

Ex-Mon: An Architectural Framework for Dynamic Program Monitoring on Multicore Processors

Guojin He, Antonia Zhai and Pen-Chung Yew
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455, USA
{guojinhe, zhai, yew}@cs.umn.edu

Abstract

For modern software systems that operate in complex execution environments, reliability is key. The ability for observing the internal states of an executing program can facilitate a spectrum of program execution monitors to enhance software reliability; and modern multicore processors provide the computing power needed by such monitors.

This paper presents Ex-Mon, novel hardware and software supports that enable efficient and flexible dynamic program execution monitoring. In Ex-Mon, a hardware-based extraction logic that can be configured dynamically by the monitoring software, is integrated onto each processor core. The extraction logic forwards events that are of interests to the monitoring software for correctness verification. We evaluate the effectiveness and efficiency of the proposed system by using it to detect memory-bugs in the SPEC2000 benchmark suite. The experiments show that performance overhead of Ex-Mon is 15% on average and 41.4% in the worst case. The bandwidth requirement of the proposed system is below 10 bits per cycle, which is acceptable considering the bandwidth capacity of today's state-of-the-art on-chip interconnect network.

1 Introduction

Today's software systems are extremely complex entities operating in more and more complex execution environments. Ensuring correct execution of such software systems is a challenging but important task. Modern multi-core microprocessors that are becoming increasingly common, provide significant computing power that can potentially be utilized to improve software reliability. One way to achieve this goal is to designate some of the cores as monitors to observe and verify the execution of other cores dynamically. With adequate hardware and

software support, such dynamic monitors can potentially enable the programmers to detect a wide range of software errors and enhance software reliability, with minimal performance penalty.

Software-based dynamic monitoring systems observe the internal states of executing programs through explicit instrumentation [12, 7, 11, 1]. Explicit instrumentation has two disadvantages: instrumentation can potentially change program behavior, which may not be acceptable for some software systems; it can also lead to significant performance overhead. For example, to detect memory bugs in general-purposed applications, Valgrind, a general dynamic binary analysis framework, incurs 22 times slowdown compare to un-monitored execution [12].

To reduce the performance overhead associated with software-based monitors, several researchers proposed architectural supports for dynamic monitoring [22, 16, 23, 13, 21, 18]. However, these proposals often target specific software bugs and cannot be extended to monitor generic program behaviors. Furthermore, most of these supports require some instrumentation to the original program that may not be acceptable for some monitoring requirement. Most importantly, once the proposed mechanism is activated, the software has little control over the hardware.

In this paper, we propose Ex-Mon, a novel dynamic monitoring framework that utilizes the computing power of multi-core processors. On one hand, Ex-Mon is flexible enough to support generic monitoring activities without requiring any instrumentation to the monitored programs; on the other hand, Ex-Mon incurs minimal performance penalty with the help of additional hardware supports. Ex-Mon provides the monitoring software with full control of the underlying hardware support to further lower the overhead associated with monitoring.

In Ex-Mon, a hardware-based extraction logic is integrated onto each core. This logic can be configured dynamically by the software, thus it allows the programmers

to selectively extract and forward information needed for monitoring. Monitoring activities are performed by monitor programs running on separate cores. The extraction logic is an attachment to the existing micro-architecture and Ex-Mon leverages existing on-chip interconnection network to deliver information from the monitored program to the monitoring program.

To evaluate the performance overhead of our design, we utilize Ex-Mon to detect memory bug in general-purpose applications. Detecting memory-bugs requires information of every memory operations in the monitored program, thus pose significant stress on the underlying hardware and software supports.

The main contributions of this papers are the following:

- Presenting Ex-Mon, a program execution monitoring framework that is capable of supporting a large spectrum of monitoring activities;
- Proposing flexible hardware supports that are capable of selectively extracting runtime information from the monitored program based on software specifications.
- Evaluating the effectiveness and the efficiency of the proposed framework by utilizing it to detect memory bugs.

The rest of paper is organized as follows: Section 2 gives an overview of the Ex-Mon framework; Section 3 describes the hardware support necessary for Ex-Mon; Section 4 describes how the monitoring framework can be used to detect memory-bugs; Section 5 evaluates the performance of Ex-Mon for detecting memory bug; Section 6 discusses related works; and Section 7 concludes this paper.

2 Overview

Ex-Mon is a framework for supporting dynamical execution monitors. To design and utilize this framework, we must address three issues, as shown in Figure 1: (i) parsing and interpreting monitoring specifications with programming language support; (ii) generating monitoring programs with compiler support; and (iii) monitoring program execution with hardware supports. While this paper focuses on the design and implementation of hardware supports, we briefly discuss the design issues for the other two.

