Hardware Supported Flexible Monitoring: Early Results*

Antonia Zhai, Guojin He, and Mats P.E. Heimdahl

University of Minnesota

zhai@cs.umn.edu
guojinhe@cs.umn.edu
heimdahl@cs.umn.edu

Abstract. Monitoring of software's execution is crucial in numerous software development tasks. Current monitoring efforts generally require extensive instrumentation of the software or dedicated hardware test rig designed to provide visibility into the software. To fully understand software's behavior, the production software must be studied in its production environment. To address this fundamental software engineering challenges, we propose a compiler and hardware supported framework for monitoring and observation of software-intensive systems.

We place three fundamental requirements on our monitoring framework. The monitoring must be *non-intrusive*, *low-overhead*, and *predictable* so that the software is not unduly disturbed. The framework must also allow *low-level monitoring* and be *highly flexible* so we can accommodate a broad range of crucial monitoring activities.

The general idea behind our work is that to make dramatic progress in non-intrusive, predictable, and fine-grained monitoring, we must change how software is compiled and how hardware is designed; a software-monitoring framework covering the development of monitors, through compilation, and down to the hardware is essential. To achieve our goals, we have pursued an approach leveraging the rapid emergence of multi-core processor architectures to achieve a non-intrusive, predictable, fine-grained, and highly flexible general purpose monitoring framework.

In this report we describe our initial steps in this direction and provide some preliminary performance results achieved with this new multi-core architecture. We use separate cores for the execution of the application to be monitored and the monitors. We augment each core with identical programmable extraction logic that can observe an application executing on the core as its program state changes.

1 Introduction

Monitoring of software's execution is crucial in numerous software development tasks. For example, test oracles monitor an application's execution (the outputs

* This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, and the L-3 Titan Group.

and typically part of the application's internal state), test adequacy coverage analysis tools must determine what portion of an application have been executed (and possibly how those parts of the application were reached), run-time security and safety monitors in critical systems must determine if security and safety policies are maintained by the application, and—of course—all other run time verification tasks envisioned in the Run-Time Verification series of workshops. These monitoring tasks generally require three crucial properties. First, to ensure that the performance of the monitored software is not degraded to a point where the monitoring is simply infeasible, the monitoring must incur *low overhead*. Second, since monitors are likely to change during the lifetime of the monitored software (for example, if safety and security policies change) or—in the case of test oracles and coverage measurement tools—will be removed entirely at some point, we must have *predictable behavior*, in terms of both functional behavior and performance, so that we can predict the impact of changed or removed monitors. Finally, to enable access to internal program state information (crucial information in all testing, and safety and security monitoring), we must have *fine-grained monitoring* where both the state information in the monitored program as well as the program point where the monitoring takes place can be selected to suit the task at hand.

Several communities have addressed the monitoring problem from various angles. For instance, test oracles are developed largely ad hoc and rely on intrusive software instrumentation of the software under study [**?**]; runtime verification generally relies on software instrumentation with high overhead [**?**], and most of the dedicated hardware solutions are targeted towards the monitoring for narrow properties [**?**,**?**,**?**,**?**,**?**,**?**,**?**]. Unfortunately, these approaches are fragmented, largely ad-hoc, and address narrow aspects of the monitoring problem (such as efficient implementation of monitorspropagation [**?**,**?**,**?**,**?**,**?**,**?**,**?**]).

These monitoring efforts typically require extensive instrumentation of the software and/or execution of the software in a dedicated hardware test-rig or emulator. Under such conditions the software's behavior is not the same as it would be in its intended target environment. To fully understand software's behavior—in particular embedded software's behavior—the *production* software *must* be studied in its *production environment*.

To alleviate the problems with overhead and predictability of instrumentation for monitoring purposes, we have pursued an approach leveraging the rapid emergence of multi-core processor architectures [**?**,**?**,**?**,**?**,**?**] to achieve a non-intrusive, predictable, fine-grained, and highly flexible general purpose monitoring framework through *monitoring-aware* compilers coupled with *novel architectural enhancements* to the multi-core architectures.

In this report we describe our initial steps in this direction and provide some preliminary performance results achieved with this new multi-core architecture. We use separate cores for the execution of the application to be monitored and the monitors. We augment each core with identical programmable *extraction logic* that can observe an application executing on the core as its program state changes. If a state change that needs to be monitored occurs, the extraction

logic will pack the state change into a message and send it to one of the *monitor cores* for verification. In this architecture, one or more cores can be used to monitor and potentially share the workload, while introducing little or no intrusion to the software being monitored. If and when the monitors are no longer needed, the processor capacity previously occupied by the monitors can be reclaimed and allocated to production software without affecting the software originally being monitored. The communication between cores can be achieved either through a dedicated or an existing on-chip interconnection network depending on the need for predictable behavior. In the work presented here, we use an existing interconnection network and provide performance data for two monitoring problems—tracking memory problems such as memory leaks and taint analysis—that thoroughly stress the ability to efficiently extract data from an application and communicate that data to a monitor.

