input data, such as data from the network, is treated as suspect, or *tainted*. TAINTCHECK tracks taint values by maintaining a bitmap (one taint bit per byte of the original address space) similar to the accessible bitmap of ADDRCHECK. However, a significant difference between the two lifeguards is that TAINTCHECK must track the *propagation* of taint through the data structures of the program. That is, an access to tainted memory is not an error in itself. Rather, an error is only raised as the result of an inappropriate use of tainted data such as in jump target addresses, format strings, or system call arguments. Consequently, TAINTCHECK tracks the propagation of taint values: if a tainted value is loaded from memory, the destination register is marked as tainted, if the tainted register value is combined with another value (e.g., through an add instruction), the resulting register value is marked as tainted, and if the resulting register is stored to memory, that location is marked as tainted. Thus, TAINTCHECK must consult and update the taint bitmap on most instructions.

2.2 Valgrind: A Contemporary Lifeguard Infrastructure

Our case study, Valgrind [20], is a popular, open source tool for dynamic binary analysis and instrumentation, designed for the x86/Linux platform. Individual lifeguards are written as C language plug-ins that are executed in the context of a Valgrind core engine. Lifeguards export a small collection of functions that the core engine invokes, the primary one being a function that adds analysis code to a given basic block of monitored program code. Figure 1 depicts the high-level components of the Valgrind engine. An executing main program binary is translated on-the-fly into uninstrumented RISC-like "UCode", one basic block at a time. The UCode is then passed to the lifeguard function, which instruments the UCode by adding



(a) ADDRCHECK (b) TAINTCHECK

Figure 2: Breakdown of Valgrind overheads.

lifeguard-specific analysis code. Valgrind then recompiles the instrumented UCode into new x86 code. This code is cached in the Valgrind code cache, in order to avoid this translation/instrumentation/recompilation process when returning to a previously seen basic block. A Valgrind runtime orchestrates and supports the entire execution, providing a variety of functionality such as saving and restoring main program state.

Typically, the analysis code a lifeguard inserts includes both inlined functionality (for relatively simple analysis tasks) and invocations to lifeguard-specific call-back functions (for more complicated tasks). Valgrind also supports a variety of generic call-backs, e.g., for memory events such as malloc() and mmap(). Lifeguards provide the (C or assembly) code for call-back functions, as needed.

2.3 Breakdown of Valgrind Overheads

We found that the Valgrind-based implementations of ADDRCHECK, LOCKSET, and TAINTCHECK , slowed down these test programs' execution by a factor of 9.2–38.5, 56.8–85.5, and 13.8–84.8, respectively, when applied to our set of benchmark applications (described in Table 2, Section 4.1). This is consistent with the slowdowns reported in [20, 22].

To further understand the sources of these overheads, we studied the normalized instruction counts for various components of Valgrind, as shown in Figure 2. (We used instruction counts because of the difficulties in charging execution cycles to tightly overlapped components.) The counts are normalized relative to the number of instructions executed by the test programs without Valgrind. Shown are the breakdowns for ADDRCHECK, a relatively light-weight lifeguard, and TAINTCHECK, a relatively heavy-weight lifeguard.[1] The instruction counts are divided into six categories: *init* (for initialization), *base-trans* (for translating x86

[1] We use optimized versions of the valgrind lifeguards here. More details will be in Section 4.

to and from UCode), *base-run* (for running the base Valgrind functionality, excluding any lifeguard-specific code), *LG-trans* (for inserting Lifeguard instrumentation and translating the inserted UCode), *LG-other* (for running the lifeguard-specific code, excluding the next category), and *LG-run* (for performing the actual lifeguard functionality). As will become clear when we describe our LBA-based implementations of these tools, the *LG-run* category is the "real work" that an LBA lifeguard performs; the *LG-other* category is other work Valgrind performs in support of the real work, such as instructions initiating a lifeguard call-back.

As we can see from Figure 2, initialization, translation, and base execution account for a significant instruction overhead, a factor of 6.0–21.6 for ADDRCHECK and TAINTCHECK. w3m has the largest (normalized) initialization and translation counts primarily because its original program has the smallest instruction counts. The total overhead, excluding *LG-run*, is 9.5–28.8X for ADDRCHECK and 15.0–41.9X for TAINTCHECK. Thus removing these overheads will significantly improve the lifeguard performance. In Section 4 we show that LBA removes these overheads from the main core, and moreover, greatly reduces both the *LG-run* instruction count and its execution time slowdown.

2.4 Sources of Overheads in Dynamic Binary Instrumentation

The overheads observed in Valgrind are likely to arise in any tool based on dynamic binary instrumentation that supports a wide variety of lifeguards and executes the lifeguard on the same core as the main program. This class of approaches, which we call *DBI* approaches, includes tools such as Pin [17] and DynamoRio [3].

We identify three primary sources of overheads in DBI approaches:

(1) *The lifeguard and the main program compete for cycles.* Instructions to perform lifeguard analysis and any other associated tasks (such as translation) are executed on the same core as the main program.

(2) *The lifeguard and main program compete for the memory hierarchy.* Because the lifeguard and main program switch between their respective functionality potentially every cycle, significant thrashing can result, reducing the effectiveness of L1 and other local caches. Moreover, the competition for registers requires that they be frequently saved/restored in order to avoid clobbering.

(3) *The DBI software must recreate hardware state.* In order to provide lifeguards with a view of important architectural state, software needs to recreate the main program's instruction pointer (EIP in x86), effective addresses (EA), etc., and store them in places where, by convention, the lifeguard can find them.