Parsing and Interpreting Monitoring Specifications: Program correctness can be specified by a variety of logics and programming languages, such as the computational

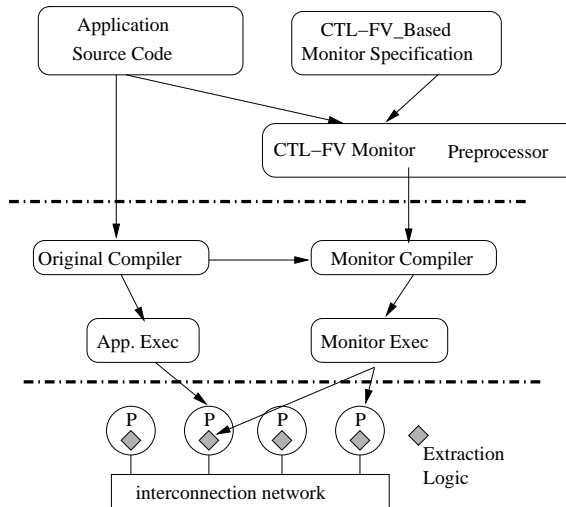


Figure 1: Dynamic program monitor framework

tree logic with free variables (CTL-FV) [9, 8]. A preprocessor parses these specifications, and converts them into monitoring routines that perform monitoring activities at runtime. Each monitoring routine is augmented with an event list indicating when the routine should be invoked¹. Figure 1 shows a CTL-FV monitor preprocessor that takes two inputs: the original source codes of the application, as well as a list of CTL-FV monitor specifications; and generates annotated monitoring routines.

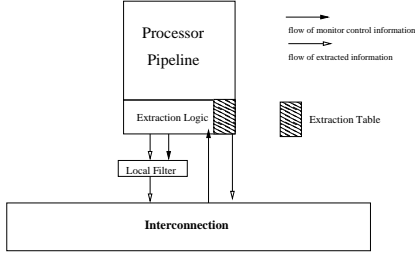
Generating Monitoring Programs: The annotated monitoring routines can be automatically converted to a monitor executable with the help of a compiler, as shown in Figure 1. To support execution monitors, an additional compilation path is incorporated into the compilation infrastructure. On this path, the monitor compiler takes the output from a preprocessor, and generates the monitoring executable. The monitoring executable contains three parts: a list of *monitor routines*, a dispatching routine that determines which monitor routines to invoke, as well as an updating routine that initializes and updates the extraction logic.

The preprocessor and the monitor compiler are currently under construction.

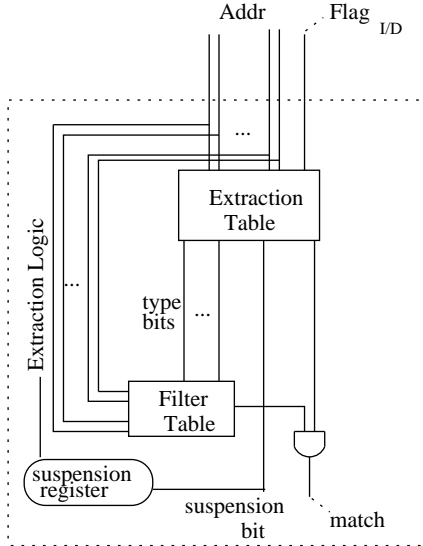
3 Hardware Support for Ex-Mon

In a multi-core environment, using one core to efficiently monitor the execution of another, requires additional hardware support. One of the key functionality of such hardware support is to selectively extract information needed by the monitoring core from the application core and the

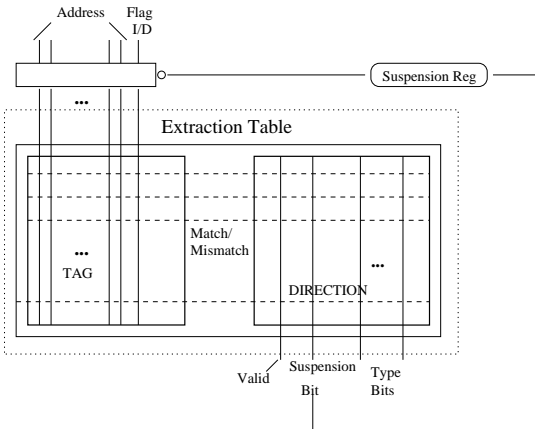
¹In this paper, we refer to every operation performed by the core that runs application as an event.



(a) Augment each core with extraction logic.