Although our work on a monitoring aware compilers and multi-core architecture is far from complete, the initial steps and performance evaluation presented in this report illustrate the potential for this approach as we attempt to make run-time verification and monitoring in the *production* environment standard practice.

The remainder of the paper is organized as follows. We provide the motivation for our work in Section 2 and an overview of our compiler and architecturally supported vision in Section 3. We present the details of the Ex-Mon architecture in Section 4; illustrate the effectiveness of the proposed architecture with two case studies in Section 5. Finally in Sections 6, we discuss the implications of our results and point to future directions.

2 Motivation and Problem Overview

Monitoring of the execution of a software system plays a central role in numerous software development activities. Monitoring is prevalent in testing (e.g., test oracles and test coverage tools), debugging (e.g., breakpoints and watch variables), run-time verification, safety and security monitors and interlocks, etc. Unfortunately, the performance penalties (both in terms of cost and predictability of the executions) are significant obstacles to effective use in the software development process. The motivation for our work in compiler and hardware architecture support for monitoring have been from problems and opportunities in primarily two areas: (1) the cost and difficulty of thoroughly testing embedded critical applications and (2) the opportunities for monitoring offered by model-based software development.

Software Testing: A test oracle must monitor both the outputs from an application as well as internal state information since the fault finding can be severely affected by which and how many data items are being observed by the oracle. Thus, instrumentation of the application is generally required to collect test information in log-files or provide it to on-line oracles. Either way, the overhead associated with the data collection can be large enough to delay projects for

months. In practice, we have with our industrial collaborators seen that determining that a small modification has not "broken" an embedded subsystem by simply rerunning its test suite can take weeks. If additional modifications are needed, the delays add up and quickly lead to costly schedule delays. Accelerated testing through hardware support could provide orders of magnitude speedup of this process.

A worse situation can emerge when one attempts to measure how well a test suite has covered an application as judged by some test-adequacy criterion. Of particular interest in our previous work has been the Modified Condition and Decision Coverage (MC/DC) criterion [**?**] used in the avionics industry and required in a standard such as DO-178B [**?**]. For the most critical applications, MC/DC has to be demonstrated on the *object code* (as opposed on the source code) and extensive instrumentation of the code is needed to establish this coverage. Unfortunately, current approaches relying on instrumentation leads to such performance degradation and increase in code size that only portions of the application can be instrumented at any time; the full test suite is run to establish coverage of the instrumented part of the application, another part is instrumented, the test suite rerun, repeat. The problem is so severe that simply establishing coverage can be comparable in cost to test development. In addition, all testing will have to be repeated *without* the instrumentation since there are no guarantees that the instrumentation did not change the behavior of the application under test. Clearly, low-overhead and *predictable* monitoring would be hugely beneficial.

Opportunities in Model-Based Development: In model-based development, the development effort is centered around a formal description of the proposed software system. There are currently numerous commercial tools that attempt to provide these capabilities—commercial tools are, for example, CADE from Esterel Technologies [**?**], Statemate from i-Logix [**?**], and Simulink and Stateflow from The Mathworks Inc. [**?**,**?**].

Note here that this process leaves us with several development artifacts that are in an executable form; first, the source code that will be used to control the system under development (typically C or C++ code); second, the formal (or semi-formal) models from which the source code was derived (in our application domain, most likely Simulink and Stateflow models); third, collections of formalized required properties of the software derived for verification and testing purposes (generally captured as synchronous observers expressed in the modeling language or as temporal logic formulas for model checking). Currently, after the source code has been developed and tested, the model and property artifacts are used only for maintenance and documentation purposes. In our work, we envision these artifacts to see additional use as monitors after software deployment.

As a concrete example, consider a recent project where we in collaboration with Rockwell Collins Inc. developed a formal model for the mode logic of a Flight Guidance System [**?**]. In the project, the system requirements were provided as informal "shall" statements. These requirements were relatively mature and well-understood. We then created a model using Simulink from Math-

works [?]; the model when completed consisted of about 4,500 Simulink blocks. Throughout creation of the model, we continually used the execution capabilities of Simulink to execute the model and informally confirm that it behaved as we expected. In the formal verification phase, we manually translated the shall statements into formal properties stated over the Simulink model in CTL and the NuSMV model checker [?] was then used to confirm whether the property held over the model or not. The effort resulted in 300+ CTL properties based on the informal requirements.