To illustrate how these sources of overhead arise in DBI approaches, Figure 3 presents a simple case

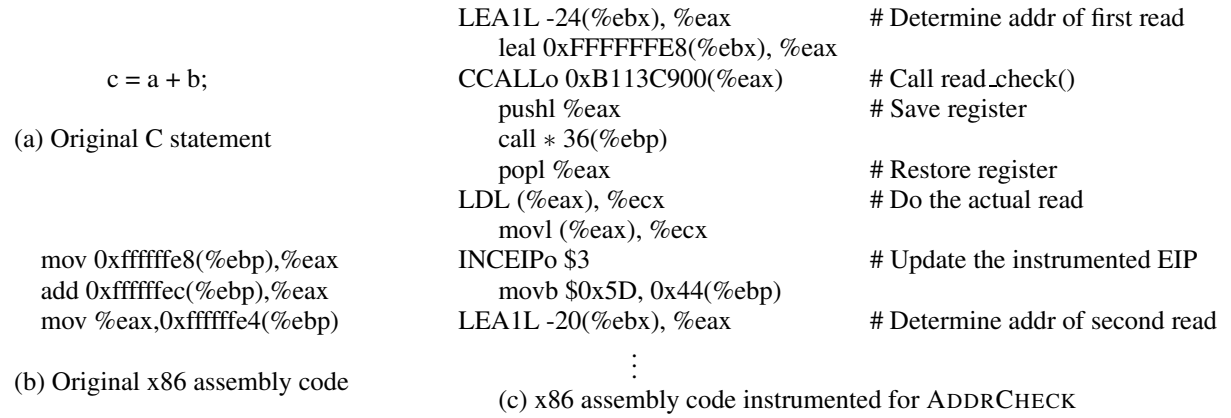


Figure 3: Illustrating the sources of overheads in dynamic binary instrumentation.

study for a Valgrind-based ADDRCHECK processing a single C statement. The C statement in Figure 3(a) becomes 3 x86 assembly instructions in Figure 3(b), which become 27 assembly instructions after instrumentation (the first few of which are shown in Figure 3(c)). These 27 instructions do not include the additional instructions for the call-back functions (e.g., read_check()), and we have also not accounted for any other Valgrind runtime tasks (translation, re-compilation, etc.). Of the 24 extra instructions ($27 - 3$), 14 are used to capture and restore register state, 7 are used to recreate effective addresses and instruction pointers, and the remaining 3 are calls to call-back functions.

3 Architectural and System Support for Log-Driven Lifeguards

To reduce the slowdown associated with dynamic program monitoring, we propose a combination of hardware and system software support that will capture a log of the monitored application and use it to drive the lifeguards more efficiently. Conceptually, the process of connecting a lifeguard to a monitored application works as follows. First, the lifeguard specifies what application it wishes to monitor and what information it would like to see among the set of available options (e.g., the data addresses for all load and store instructions). The operating system then checks whether the lifeguard has permission to access this information before establishing the connection. If so, then the processor that the application to be monitored is running on is configured to begin logging the appropriate information for the committed instruction stream and passing it (in FIFO order) to the given lifeguard. Using an API with an appropriate level of abstraction, the lifeguard consumes the log to drive the execution of its monitoring task.

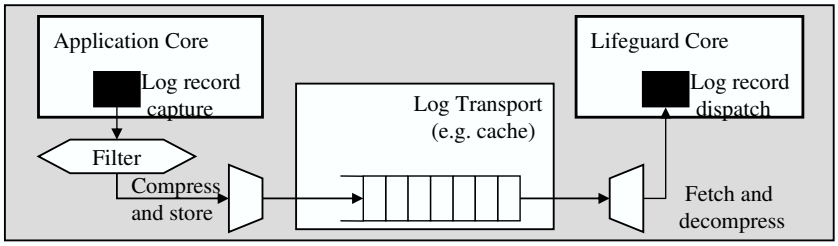


Figure 4: Overview of hardware support for logging.

While software may view the log as a simple queue of information flowing from one processor to another, the actual hardware implementation is more sophisticated, as illustrated in Figure 4. As we see in the figure, the first step is for hardware on the monitored processor to capture the desired logging information, which is then compressed (by hardware) before it is stored in the log buffer and transported to the lifeguard. The compressed log is decompressed by complementary hardware on the lifeguard's processor and used to efficiently dispatch the appropriate lifeguard code in response to each log entry. We describe this process in more detail in the following subsections.

3.1 Capturing the Log

In general, a log may contain all state changes that are either architectural or microarchitectural due to instructions. Without loss of generality, in this paper we focus on architectural state changes. This information includes the program counter, the instruction opcode, the operand specifiers and the operand values, and architectural state not directly visible to the program such as condition codes for all instructions that retire from the pipeline.

Fortunately, much of the information to be captured in the log is already maintained in modern microprocessor pipelines. For instance, out-of-order microprocessor cores that use a centralized mechanism for scheduling, renaming and reordering instructions such as a register update unit (RUU) [28] (e.g., the Intel Core microarchitecture [16] derived from Pentium Pro [27]) maintain all the necessary information for a log entry per instruction. [2]

[2] In general, the information corresponding to a program instruction may span multiple entries in the RUU (e.g., due to breaking instructions into uOps in Intel-based implementations). Nevertheless, the RUU maintains information regarding which uOps belong to a program instruction to allow for precise interrupts. The log can capture this information to allow for identifying program instruction boundaries across log entries.

In general, the information needed for the log may not be available in one place in the datapath. For instance, in scalar pipelines the ALU operand values are available immediately prior to execution and data memory address and value is available prior to a data cache access. As such, the pipeline latches must be extended to record and transfer the operand values to be recorded in the log. Similarly, in out-of-order pipelines with a decoupled reorder buffer, issue and load/store queues and a renamed register file, (e.g., Pentium 4) the reorder buffer must be extended to maintain the operand values for instructions (i.e. the reorder buffer will be organized much like an RUU to allow for log capture).

3.2 Compressing and Decompressing the Log

To reduce the log storage and communication bandwidth requirements, we propose using on-the-fly compression as the log is produced. The compression subsystem design must balance compression rate, speed, and hardware cost. We considered two main compressor classes. The first class, general purpose compressors such as *gzip, bzip2*, have been shown [4, 5] to be less time/space efficient when compressing program traces than the second class, special-purpose trace compressors. In our design, we build on a significant body of research in the field of trace compression [6, 15, 18]. One particularly interesting design point in this space, shown to yield good results, is value prediction-based trace compression [4].