(b) Inside the extraction logic: local filter and extraction logic.



(c) Extraction table

Figure 2: Hardware support for Ex-Mon

extraction must operate at the speed of the processor to avoid stalling the application. This hardware support is referred to as extraction logic, as shown in Figure 2(a). The inputs to the extraction logic are the program counters and the memory addresses of committing memory in-

structions. The extraction logic decides whether the input corresponds to an event of interests to the monitoring software; and packs and forwards the necessary information if it does. As described in the The extraction logic is explicitly managed by the monitoring software. This is achieved by allowing the monitoring software to initiate and update the extraction logic at runtime.

3.1 Extraction Logic

To selectively extract and forward information from the application core, the extraction logic performs the following functions: i) obtains information from the monitored core; ii) determines whether the information is relevant; and iii) packs and forward the information. In the rest of this section, we provide a detailed description of the extraction logic.

3.1.1 Fetching Information from the Pipeline

In most modern processors, when an instruction reaches the commit stage, the PC and the result of the committed instruction are available in Reorder Buffer(ROB) [15]. Thus, the extraction logic can obtain information needed from the ROB. For memory instructions, the extraction logic can obtain the accessed address and value from the load/store queue.

3.1.2 Extraction Table

The key component of the extraction logic is the extraction table, as shown in Figure 2(c). The inputs to the extraction table are the PC of the committing instruction or the memory address of the committing memory instruction, as well as a one-bit I/D flag differenting the two. The outputs include a one-bit valid signal, a one-bit suspension signal, and a two-bit *type* signal connected to the local filter. The valid signal indicates whether the incoming address has a match; the suspension signal indicates whether the the committing instruction corresponds to a monitored event that suspends the extraction table; and the *type* bits can be used with the local filter to avoid forwarding redundant information. The *type* bits will be described later in this section.

The structure of the extraction table is shown in Figure 2(c). The extraction table consists of two components. The TAG component has the keys to be matched and is implemented with content-addressable-memory (CAM)[14]. Each entry of the CAM corresponds to a key and a one bit I/D Flag. The DIRECTION component of the extraction table is implemented as a small cache. Each word-line of this cache is driven by the corresponding output from the CAM. When an instruction commits, the PC of

this instruction is sent to the CAM and the I/D flag is set to **I**, if a block in the CAM matches the input, the corresponding wordline is set and DIRECTION bits are retrieved from the small cache. If the instruction is a memory instruction, the process is repeated with data address accessed and the I/D flag set to **D**. By checking the valid bit, the extraction logic can determine whether to forward the result of the committing instruction. To support simultaneous lookups in the table, this structure can be extended with multiple ports. Many monitor activities require monitoring contiguous memory addresses, thus having entries representing ranges of memory locations is beneficial. This feature can be supported with a ternary CAM where a storage cell represents "0", "1" or "X" indicating "don't care" [14]. For example, to monitor all accesses to the elements of an array located between 0x8000a000 and 0x8000afff, only one entry with address key as 0x8000aXXX needs to be entered to the table.

For some segments of execution, monitor software may require the extraction logic to forward all events and bypass the extraction table. For example, when software is updating the extraction logic, the extraction table becomes temporarily inconsistent and the use of it should be prohibited until the update is done. To support this function, Ex-Mon uses a one-bit *suspension* register to control the use of the extraction table—the extraction table is bypassed when suspension register is set. The mechanism of updating the extraction table is the following: at the initialization stage, suspension register is initially unset and monitor software set the suspension bit of every extraction table entry that can cause a table update. At runtime, every time an extraction table entry is matched, its suspension bit is copied to the suspension register. When a committing instruction matches an entry that has suspension bit as "1", the monitor software is notified to start updating the extraction table. Meanwhile, from the next instruction, the extraction logic bypasses the extraction table and forwards every instruction until the monitor software reset the suspension register at the end of the update.

3.1.3 Information Forwarding

Once an event of interest is identified, the extraction logic packs the necessary information and writes them to a dedicated area of the shared memory, which we refer to as the *communication queue*. The *communication queue* is accessed with regular load/store mechanisms. To support accesses to the *communication queue*, each core is augmented with four registers: one indicates the base address of the communication queue; one corresponds to the end

of the *communication queue*. The *communication queue* is a circular buffer, thus the other two are used as the head and the tail of the buffer. The head points to the next available slot and the tail points to the next element to be consumed. While the head is update by the extraction logic, the tail is updated by the monitor. When the queue is full, the monitored program must be stalled. This is the major reason for performance overhead in Ex-Mon. The monitoring program continuously consumes packets from this queue in a FIFO order. Each packet contains the following fields:

- Address:** the address of the instruction committed or address accessed;
- I/D flag:** indicate whether the entry is an instruction address or a data address.
- Value:** the result of the instruction committed or the value stored by a store instruction.