In a production setting, after the Simulink model has been adequately validated, it would be used as a basis for the manual design and implementation of the production software (here the development was governed by a standard for airborne software—DO-178B [?]—that takes a highly skeptical view of code generation). If we could execute the original Simulink model as a synchronous observer next to the production software, we could conceivably detect potential design or implementation faults as well as possible hardware faults (single bit upsets, stuck at faults, etc.) that might affect the execution of the production software; this would be a highly valuable monitoring capability that would complement the fault detection of failures in the sensors, actuators, and the environment outlined in the previous section. In addition, if the requirements on the system were formalized as declarative properties (such as the CTL properties discussed above), these properties could be deployed as additional monitors used to detect both possible faults in the original model as well as additional design, implementation, and hardware faults affecting the correct operation of the software. Given appropriate support for efficient monitoring, we hypothesize that such monitors working in concert will provide a highly effective fault detection scheme for the embedded application of interest in our proposed effort.

3 Monitoring Overview

A program execution monitor observes the internal states of an application and verifies that a set of properties defined over the application are satisfied. Note here that access to fine-grained internal application state information is essential for our target monitoring tasks, for example, a test oracle monitor will most often need access to many internal variables and a test adequacy coverage monitor might need to know about all branch decisions as well as the conditions guarding the branches (as in the case of MC/DC).

To perform the monitoring activities, a subset of the internal state of the application is exposed to the monitor through some form of communication. The implementation of the monitor can be viewed as a collection of *monitoring routines*. For example, the monitor might be a test oracle monitoring for a large number of required functions or invariants, each implemented as a separate monitoring routine. At runtime, upon receiving a state update the monitor invokes the subset of the of monitor routines that are affected; monitoring routines not affected by the state change need not be evaluated. The monitor is

responsible for keeping track of the persistent state information needed to verify the properties of interest.

Monitoring routines can be invoked in two distinct ways: *explicit* and *implicit* invocation. Monitoring routines with *explicit invocation* are invoked when the execution of the application under study reaches a certain program point. Typical examples are monitoring routines performing pre- and post- condition checking that must be invoked at the entry and exit of a function. A monitoring routine with *implicit invocation* is invoked when an application state component of interest to the monitoring routine changes. A typical example would be a monitoring routine checking a program invariant; such a routine needs to be notified anytime any state variable covered by the invariant—possibly through aliases—is changed. In summary, there are four key steps when performing monitoring: (1) extracting the state information in the application needed by the monitor; (2) communicating this state information to the monitor; (3) updating the state in the monitor; and (4) dispatching the set of monitoring routines in response to the state change in a timely fashion.

The simplest way to monitor the execution of an application is to integrate the set of monitoring routines with the application through instrumentation [?,?]. This way, the application state is completely visible to the monitor, thus no explicit state extraction and communication is needed. The monitor maintains its own state in the same address space as the application. In this scenario, dispatching *explicit* monitor routines is straightforward—calls to the appropriate monitor routines can be insert as instrumentation in the original program. To dispatch the appropriate *implicit* monitoring routines, all instructions that can generate a state change of interests must be instrumented. At runtime, all relevant state changes must be examined, and the proper monitor routine dispatched. If instrumented instructions occur often, this can be a source for significant performance overhead.

As an example to illustrate our points, consider a simplified memory bug detection monitor that observes heap accesses and determines whether or not the following rules are violated by an application when accessing the memory: (i) no read to uninitialized memory locations; (ii) only allocated memory can be accessed; (iii) all allocated memory must be freed eventually; (iv) parameters to calls to the `free` function must be allocated memory addresses and be the return values of previous `malloc` function calls. In the case of explicit software instrumentation, all access to heap memory as well as calls to `malloc` and `free` must be instrumented [?,?,?,?]; separate shadow memory must be maintained to keep track of the allocation and initialization history. The performance impact of this implementation is significant, previous work has reported a 20x slowdown, even with aggressive optimizations [?].

One way to mitigate the performance degradation is to migrate the monitor to a separate processor [?,?]. This will allow the monitor and the application to execute in parallel and—if the monitor can keep up with the application—the performance penalties are now limited to the overhead associated with extraction and communication of state information and competition for shared

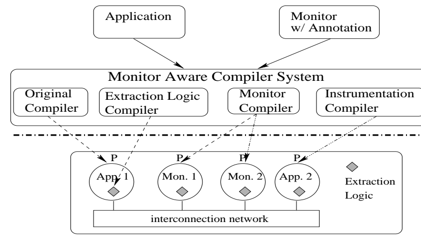
resources. Unfortunately, the state extraction and communication overhead can be significant, and the instrumentation needed to extract the information from the application leads to—for our purposes—unacceptable perturbation of the application.

Recently, there have been proposals for hardware support for a variety of monitoring activities; in particular to support fine-grained monitoring. Nevertheless, most of these proposals support one narrow type of monitor, such as monitoring memory bugs [**?**,**?**] or taint analysis [**?**,**?**,**?**,**?**,**?**,**?**]. These solutions can provide extremely low overhead monitoring, but they are all targeted to specific monitoring tasks and allow little or no customization.