3.2.1 Value Prediction-Based Trace Compression

As the name implies, this technique uses a set of predictors trained on trace data to forecast likely next values in the trace. When compressing a trace, the current value is compared to the predicted values. If at least one of the predictions is correct, an identifier for the correct predictor is recorded in the compressed trace rather than the predicted value. If none of the predictions is correct, a mispredict identifier is recorded, followed by the mispredicted value. For decompression, a set of identical predictors is maintained and updated on the decompresser side.

We study three predictor types that produce good prediction rates in the context of our log data for reasonable hardware cost. The simplest one is Last Value predictor (LV[n]) [6], which stores the most recent n values in a FIFO. All n values are provided when a prediction is requested, LV[n] essentially acting like n independent predictors. LV[n] works best with alternating values or repeating sequences of no more than n values. We use it for predicting memory addresses.

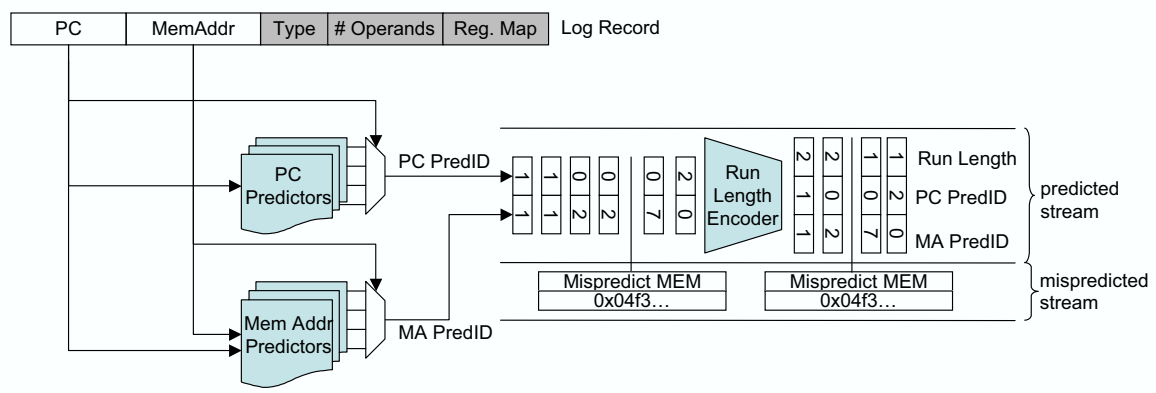


Figure 5: Prediction-based log compression engine. Two sets of predictors are used to predict PC and memory address values. The predicted stream is further compressed with RLE.

The Finite Context Method predictor (FCM$x[n]$) [25] computes a hash of the x most recently encountered values which is then stored in a first level table. The hash is used to index the predictor's second level table which works just like the LV[n] predictor. The FCM predicts the last n values that were observed to follow a context of x values. This predictor works best with sequences of repeating values that depend on some context, making it suitable for predicting program counter values.

Finally, the Differential Finite Context Method (DFCM$x[n]$) [13] predictor is similar to FCM, except it predicts (and stores) differences (strides) between consecutive values. To form the final prediction, the predicted stride is added to the last value seen. DFCM has the advantage that it can predict sequences of offsets that repeat. We used it for predicting memory addresses in conjunction with the LV predictor.

3.2.2 The Compression/Decompression Mechanism

The records in the log can contain a number of different fields (program counter (PC), instruction type, number of operands, register ids, memory addresses). For now, we assume there is little or no application data (e.g. register values) being sent through the log. We distinguish two types of fields based on whether the information they contain can change with respect to a given PC. The instruction type, the number of operands, the register ids are always static with respect to the program counter. The memory addresses are generally dynamic with respect to the PC. Figure 5 shows the layout of the compression engine.

For compression we use two sets of predictors, one for PC values and one for memory addresses. Each predictor set generates a stream of predictor ids. The static information associated with the PC is cached at the receiver side, in a table that is associated with the PC predictor table. The send and receive tables

naturally remain coherent such that when a PC is correctly predicted, the associated static information can be extracted from the corresponding entry in its table. Therefore the static fields only need to be sent when the PC is mispredicted.

Further compression is obtained by performing run-length encoding (RLE) on the prediction stream. In order to simplify decoding at the decompresser side, the PC and memory address streams are combined into a single stream (prediction stream in Figure 5) which is then compressed using RLE. This means that a single Run Length entry is sufficient for both the PC and memory address streams.

3.3 Buffering and Transporting the Log

Once the log information has been compressed, it is ready to be shipped to any consuming lifeguards within the CMP. Because the log is a continuous stream of information flowing from one processor (or hardware thread) to another, many of the techniques for optimizing stream-style computation are relevant for logging [1, 9, 14, 26]. For example, if the CMP already contains an interconnect that is optimized for streaming and structures such as a streaming register file [1], these can be used to transport the log.

Alternatively, if the CMP is not already optimized for on-chip streaming, we can also buffer and communicate the log through the on-chip cache hierarchy. The idea is for the system to allocate a circular buffer in memory that will fit within an appropriately small fraction of the largest physically-shared on-chip cache (the L2 in our experiments). To help manage the flow of log data within the on-chip cache hierarchy, the system can use memory references with explicit replacement hints [14] to avoid having this circular log buffer either pollute the L1 caches or else become systematically displaced from the largest on-chip cache.[3]

A natural concern with transporting the log is whether it consumes too much bandwidth or on-chip storage. Fortunately, using the compression techniques described earlier in Section 3.2 (and evaluated later in Section 4.3), the log will generate roughly 0.8 bytes per instruction in the monitored program, which is roughly half of the bandwidth generated by L1-to-L2 data cache misses for the 16KB data caches used in our experiments. Hence it appears to be feasible to transport this volume of data within the chip. (The fact that the log is not transported off-chip is very helpful.) In terms of storage capacity, if we can allocate 128KB of the largest on-chip cache to the circular log buffer, this will capture roughly 160,000 instructions worth of detailed logging information.