3.2 Eliminating Redundant Forwarding

Although the extraction logic allows programmers to specify events of interests, traffic for some monitor activities can still be excessive. Fortunately, there are optimization opportunities. For example, in the case of detecting dangling pointers, when an accessed memory location is proven to be in an allocated block, all future accesses to the same location do not need further verification until the state of this memory location is changed by certain events, such as function calls to `realloc` and `free`.

To take advantage of this opportunity, we add an auxiliary logic to the extraction logic, the local filter. The local filter uses a small fully associative cache to store recently forwarded items. If an input address matches an entry in this filter, the event will not be forwarded. The local filter is initially empty and updated by extraction logic. This can be achieved by utilizing the *type* bits in the extraction logic. The *type* bits are the control signals for the local filter. The two *type* bits are:

- ONCE:** when the `once` bit is set, the event only needs to be forwarded once, thus should be entered in the local filter;
- FLUSH:** the local filter must be cleared when this event occurs.

Figure 2(b) shows how the local filter works together with the extraction logic. The local filter is initially empty and when an entry with ONCE bit set in the extraction table is matched, the address with I/D flag is written into the local filter. When the address appears again, it will not be

forwarded. If the event is a FLUSH event, the entire local filter is flushed. Our experiments show that a 32-entry local filter with least recently used replacement policy can improve the performance of memory bug detection significantly.

3.3 Extraction Table Overflow

Although a relatively small extraction table is sufficient for the experiments in this paper, it is possible for a monitor to require more extraction table entries than what are available. This can be resolved by taking advantage of program locality and utilizing the extraction table updating mechanism. While the monitoring software maintains a list of all entries that should be in the extraction table, it is possible that only a subset of entries are in the hardware extraction table at runtime. The monitoring software initializes the extraction table with entries that are most likely to be used in the near future, when the application core starts to execute instructions in a different part of the program, or access data in a different part of the memory, the monitoring software updates the extraction table by replacing old entries that are not likely to be used in the near future. To invoke the monitoring software’s updating mechanism, entries in the extraction table are reserved to represent events not in the extraction table. When an input matches one of these entries, the extraction table is suspended and a message is sent to the monitoring core requesting updates.

3.4 Monitoring Software

In Ex-Mon, monitor software contains (i) a set of monitoring routines, (ii) a dispatching routine that activates the appropriate monitoring routines, and (iii) routines to initiate and update the extraction table.

The dispatching routine invokes the desired monitoring routines for every incoming event. Monitoring routines not only verify whether the incoming event violates any correctness specification, but also update the states that is needed for future verifications. The monitor program must maintain sufficient states to verify all specifications. Ideally, to implement an efficient monitoring system, only the minimal amount of events are extracted and forwarded to construct the states. It is worth pointing out that as the number and types of events increase, the efficiency of the dispatching routine can become important. Thus, the dispatching logic must be implemented efficiently. Currently, the monitor compiler is under development, and the monitor programs used in this paper are developed manually.

4 Catching Memory-Bugs with Ex-Mon

To evaluate the effectiveness and efficiency of Ex-Mon, we will demonstrate how to utilize it to detect memory bugs. Memory bugs include memory leak, dangling pointers, loads to uninitialized memory locations and double free. In the rest of this section, we will first show an example of how to specify memory bug with CTL-FV [9, 8] rules; then provide an algorithmic description of the monitoring software; finally, we show how the monitor program works with the proposed hardware supports.

4.1 Rule Specification for Memory Bug Detection

Memory-bug detection has a set of well-known rules, in this section we attempt to specify these rules with mathematical logic. We use the Computational Tree Logic with Free Variables (CTL-FV) to formalize the description of the rules so that preprocessor can process them for the monitor-aware compiler. For example, the statement an address passed as a parameter to the free function must be a return value of a malloc function can be specified in CTL-FV using the following statement: $\mathbf{AG}(free(addr) \rightarrow addr = malloc(_))$

Here **A** and **G** are operators of CTL-FV, indicating that the formula follows it must be true on all possible execution paths of the program. *Addr* is a free variable, it can be substituted by all applicable addresses. These rules define the correctness of the monitored programs.

4.2 Generating Monitor Program For Memory Bugs Detection

The monitor program can be automatically generated from the rules with the help of a preprocessor and a monitor-aware compiler. In this paper, we manually created the monitor program, because the monitor compiler is under development. The algorithm of the dispatching routine, is showed in Figure 3(a).

