Log-Based Architectures for General-Purpose Monitoring of Deployed Code

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], Bin Lin[1,4], Steven W. Schlosser[1]

[1]Intel Research Pittsburgh [2]Carnegie Mellon University [3]UIUC [4]Northwestern

Categories and Subject Descriptors: C.4 [Performance of Systems]: Reliability, availability, and serviceability; B.8.1 [Performance and Reliability]: Reliability, Testing, and Fault-Tolerance; C.5.3 [Computer System Implementation]: Microcomputers — *Microprocessors*.
General Terms: Design, Reliability.
Keywords: Log-Based Architectures, general-purpose task monitoring, chip multiprocessors.

1. INTRODUCTION

Runtime monitoring tools are invaluable for detecting various types of bugs, in both sequential and multi-threaded programs. However, these tools often slow down the monitored program by an order of magnitude or more [4], implying that the tools are ill-suited for always-on monitoring of deployed code. Fortunately, the emergence of chip multiprocessors as a dominant computing platform means that resources are available on-chip to assist in monitoring tasks. In this brief note, we advocate *Log-Based Architectures* (LBA) that exploit such on-chip resources in order to dramatically reduce the overhead of runtime program monitoring. Specifically, we propose adding hardware support for logging a main program's trace and delivering it to another (otherwise idle) processing core for inspection. A *lifeguard* program running on this other core executes the desired monitoring task.

In contrast to previous proposals that add special-purpose hardware support for specific types of lifeguards [7, 8] (e.g., checking memory references or function call/return pairs), LBA is a general-purpose infrastructure, aimed to enable efficient monitoring for a wide variety of program bugs, security attacks, and performance problems. (We show three diverse lifeguards in our evaluation section.) We believe that the benefits of LBA will more than warrant the costs of adding the requisite hardware support, especially because the costs are amortized over the diverse set of lifeguards supported.

Software-only approaches (e.g., using dynamic binary instrumentation), while also general purpose, suffer from two key sources of performance overhead. First, because the monitoring task (i.e., the lifeguard) and the monitored program run on the same core, they compete for processor resources such as cycles, registers, and cache space. Second, these software-based approaches frequently expend considerable effort recreating hardware state not exposed through the architecture (instruction pointers, effective addresses, etc.).

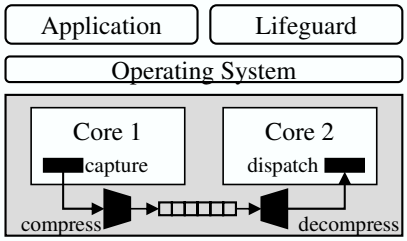


Figure 1: Dual-core LBA system

In contrast, LBA lifeguards run on different cores than the monitored programs, and hence do not compete for cycles, registers or L1 cache. Moreover, the hardware-based logging captures hardware state directly. As a bonus, the lifeguard functionality can be split across multiple cores, exploiting further parallelism to speed up lifeguards.

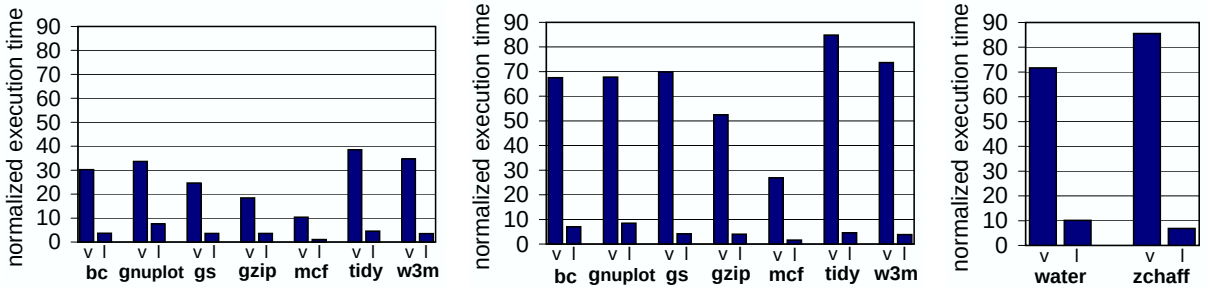
A key advantage of a log-based approach is that the log captures the dynamic history of a monitored program. Thus it enables lifeguards to use this history to *detect* sophisticated bugs or answer "how did I get here" *analysis* questions, as well as providing a means, when a problem is detected, to (selectively) *rewind* the monitored program and possibly perform on-the-fly bug *repair* [3].

Even when considering only bug detection, there are already significant challenges in making LBA efficient, including issues in capturing the log, reducing the log storage and communication bandwidth requirements, buffering and transporting the log, and consuming the log. Our initial design (Section 2) has begun to address these challenges, with some promising initial results (Section 3).