[3] If the latter issue becomes problematic and cannot be resolved through replacements hints, another possibility might be for the OS to explicitly pin the log buffer into the cache, if the hardware supports this.

3.4 Consuming the Log

Upon transport and decompression of the log, entries are placed into a log dispatch buffer, and would be consumed by the lifeguard in the order generated. In the simplest form, the dispatch buffer would be memory-mapped into the lifeguard's address space with an interface both to read and to advance log entries. The lifeguard would run in a driver loop (much as an Active Message [30] communication subsystem would) reading one entry at a time, looking up a handler table to dispatch code for the corresponding log entry type, executing the handler, and moving on to the next log entry.

Unfortunately, our lifeguards can be arbitrarily lightweight and therefore such a log dispatching approach would prohibitively incur high overheads. Because a lifeguard may perform a few ALU operations per instruction in the monitored program (e.g., when computing register values taints), handler dispatch overhead (i.e. the overhead of executing the dispatch loop) can directly impact overall performance.

Ideally, the log entries would magically turn into a stream of handler instructions that would flow into the pipeline. Unfortunately, a system that provides such a stream directly would require replacing the front-end of the core to eliminate the instruction fetch and sequencing logic.

Instead, we propose two hardware dispatch optimizations that dramatically reduce dispatch overhead and accelerate lifeguard execution with moderate modifications to the core. We make the observation that the code within the handler can be fetched and sequenced with the existing front-end mechanisms (i.e. the branch and fetch units and the instruction cache) and the only modification necessary is when control is transferred from one handler to another corresponding to subsequent log entries.

The first optimization is a the introduction of a `nlba` instruction which lifeguard software uses to signal the return from log record handlers. The core also has access to a table, indexed by log record type, of handler entry points, so that upon execution of `nlba`, the core transfers control directly to the handler corresponding to the type of the next log record. As such, the `nlba` instruction eliminates the software decode and control transfer overhead of a driver loop, reduces the control transfer to a single instruction, and provides a mechanism through which handler dispatch operations may be perfectly predicted assuming a log queue of more than record. The second optimization reduces the overhead of marshaling the log entry data by allowing a direct transfer of log data into pipeline registers.

Figure 6 depicts an example log record handler from ADDRCHECK in C pseudo-code with its assembly code (a), and the corresponding log dispatch hardware mechanisms we propose. The example describes a

```
void adrchk_ld4B (unsigned int addr) {
  ABits* abits = ABits_array[addr>>16];
       // movl    %eax, %edx
       // shrl    $16, %eax
       // movl    AccessBits_array(,%eax,4), %ecx
  UINT32 off  = (addr & 0xffff) >> 3;
       // movzwl  %dx,%eax
       // shrl    $3, %eax
  if (abits->map[off] == 0)  next_lba_record();
       // cmpb    $0, (%eax,%ecx)
       // jne     fast_path_failed
       // nlba    %eax
  slow_path_load_4B(addr);
       //fast_path_failed:
       // movl    %edx, (%esp)
       // call    slow_path_load_4B
       // nlba    %eax
}
```

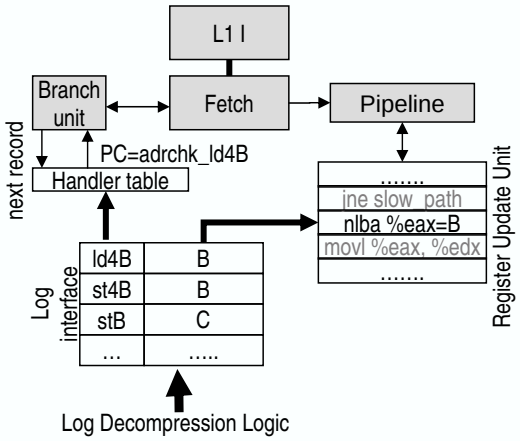


Figure 6: (a) Example log record handler and (b) Dispatch hardware

handler for checking whether a load to a four-byte address is legitimate. The lifeguard maintains a bit per byte of memory, and a bit value of 0 indicates addressability. In the common case, the data is addressable and the bit values for the four-byte item and all its nearby neighbors are 0. So the handler first checks to see if an entire 8-bit mask corresponding to a 8-byte region containing the data is 0. If so, no other checks are necessary. Otherwise, the handler must peruse the bitmap more carefully, possibly raising an error condition, by calling into extra handler code (i.e. at "`fast_path_failed`").

The assembly code (Figure 6(a)) corresponding to the handler, assumes that the input address `addr` is directly extracted from the log entry and placed in `%eax`. The code indicates that, in this example, besides the raw lifeguard code to check for addressability, all the log dispatch and processing overhead is reduced to a single `nlba` instruction.

Figure 6(b) depicts the anatomy and functionality of the proposed hardware optimizations for a Intel's Core architecture [16]. The `nlba` instructions are placed in the RUU and behave like control transfer instructions when executed. These instructions additionally advance the dispatch buffer to allow for subsequent log entries to be dispatched.

Because the handler instructions corresponding to a log entry are control-dependent on prior program-order conditional branches, they may be squashed and restarted, requiring the log entry to remain in the dispatch buffer until the `nlba` instruction corresponding to a subsequent log entry is retired. Moreover, because multiple handlers can be in flight simultaneously requiring access to the corresponding log entries from within the pipeline, each RUU entry keeps a back pointer to the corresponding log entry so that multiple

entries within the dispatch buffer.

Much like other control transfer instructions, `nlba` instructions are placed in the BTB in the branch unit. Upon lookup in the BTB corresponding to an `nlba` instruction, however, rather than predict a target address, the branch unit fetches the log entry type from the head of the log, looks up the handler table for the starting address of the corresponding handler, and uses this address as the predicted target address to fetch instructions from. The handler table is a small (e.g., about 256 entries) SRAM-based RAM structure that is initialized upon starting the lifeguard code, and maintains the starting address for the frequent handlers (e.g., memory instruction for address checking). Log entries corresponding to the infrequent handlers (e.g., system calls) invoke a special "slow path" handler in which the lifeguard software decodes the log record event type further to determine the appropriate handling for the record.