The dispatching routine is well structured. It initializes the extraction table with the *Event_List* created by the monitor compiler (manually in our case). The dispatching routine is a loop with a switch-case statement. It invokes the appropriate monitor routines when an event is observed in the communication queue. A special event, EXIT, corresponds to the termination of the monitored program, will lead to the termination of the dispatching routine.

data declarations:

```
typedef struct event_type
    {Address, Value, I/D flag }
event_type event;
List of events for initializing and
updating the extraction table.
event_type Event_List[];
```

algorithm DispatchRoutine:

```
InitializeExtractionTable(Event_List);
InitializeDispatchMap();
while (true)
    event = ReadQueue();
    switch (DispatchMapLookup(event.Address))
        case MALLOC:
            monitor_malloc(event);
            break;
        case FREE:
            monitor_free(event);
            break;
        case LOAD:
            monitor_dangling_pointer(event);
            break;
        . . .
        case EXIT:
            monitor_exit(event);
            break;
    end switch
end while
end algorithm
```

(a) Dispatching routine.

monitor_free(event)

```
begin
    if (AllocatedBlockLookup(event.Address))
        AllocatedBlockRemove(event.Address);
        return;
    else
        ReportIllegalFree(event);
    end if
end
```

(b) *FREE* event monitor routine.

Figure 3: Monitoring software for detecting memory bugs.

An example monitor routine for *FREE* event is shown in Figure 3(b). This routine is executed when a call to the *free* function is observed. The *monitor_free* first looks up freed block on the allocated block list, which is maintained by the monitor software at run time. If the address passed as a parameter to the *free* function matches the address of an allocated block, the call to function *free* is proven to be safe, and the correspondent block should be

removed from the allocated block list. Otherwise, an illegal call to *free* function is detected and reported.

4.3 Catching Memory Bugs Dynamically

Now we show how Ex-Mon hardware support facilitates the detection of memory bugs by describing four aspects of monitoring work: first we will introduce the initialization of the extraction table; then we will show how monitor software checks incoming events; we will also examine how the local filter can play a role to improve performance; and finally, we will discuss how to handle cases where the return value of memory management library is not available outside of the library codes.

For memory bugs detection, the events of interests are commits of instructions related to calls of memory management functions, such as *malloc*, *free*, *realloc* and *calloc*, as well as all accesses to the heap memory. The instructions related to call to memory management functions include parameter set up instructions and return value copy instructions. We can initialize the extraction table with the PC of these instructions. However, we cannot afford to initialize the extraction table with all heap memory locations. In stead, we initialize the extraction table with a range that corresponds to the entire heap memory space.

At runtime, the monitoring software parses incoming events and maintains a data structure that corresponds to the allocated block in the heap memory. Based on this data structure, the monitor software is able to verify whether any correctness specification has been violated. For example, when a memory location is read, the monitor software checks if the location has been allocated and initialized. If not allocated, the load is through a dangling pointer; if allocated, but not initialized, the load is an uninitialized read. When the monitored application has completed, the commit of one epilogue instructions will match an entry in the extraction table, and trigger the monitoring program to detect memory leakage bugs and wrap us execution.

In memory bug detection, we can use the local filter to improve performance as described in Section 3. Here all heap reference events have their *ONCE* bit set, therefore, when an accessed address is forwarded, it is entered in the local filter to avoid repeated forwarding. Thus, if a memory location of a load is already sent to the monitoring software, it will not be sent again until the state of that memory allocation changes. Commits of call instructions to memory management functions has their *FLUSH* bit set, because these functions can change the memory states.

With aggressive compiler optimizations, it may not be

easy for the extraction logic to determine the return value of a memory management function without accessing the source codes. In this case, we force the extraction logic to forward information of every committed instruction to the monitoring software while the execution stays in the memory management functions. This can be achieved by placing entries that correspond the calls of the memory management function in the extraction table with the suspension bit set. When the calls occur, suspension register will be set, and the extraction table will be bypassed so that all the instructions in the memory management function will be forwarded. Once the monitoring software has identified the return value from those forwarded information, it resets the suspension register.

5 Evaluation

We evaluate the Ex-Mon framework using the Simics [5] simulation environment, a full system simulation platform. We augment the simulator with the Wisconsin GEMS [10] infrastructure for a detailed simulation of the underlying memory hierarchy. We simulate a multicore system with 2 cores, each core has its own private L1 instruction and data cache, while sharing a unified L2 cache. The private L1 caches are 64KB in size and 4-way set-associative. They have 64Bytes cache lines and 3-cycle access latency. The L2 cache is 8MB in size and 4-way set-associative. It has 64Bytes cache lines, and 6-cycle access latency. The main memory is 2GB in size with 80-cycle access latency. The extraction table has 1K entries and the local filter has 32 entries.