2. LOG-BASED ARCHITECTURES

Figure 1 depicts an example dual-core LBA system. As an application instruction retires, the capture hardware creates

ASID'06 October 21, 2006, San Jose, California, USA.



(a) AddrCheck (b) TaintCheck (c) LockSet

Figure 2: Execution times of LBA lifeguards (l) vs. Valgrind lifeguards (v), normalized to unmonitored execution times.

an event record that contains the instruction's (a) program counter, (b) type, (c) input and output operand identifiers, and (d) load/store memory address, if present.[1] Then, a hardware engine compresses the record to reduce the bandwidth pressure and buffer requirements on the log transport medium (the cache hierarchy in our design). We adapted value prediction-based compression [1] to achieve less than one byte per instruction with moderate chip area requirements.

Log record fetch is driven by the lifeguard, which is primarily organized as a collection of event handlers, each of which terminates by issuing an *nlba* (next LBA record) instruction. This operation causes the dispatch hardware to retrieve the next record from the decompression engine and execute the lifeguard handler associated with that type of event. Certain event values (such as the memory addresses of loads and stores) are simultaneously placed in the register file by the dispatch engine for ready lifeguard handler access.

To reduce overheads, the application core and the lifeguard core are not synchronized. They coordinate only through the log buffer, and hence log entry consumption at the lifeguard core typically lags behind event retirement on the application core. This enables pipeline-style processing at the lifeguard core. For example, although each *nlba* instruction causes a jump table lookup to retrieve the lifeguard handler address, the index can be determined very early.

While this lack of tight synchronization significantly improves performance, it also implies that there is typically a lag between the occurrence of a problem and its detection by a lifeguard. In order to contain the effects of bugs, the OS stalls each application syscall until the lifeguard finishes checking the remaining log entries that executed prior to the syscall invocation. In this way, lifeguards can prevent the propagation of errors beyond the application's process container.

3. PRELIMINARY EVALUATION

Our initial evaluation uses three diverse lifeguards: (i) AddrCheck [4] detects accesses to unallocated memory, double free(), and memory leaks; (ii) TaintCheck [5] detects security exploits by tracking the propagation of inputs, and checking if they eventually modify jump target addresses or other critical data; and (iii) LockSet [6] detects possible data races in multithreaded programs using the LockSet algorithm.

We ran the standard Fedora Core 2 and Valgrind 2.2.0 on Simics 2.2.14. Valgrind is a popular software-only approach that uses dynamic binary instrumentation to augment the monitored program with the desired lifeguard functionality [4]. We model single-CPI in-order cores with 16KB private split L1 caches and a 512KB shared L2 cache. For LBA support, we developed a trace generation tool to produce log record traces from applications, and a Simics extension module to read the log traces and perform event-driven lifeguard executions. We selected seven single-threaded benchmarks and two multi-threaded benchmarks, all of which were run to completion. On average, a benchmark executes 209 million x86 instructions, of which 51% are memory references. (See [2] for more details.)

In Figure 2, the Y-axis can be regarded as the slowdown of the monitoring approaches compared to normal (unmonitored) executions. We see that Valgrind lifeguards incur 10-85X slowdowns, which is consistent with the slowdowns reported in [4, 5]. Compared to Valgrind lifeguards, LBA lifeguards are 4-19X faster.

However, the overall LBA lifeguard slowdowns are still significant: on average, 3.9X slowdowns for AddrCheck, 4.8X slowdowns for TaintCheck, and 9.7X slowdowns for LockSet. We are working on a variety of techniques to further reduce this overhead, including parallelizing lifeguards and employing filtering techniques (e.g., address-range based filtering).

4. RELATED WORK

Our work contrasts with projects involving off-line reconstruction such as Flight Data Recorder [9] and BugNet [3] in that our goal of constant monitoring requires continuous consumption of the execution log. The work that is closest to ours in terms of its motivation is iWatcher [10], which invokes monitoring code in response to accesses to certain ranges of memory addresses. LBA differs in that it supports tracking data flow through all instructions—a crucial attribute for certain lifeguards such as TaintCheck. Other work [7, 8] uses dual-core processors to do dynamic checking only for specific lifeguards.

5. REFERENCES

[1] M. Burtscher. VPC3: A fast and effective

[1]Additional fields would be needed to enable rewind.

trace-compression algorithm. In *SIGMETRICS/PERFORMANCE*, 2004.

[2] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, and S. W. Schlosser. Logs and lifeguards: Accelerating dynamic program monitoring. Technical Report IRP-TR-06-05, Intel Research Pittsburgh, May 2006.

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

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

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

[6] 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.

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

[8] W. Shi, H.-H. S. Lee, L. Falk, and M. Ghosh. An integrated framework for dependable and revivable architectures using multicore processors. In *ISCA*, 2006.

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

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