To allow for filling a handler parameter value directly in hardware upon dispatch, the `nlba` instruction also encodes a destination register corresponding to the architectural register in which a parameters may be passed—i.e. `%eax`. Because an `nlba` instruction occupies an entry in the RUU, it decodes and renames like other instructions. Moreover, the rename unit fetches the operand value for `%eax` from the corresponding log entry, therefore the execution of the `nlba` instruction can be folded (i.e. its result is available upon placement in the RUU). Moreover, subsequent reads to `%eax` read from the RUU entry corresponding to the preceding `nlba` instruction.

4 Evaluation

In this section, we evaluate the effectiveness of the LBA approach to running lifeguards. Section 4.1 describes our experimental methodology, lifeguard implementation details, and the benchmarks used in our study. Then, Section 4.2 presents the experimental results for the three example lifeguards described previously in Section 2.1. Finally, Section 4.3 evaluates prediction and compression techniques to reduce the bandwidth requirement for transporting the log.

4.1 Experimental Setup

Methodology. The performance of log-driven lifeguards is influenced by three main components: (i) the rate of monitored events (e.g. ADDRCHECK mainly checks load and store operations, while TAINTCHECK also monitors register-only operations); (ii) the average code path length of lifeguard event handlers (measured in

Table 1: Simulation platform.

Simulator description	
Simulator	Virtutech Simics 2.2.14
Target OS	Fedora Core 2
Processor core	In-order scalar
Cache simulation	g-cache module in Simics

Memory parameters	
Private L1I	16KB, 64B line, 2-way assoc, 1 cycle
Private L1D	16KB, 64B line, 2-way assoc, 1 cycle
Shared L2	512KB, 64B line, 8-way assoc, 10 cycles
Main Memory	200-cycle latency

lifeguard instructions per application instruction); and (iii) CPI, which may vary significantly with lifeguard. For example, the working set of LOCKSET may be substantially larger than that of ADDRCHECK.

To evaluate the benefits of introducing logging support, we measure the performance of several benchmark applications running with LBA-supported lifeguards compared to the performance of the same applications running without lifeguards and running with lifeguards supported by a software-only approach such as Valgrind. We use the the Simics [29] full-system simulation platform, which ran a standard Fedora Core 2 operating system in our experiments. Both the host and simulated platforms are x86-based. Because we were primarily concerned with the above three major components of performance, we modeled an single-CPI in-order core. The cache parameters are shown in Table 1. We report instruction counts, cache miss statistics, and execution time from Simics output.

We developed a Simics extension module that implements the LBA support (such as `nlba`) by recognizing special instructions (`prefetchnta` in our implementation) issued by the lifeguards. The lifeguards run as standard Linux processes in the hosted Fedora Core 2 environment. To ensure reproducibility, the log records are fed from an execution trace of the application under test obtained by applying a Pin [17] based trace collection tool. The traces include the instruction addresses, register information, memory references, and other information such as system calls that would be included in the log on a machine supporting LBA. As the lifeguard executes, we inject memory references from the application's trace into the simulated shared L2 cache to model cache interference between the application and lifeguard.

The same simulation platform and parameters were also used to measure performance for the normal executions of benchmark applications and running the applications with Valgrind lifeguards.

To quantify the bandwidth requirements of transporting the log, we study various online prediction and compression techniques on the application execution traces in Section 4.3.

Lifeguard Implementation Details. We measured the event frequency in the application execution traces and designed the log record handler interface to provide more optimizations to frequent events (such as memory references, and 4B register accesses) while using generic interface for infrequent events (such as

OS system calls, memory allocation calls, and pthread calls). For example, memory references are further distinguished as 1/2/4/8-byte aligned loads/stores, and unaligned accesses. In this way, handler code for a specific type of memory reference does not need to check the size of a memory reference or whether it is aligned, thus reducing the code path length. In the same spirit, frequently needed information (such as memory addresses) for a particular event is directly put into registers before dispatching the event handler.

We ported three lifeguards, namely, ADDRCHECK, TAINTCHECK, and LOCKSET, from Valgrind 2.2.0 to the LBA system. We optimized the handler code for frequent events. A typical optimization is to test for a filter condition, and only perform more detailed checking if the filter test fails. As illustrated previously in Figure 6(a) in Section 3.4, for ADDRCHECK, we observe that most memory references are to the middle of allocated buffers, and therefore we use a fast 8-byte test as a filter for smaller-sized memory references.

LOCKSET maintains a 32-bit state for every 4-byte memory word in the application program. It encodes the sharing state (exclusive, read shared, or read/write shared) and the lock set pointer of the memory location. For each memory reference, LOCKSET performs an expensive lock set intersection operation. However, we observe that the lock set of a memory location typically converges quickly to one of the three stable states: (i) exclusively owned by one thread; (ii) read-only by all threads without holding locks; or (iii) read-write protected by a stable lock set. Moreover, the same stable lock set often protects multiple memory references. Therefore, we maintain two global state templates for the exclusive state and read-only no lock, and a per-thread state template to remember the most recently seen read-write stable lock set. Then, for a memory reference, its state information is compared with the three templates as a filtering step.

The most frequent operation of TAINTCHECK is to propogate taints. Since 4B memory references are the most frequent, we use 2 bits to store the taint for every memory byte in the application program. Thus the taint of 4-byte word can be retrieved or updated through one-byte load or store instructions.

To make fair comparisons, we applied the above optimizations to the Valgrind lifeguards. We report results for the original Valgrind lifeguards, optimized Valgrind lifeguards, and the LBA-supported lifeguards.

Benchmarks. The benchmark applications selected for this study are listed in Table 2. The first eight benchmarks listed are single-threaded applications included in the BugNet [19] study. The remainder are a multi-threaded benchmark from the SPLASH-2 [31] benchmark suite and a parallel SAT solver [23]. Table 2 also reports the dynamic instruction counts for the benchmark applications. Note that there is roughly one memory reference for every two instructions in this set of benchmarks.