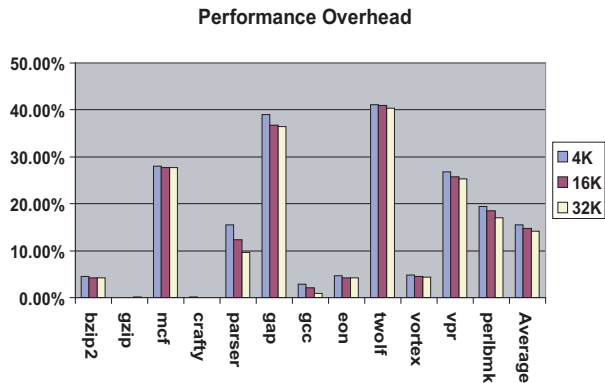
We simulate execution and synchronization of the monitored and monitoring programs, as well as the bandwidth requirements of the extraction logic.

5.1 Benchmarks

To evaluate the effectiveness and efficiency of Ex-Mon, we must identify the real-world applications with correctness specifications. However, there is no standard benchmark for evaluating program execution monitor systems. In this paper, we evaluate the performance overhead of memory bug detection using the SPECINT 2000 [19] benchmark suite, with injected memory bugs.

Under Ex-Mon framework, we manually developed monitoring software that detects a set of well-known memory bugs, including double free, memory leak, dangling pointer, and uninitialized load dynamically. In our implementation, memory bugs are logged when they are detected, and reported at the end of the execution.

We simulate the SPECINT 2000 benchmarks with the



(a) Performance overhead for different communication queue sizes.

Benchmark Name	Number of Cycles Per Event	Time Spent Processing Event	Time Spent Fetching Event
bzip2	71.27	92.98%	7.02%
mcf	15.83	68.41%	31.59%
crafty	629.65	99.21%	0.79%
parser	28.55	82.49%	17.51%
gap	14.29	65.01%	34.99%
gcc	21.12	76.33%	23.67%
eon	33.09	84.89%	15.11%
twolf	13.28	62.34%	37.66%
vortex	13.54	63.08%	36.92%
vpr	13.10	61.84%	38.16%
perlbnk	49.38	89.87%	10.13%
Average	100.49	78.73%	21.27%

(b) Execution time breakdown of the monitoring core.

Figure 4: Performance evaluation of Ex-Mon for detecting memory bugs on SPEC2000 integer benchmarks.

ref input set. For a reasonable simulation time, we simulated one billion instructions after skip the initialization phase of the benchmark.

5.2 Results

At runtime, the monitor program consumes data from the *communication queue* in a FIFO order. In some portion of execution, the workload of the monitoring program is higher than that of the monitored program, and thus packets in the communications queue cannot be processed in a timely manner. When the communication queue is full, the execution of the monitored program must be stalled, and performance degrades. This is the main performance penalty evaluated in our work. Figure 4(a) shows the performance impact of Ex-Mon, compared against un-monitored executions. The performance penalty for different benchmarks vary significantly. For a 4K communication queue, the average performance overhead is 15.58%. However, for some benchmarks, such as CRAFTY, the performance overhead is negligible, while for some other benchmarks, such as TWOLF and GAP, the

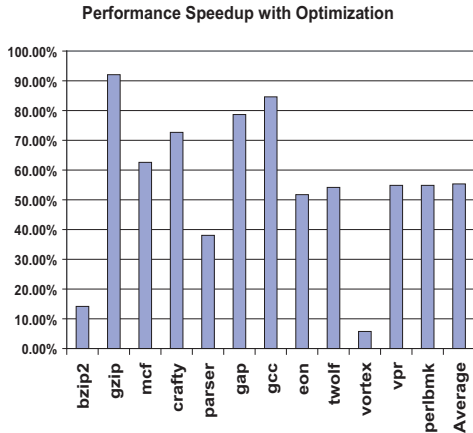


Figure 5: Speedup achieved by the local filter.

performance overhead is nearly 40%. We vary the size of the communication queue to evaluate the sensitivity of our performance results. We found that the benefit of increasing the communication queue size is marginal, e.g. the average overhead is reduced from 15.58% to 14.21% when the queue size is increased from 4K to 32K.

Figure 4(b) shows the breakdown of monitor core execution time with 32k communication queue. The execution time of monitor core can be divided into two categories: processing time and fetching time. Here the processing time includes the time of dispatching the information to different monitor routines and the time of running monitor routines. The fetching time includes the time of waiting for the communication queue to be ready and the time of actual reading. Cycles per event is the average interval of two consecutive events that tells how fast events come into the communication queue. As we can see, the faster the events come, the more time is spent on reading the communication queue.

5.2.1 Impact of Local Filter

Utilizing the local filter to reduce the number of packets forwarded through the communication queue has a significant performance impact. Figure 5 shows the speedup achieved by the Ex-Mon system with a 32-entry local filter incorporated compared against one without a local filter. All benchmarks are able to benefit significantly from this optimization. On average, we are able to achieve 55.36% performance improvement. This optimization is incorporated for results in Figure 4(a).

5.2.2 Bandwidth Requirement

The bandwidth requirements for communicating packets between the monitored core and the monitoring core may

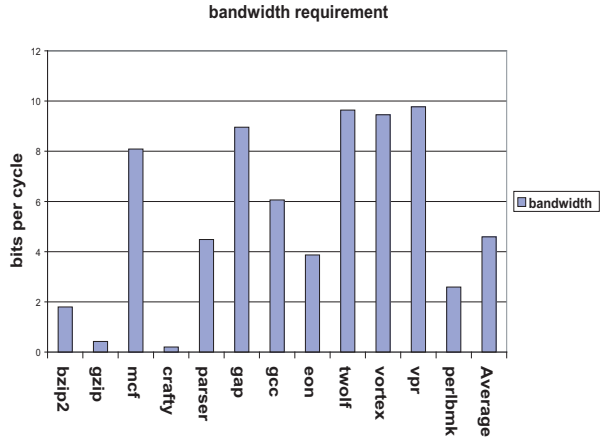


Figure 6: Bandwidth requirement of memory bug detection.

become significant and limit program performance. In this section, we evaluate the bandwidth requirement in details.

Figure 6 shows the average bandwidth requirement for all SPEC2000INT benchmarks. We observe that none of the benchmarks requires a bandwidth of more than 10 bits per cycle. For a 1.3GHz core, this translates to a bandwidth requirement of approximately 1.6GB/S. Such bandwidth requirement can be easily satisfied by the on-chip interconnection network of a state-of-the-art multi-core system [17]. However, it is worth pointing out that, the peak bandwidth requirement of the monitor system can potentially be much larger than the average bandwidth requirement. Thus, further investigation is required to estimate the system performance under peak bandwidth requirement.

Chen *et. al* [2] have proposed to optimize monitoring systems by compressing forwarded packets, and implement more efficient dispatching logic. The optimization techniques proposed by Chen *et. al* [2] can potentially be incorporated into Ex-Mon.

6 Related Work

Several software tools have been developed for runtime monitoring: *Valgrind* [12], *Purify* [7], *CCured* [3] and *DIDUCE* [6]. These tools instrument monitored codes, and creates significant performance overhead. Furthermore, instrumentation can potentially change program behavior and introduce additional vulnerability. Valgrind [12], for example, is a dynamic binary instrumentation framework that enables a variety of monitoring activities. It uses elaborate dynamic instrumentation to maintain shadow values for monitoring routines.

To overcome the performance overhead associated with the software-based approaches, several proposals have suggested hardware supports for execution monitoring [23, 2, 13, 21, 18, 4, 16, 22, 20]. Most of the proposals only target a specific class of program behavior, such as memory-bugs. MemTracker [21] and HeapMon [18] associate each word of data in memory with a few bits of state and track state transition with hardware tables. MemTracker is capable of detecting a set of well-known memory bugs for SPEC2000 benchmarks with negligible performance overhead. Raksha[4] provides architecture support for dynamic information flow tracking, and is able to detect a series of high level software security problem at runtime. It introduces tag checking, as well as propagation registers and logics at every stage of the pipeline and extends the ISA with new instructions. iWatcher [23] is a memory location based dynamic monitor that utilizes TLS supports to execute monitoring routines and instrumentation in the user programs to control these hardware features. Oplinger and Lam's [13] proposal allows the programmers to insert monitoring functions into the monitored program, and uses TLS hardware support to execute these monitoring functions in parallel with the monitored program.

The Ex-Mon framework described in this paper differs from the software-based approaches by utilizing hardware-based extraction logic to extract information directly from an executing program. On the other hand, Ex-Mon differs from the previous hardware approaches by supporting generic monitoring requirements and avoiding tapping into all pipeline stages of the micro-architecture. Furthermore, unlike most of the previous hardware approaches, Ex-Mon framework allows programmers to specify monitoring requirements efficiently and requires no instrumentation to the monitored program.