Table 2: Summary of benchmarks.

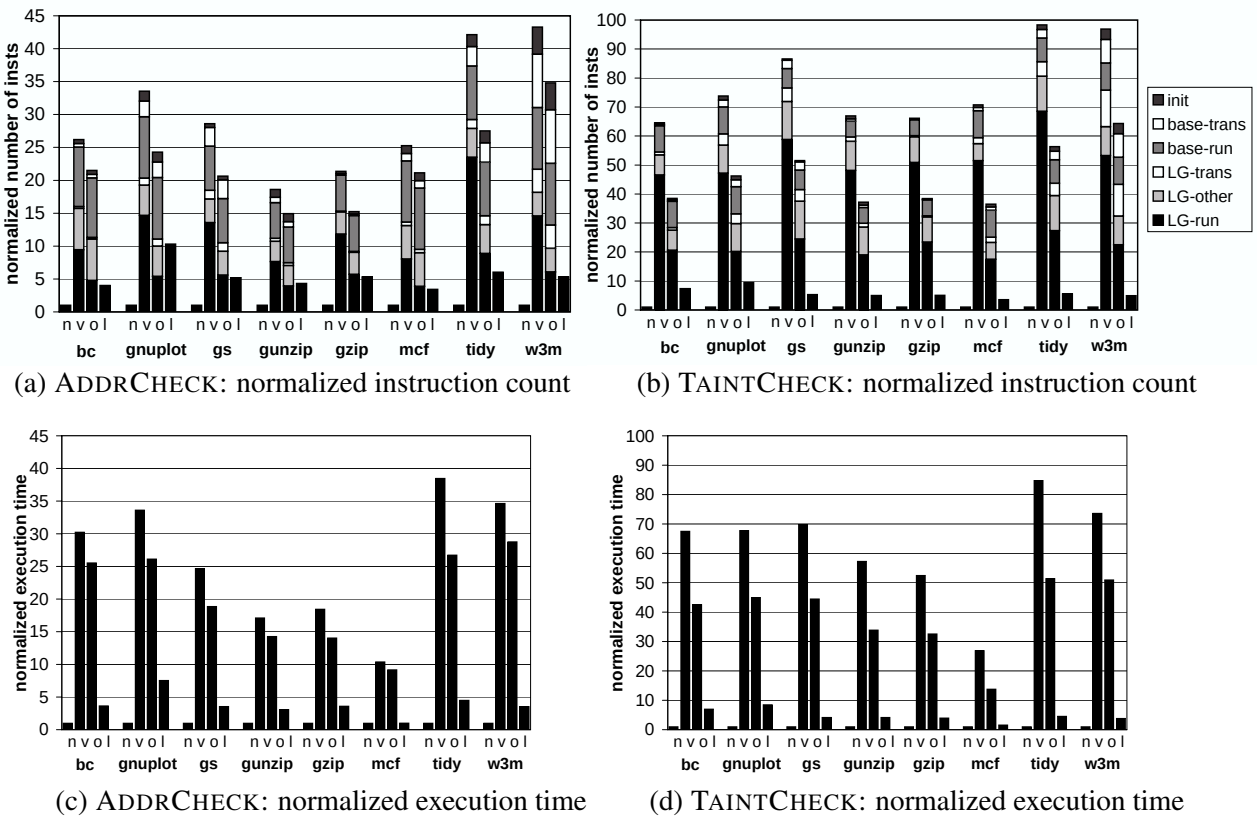
Benchmark	Version	Description	Instr. (10^6)	Loads (10^6)	Stores (10^6)
bc	1.06	Arbitrary precision calculator	262	70	41
ghostscript	8.12	Document specification interpreter	310	113	51
gnuplot	3.7.1	Math function plotter	84	25	19
gunzip	1.2.4	Data decompressor	116	41	13
gzip	1.2.4	Data compressor	430	146	83
mcf	CPU2000-1.3	Transportation problem solver	125	45	11
tidy	050826	HTML syntax checker	97	43	21
w3m	0.3.2.2	HTML-to-text renderer	36	12	6
water-nq	SPLASH-2	Molecular force evaluator	151	33	17
zchaff	2002.7.15	Boolean satisfiability solver	386	131	92

4.2 LBA Performance Results

Figure 7 shows the normalized instruction counts and normalized execution times of ADDRCHECK and TAINTCHECK for the single-threaded benchmarks. We compare the normal executions of the benchmark and three implementations of the lifeguards. From Figure 7, we see that Valgrind lifeguards dramatically increase the instruction counts by a factor of 14.9–43.3 for ADDRCHECK and 36.5–98.3 for TAINTCHECK, thus slowing down the programs by a factor of 9.2–38.5 for ADDRCHECK and 13.8–84.8 for TAINTCHECK. In contrast, log-driven lifeguards increase the instruction counts by a moderate factor of 3.4–10.3 for ADDRCHECK and 3.6–9.5 for TAINTCHECK. The resulting slowdowns are 1.01–7.5X for ADDRCHECK and 1.6-8.5X for TAINTCHECK. (The best slowdown numbers are for mcf and will be explained below.)

As shown in Figures 7(a) and (b), each Valgrind bar is broken down into six categories, as previously described in Section 2.3. Basically, *LG-run* corresponds to the actual lifeguard functionality, while the other five categories are instruction overhead in translating x86 code to and from UCode, inserting instrumentations, and other supports to run the lifeguard-specific code. As we can see from Figures 7(a) and (b), these overheads are 9.5–28.8X for ADDRCHECK and 15.0–41.9X for TAINTCHECK. The larger overhead for TAINTCHECK is mainly because TAINTCHECK does more instrumentation than ADDRCHECK. By removing these overheads, LBA significantly improves lifeguard performance.