The most relevant related work is the log-based architecture (LBA) proposed by Chen *et. al* [2]. While the LBA and the Ex-Mon frameworks both require information of committed instructions to be forwarded to the monitors core, they target different optimization opportunities with different hardware support. LBA uses a hardware compression/decompression logic to reduce information forwarding and improve performance, while Ex-Mon allows the programmer to specify what information is needed by the monitor. We believe that the optimization proposed in the LBA and the Ex-Mon framework are complementary, and can work in tandem.

7 Conclusions

This paper presents Ex-Mon, software and hardware supports that facilitate a wide spectrum of program execution monitors. Ex-Mon targets multicore architectures by augmenting each core with extraction logics to obtain information from an executing program. For a particular monitoring task, the extraction logic can be configured by the monitoring program, based on the programmers' specifications. The monitoring program can be automatically generated by a monitor-aware compiler. During the execution of monitored program, the extraction logic selectively extracts information when events of interests occur and forward the information to the monitoring software running on a separate core. Only forwarding information for events of interests can reduce performance overhead and improve the efficiency of the proposed system. We evaluate the performance overhead and bandwidth requirement of Ex-Mon for memory bug detection with SPECINT2000 benchmarks. We observe a 15.58% performance overhead on average and 41.4% in the worst case. The bandwidth requirement generated by the monitoring infrastructure is below 10 bits per cycle, which is acceptable considering the bandwidth capacity of the state-of-the-art on-chip interconnect network.

References

- [1] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN 94 Conference on Programming Language Design and Implementation (PLDI'94)*, 1994.
- [2] S Chen, B Falsafi, P Gibbons, M Kozuch, T Mowry, R Teodorescu, A Ailamaki, L Fix, G Ganger, and S Schlosser. Logs and lifeguards: Accelerating dynamic program monitoring. Technical Report IRP-TR-06-05, Intel Research Pittsburgh Lab, Jun 2006.
- [3] J. Condit, M. Harren, S. McPeak, G. Necula, and W. Weimer. Ccured in the real world. In *ACM SIGPLAN 03 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 232–244, San Diego, California, 2003.
- [4] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A flexible information flow architecture for software security. In *34th Annual International Symposium on Computer Architecture (ISCA '07)*, 2007.
- [5] Peter S. Magnusson et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

- [6] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ACM SIGSOFT 2002 International Conference on Software Engineering (ICSE'02)*, Orlando, Florida, 2002.
- [7] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *"the Winter 1992 USENIX Conference"*, pages 125–138, San Francisco, California, 1991.
- [8] David Lacey and Oege de Moor. Imperative program transformation by rewriting. *Lecture Notes in Computer Science*, 2027:52+, 2001.
- [9] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Symposium on Principles of Programming Languages*, pages 283–294, 2002.
- [10] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *Computer Architecture News*, 2005.
- [11] N. Mitchell and G. Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. *European Conference on Object-Oriented Programming (ECOOP 03)*, 2003.
- [12] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN 07 Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, USA, June 2007.
- [13] Jeffrey Oplinger and Monica S. Lam. Enhancing software reliability with speculative threads. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2002.
- [14] Kostas Pagiamtzis and Ali Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, March 2006.
- [15] David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.
- [16] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, Feb 2005.
- [17] Manish Shah, Jama Barreh, Jeff Brooks, Robert Golla, Gregory Grohoski, Nils Gura, Rick Hetherington, Paul Jordan, Mark Luttrell, Christopher Olson, Bikram Saha, Denis Sheahan, Lawrence Spracklen, and Aaron Wynn. Ultrasparc t2: A highly-threaded, power-efficient, sparc soc. *Asian Solid-State Circuits Conference*, 2007, 2007.
- [18] 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. Res. Dev.*, 50(2/3):261–275, 2006.
- [19] Standard Performance Evaluation Corporation. The SPEC Benchmark Suite. <http://www.specbench.org>.
- [20] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *14th International Symposium on High-Performance Computer Architecture (HPCA-14)*, Feb 2008.
- [21] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *13th International Symposium on High-Performance Computer Architecture (HPCA-13)*, Feb 2007.
- [22] E. Witchel, J. Cates, and Krste Asanović. Mondrian memory protection. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, Oct 2002.
- [23] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iwatcher: Simple, general architectural support for software debugging. In *31st Annual International Symposium on Computer Architecture (ISCA '04)*, 2004.