We now focus on the *LG-run* components and the LBA lifeguard bars in Figures 7(a) and (b). First, the *LG-run* components of the optimized Valgrind bars are much shorter than those of the original Valgrind bars. This shows the effectiveness of the lifeguard code optimizations. Second, as expected, the LBA ADDRCHECK bars are of similar height to the *LG-run* components of the optimized Valgrind bars, since the same lifeguard functionalities are executed. However, the LBA TAINTCHECK bars are much shorter



(a) ADDRCHECK: normalized instruction count (b) TAINTCHECK: normalized instruction count

(c) ADDRCHECK: normalized execution time (d) TAINTCHECK: normalized execution time

Figure 7: LBA performance results for ADDRCHECK and TAINTCHECK. For every benchmark, we compare the normal execution without lifeguards (**n**) and three implementations of lifeguards: origianl valgrind lifeguard (**v**), optimized valgrind lifeguard (**o**), and LBA lifeguard (**l**). Number of instructions and cycles are normalized to those of normal executions.

than the *LG-run* components. This is because Valgrind introduces a large number of temporary registers for generating RISC-like UCode, and therefore TAINTCHECK has to propogate taints for the temporary registers, thus incuring more overhead. Third, the most frequent code path for checking memory references in ADDRCHECK takes eight instructions. Since there is a memory reference per 1.5–2.3 instructions, the total number of instructions is increased by a factor of roughly 4–5. Comparing the LBA ADDRCHECK and TAINTCHECK, we see that the TAINTCHECK bar is supprisingly similar to ADDRCHECK in spite of the fact that TAINTCHECK is a more heavy-weighted lifeguard. This is mainly due to the following facts. (i) On average 31% of the instructions do not modify registers or memory locations. These are mostly `cmp` instructions, and jump instructions.[4] (ii) Among the instructions that TAINTCHECK handles, an average 61% are memory references. The code path length of TAINTCHECK and ADDRCHECK are similar for these

[4]TAINTCHECK monitors only the infrequent indirect jumps to make sure jump target addresses are not tainted. [22]

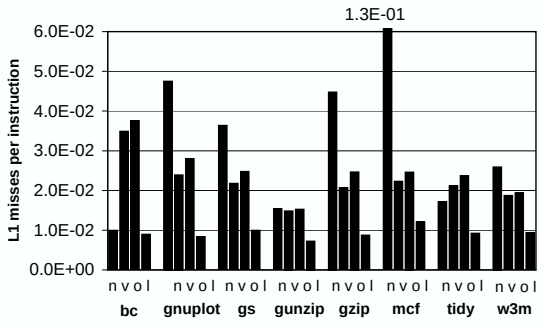


Figure 8: L1 misses per instruction for ADDRCHECK.

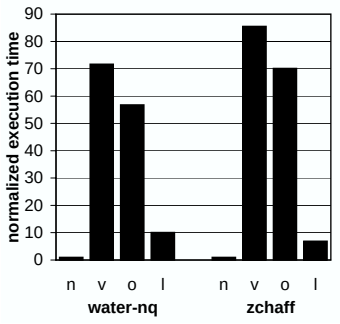


Figure 9: LBA performance results for LOCKSET.

instructions. (iii) The register-only operations can be handled in 2–3 instructions because register taints are kept in an array and can be accessed much more quickly than memory taints.

Figures 7(c) and (d) show the normalized execution times. ADDRCHECK keeps 1 bit and TAINTCHECK keeps 2 bits for every application byte. Therefore, they have a much smaller cache footprint. As shown in Figure 8, the L1 misses per instructions for ADDRCHECK are much lower than both the application and the Valgrind lifeguards. This results in a better CPI and therefore smaller normalized time than normalized instruction counts. The mcf benchmark shows an interesting case, which incurs a large number of cache misses. For mcf, LBA ADDRCHECK is only 1.01X slower than the normal execution. LBA TAINTCHECK sees a similar effect and is only 1.6X slower. Note that this working set benefit is largely lost for Valgrind lifeguards because the lifeguard and the application program are competing for the same L1 cache. Overall, compared to Valgrind, LBA lifeguards achieve 3.5–10.3X speedups for ADDRCHECK, and 5.3–19.2X speedups for TAINTCHECK.

In addition to the ADDRCHECK and TAINTCHECK experiments, we also performed LOCKSET experiments. For the water and zchaff benchmarks, the Valgrind lifeguard slows down the application by a factor of 71.6–85.5 fold, and the optimized Valgrind lifeguard slows down the application by a factor of 56.8–70.2 fold. In contrast, the log-driven lifeguard is 6.9–10.1X slower than the application, thus achieving 5.6–12.4X speedups compared to Valgrind lifeguards.

4.3 Prediction and Compression Results for Reducing Log Bandwidth

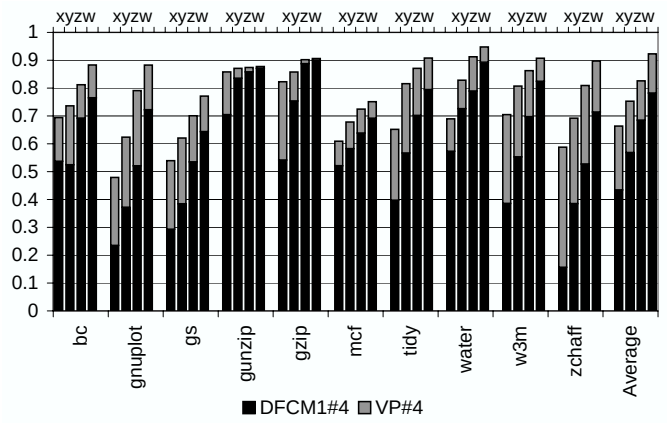
Using the traces described in Section 4.1, we evaluate the performance of our proposed log compression scheme and quantify the bandwidth and storage requirements associated with managing the log.

Table 3: Predictors and configurations tested.

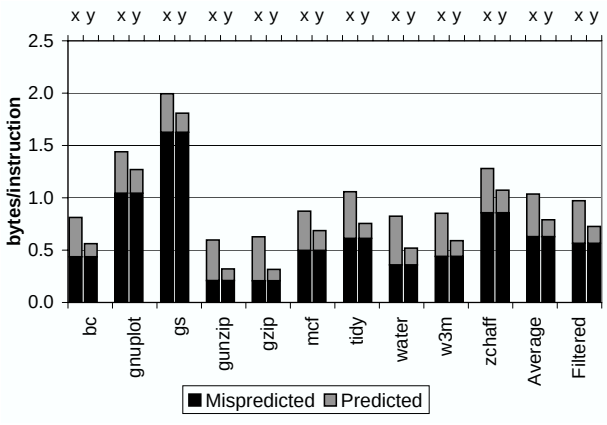
Value Predictor	LV, FCM, DFCM
Table Sizes	2KB, 4KB, 8KB, 16KB
History/Context	1-4 / 1-3

Table 4: Best performing configuration.

Stream	Predictor	History	Context
PC	FCM	4	1
MEM	LV, DFCM	2	1



(a) prediction rate



(b) bandwidth

Figure 10: (a) Memory address prediction accuracy using a hybrid (DFCM/VP) predictor. Bars x,y,z and w correspond to total predictor sizes of 2, 4, 8 and 16 KB. (b) Bandwidth required for the compressed log; x indicates value prediction and y indicates value prediction plus RLE.

To determine a good combination of predictors to use for each of the value streams present in the log, we evaluated the combinations of predictor sizes, history and context lengths shown in Table 3. Using TCgen [5] to generate software implementations of the different predictors, we determined that best performing configurations for our traces are those shown in Table 4. To ensure a fair comparison, we always use the same total size for the group of predictors that we assign to each stream of values, regardless of configuration. Hence, configurations which use longer contexts have fewer entries in the predictor tables.

Figure 10(a) reports the memory address prediction accuracy obtained for several predictor sizes. Our goal was 90% accuracy to provide approximately an order of magnitude compression, and we determined this result to be feasible with a 16KB predictor.

Using the configuration in Table 4, we quantified the bandwidth required for transporting the compressed log. The pairs of bars in Figure 10(b) show the bandwidth required using value prediction (bars marked with x) and value prediction with the optional run length encoding (bars y). All pairs except the last, labeled "Filtered", show the bandwidth for the full log. The last pair, provided for comparison, shows the reduced bandwidth required if the register map information is filtered out before the log is transmitted (as would be acceptable for lifeguards such as ADDRCHECK). The stacked bars illustrate the contribution of mispredicted

vs. predicted values in the compressed log. RLE is only applied on the streams of predictor ids and not the mispredicted data. As a result the contribution of RLE to the overall compression rate is diminished when the mispredicted data dominates in the compressed log.

With RLE, the log grows at less than one byte per instruction. The average bandwidth requirement is 0.82 bytes/instruction for a unfiltered log and 0.75 bytes/instruction for a filtered log that doesn't include register maps. When compared with the uncompressed rates of 17 and 15 bytes/instruction, respectively, we see that compression significantly reduces the resource demands of logging. This order of magnitude improvement also yields a bandwidth that is significantly less that the L2 cache bandwidth available on-chip. The associated cost is 32KB per core for the prediction tables, and an additional 16KB of storage for the static information maintained on the lifeguard side.

5 Conclusions

A logging mechanism that is integrated into the processor and system architecture may provide the missing link that would allow *lifeguards* to recognize correctness problems in complex software systems. Hopefully these mechanisms would greatly improve programmer productivity and software reliability, which will be increasingly important as we move to many-core architectures. Our results demonstrate a roughly order-of-magnitude speedup in lifeguard performance for a set of three diverse lifeguards compared with implementations in Valgrind, reducing the slowdown to a factor of 2-5. Our compression techniques reduce the size of the average log entry to less than a byte per instruction on average. Overall, the log-based approach to dynamic program monitoring appears quite promising.

References

[1] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das. Evaluating the imagine stream architecture. In *ISCA*, June 2004.

[2] S. Borkar. Microarchitecture and design challenges for gigascale integration. Keynote Speech at MICRO 2004, December 2004.

[3] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[4] M. Burtscher. VPC3: A fast and effective trace-compression algorithm. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 167–176, New York, NY, USA, 2004. ACM Press.

[5] M. Burtscher and N. B. Sam. Automatic generation of high-performance trace compressors. In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization*, pages 229–240, Washington, DC, USA, 2005. IEEE Computer Society.

[6] M. Burtscher and B. G. Zorn. Exploring last n value prediction. In *PACT*, October 1999.

[7] 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.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP*, 2001.

[9] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. A., N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *SC'03*, Phoenix, Arizona, November 2003.

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

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

[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] B. Goeman, H. Vandierendonck, and K. de Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *HPCA*, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[14] J. Gummaraju and M. Rosenblum. Stream Programming on General-Purpose Processors. In *MICRO*, Barcelona, Spain, November 2005.

[15] E. Johnson. PDATS II: Improved compression of address traces. In *IEEE International Performance, Computing, and Communications Conference*, 1999.

[16] K. Krewell. Intel looks to core for success. *Microprocessor Report*, March 2006.

[17] 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.

[18] A. Milenkovic and M. Milenkovic. Stream-based trace compression. *IEEE Computer Architecture Letters*, 2(1), 2003.

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

[20] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, Nov. 2004. http://valgrind.org.

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

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

[23] Princeton Zchaff. http://www.princeton.edu/ chaff/zchaff.html.

[24] 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.

[25] Y. Sazeides and J. E. Smith. The predictability of data values. In *MICRO*, 1997.

[26] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. In *LCTES*, June 2005.

[27] J. P. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw Hill, 2002.

[28] G. S. Sohi. Instruction issue logic for high-performance, interruptable, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, 1990.

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

[30] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *ISCA*, May 1992.

[31] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.

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

[33] M. Xu, R. Bodik, and M. D. Hill. Considerably longer race recording using less timestamp memory. In *ASPLOS*, 2006.

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