The techniques briefly discussed above (heavy instrumentation, distribution of monitors to separate processors, and hardware supported special purpose monitoring) cover three corners in the design space of software monitoring. When considering a monitoring task, we must consider within this design space the particular need for flexibility, performance, and predictability. For example, when considering continual monitoring for memory access violations the performance overhead is of utmost importance, but the need for flexibility in monitoring is not there (a special purpose monitoring task); dedicated hardware support might be the right solution. When using monitors as oracles during unit testing, the need for flexibility in the monitoring task is imperative since we are likely to use monitors for diverse tasks, for example, a functional test oracle or a test adequacy measurement tool. In this case, arbitrary monitors that can be easily modified and replaced are required; an inflexible hardware solution would not work, flexible monitors based on instrumentation would be far more suitable (if the performance penalty is acceptable). The particular challenges of our target monitoring tasks (discussed in Section 2) require us to somehow achieve the best characteristics from the previously suggested techniques while largely eliminating their weaknesses.

Given the observations on monitoring above, it has become clear that to successfully enable effective monitoring we must pursue an *architecture and compiler enhanced* approach to *software monitoring*. The architectural and compiler support is essential to provide the performance and throughput needed for realistic applications, and all monitors much be software based to allow for the flexibility we need for a diverse collection of monitoring activities. To achieve these breakthroughs, we must re-consider the four steps in monitoring.

Figure 1 shows an overview of our proposed monitoring framework that orchestrates the compiler as well as the architectural support to generate and enhance software execution monitors. As mentioned above, at run time there are four steps in monitoring: (1) extract the desired information from the core executing the application, (2) forward it to the monitor core, (3) update the monitor core state with the new information, and (4) dispatch the appropriate monitoring routine(s). All four steps must be supported with hardware to provide the performance needed. In particular, hardware support for information extraction is essential since our aim is to avoid instrumenting the application program with special instructions. To this effect, we integrate an *extraction logic*



Architectural support for execution monitoring

- Flexible extraction logic extracts information needed by the monitor routines
- Underlying communication mechanism forward information to the monitor
- Effective monitor dispatching support invokes the desired monitor routines.

Compilation support for monitor generation

- Identify the set of activities that invoke the monitor routines
- Generate three outputs: (i) the application executable (instrumented hardware support for extraction logic is unavailable); (ii) the monitor executable; and (iii) extraction logic configuration if hardware support is available.
- Update the monitor as the application is optimized;

Fig. 1. Overview of our proposed hardware/compiler monitoring infrastructure.

onto each core, and have the monitor configure the extraction logic with an event-of-interest list when the program is loaded. This extraction logic is capable of capturing a variety of instruction-level events. For example, the extraction logic can invoke monitors at specific program points; thus, it must be aware of the program counter. If we are monitoring for changes of a variable x and x is a register resident value, the event list would include the set of instructions that modify this value. On the other hand, if x is a memory-resident value, the event list would include the instructions that modify the memory location where x resides. Once such events are detected, information about the event are forwarded to the monitor. A similar approach is employed to monitor for other memory related events as well as for function invocations.

The *monitor-aware compiler* takes as inputs the application and the monitoring routines annotated with the state information in an application A of interest to the monitor and the program points in A where explicit monitor routines will be invoked. From this input, the monitor aware compiler creates three new artifacts: (1) the application executable, (2) the monitor executables, and (3) a list of instruction-level events that must be extracted from the application execution. Note: our goal is that monitored application executable shall contain no additional instrumentation as compared to a non-monitored equivalent.

The monitor executable will contain two parts: a list of *monitor routines* and a dispatching routine that determines which monitor routines should be invoked when we observe a certain event. The monitor-aware compiler is also responsible for generating the list of instruction-level events that must be extracted and

forwarded from the executing application to the monitor; this information will be used to configure the extraction logic for each core.

There is a risk that the on-chip and off-chip shared resources, such as the communication bandwidth, could become an issue even in our framework and the application being monitored might have to be stalled for the extraction logic to forward its information. In addition, should the execution time for the monitor be excessive, the application might have to be stalled for the monitor to keep up. One way to reduce the cost of state communication is to reduce the amount of state information transferred. This can be achieved with both hardware and software support. In this work, we propose compiler analysis to identify the minimal set of state information that require communication, and then only communicate this set. We also propose hardware techniques to avoid the transfer of duplicate state and compress the transferred states. For example, there is no need to transfer state information that has not changed; this is a fairly common occurrence when, for example, a piece of control software runs at a rate 5 to 10 times faster than the sensors sampling the environment—we will only see changes to certain state variables in the control software every 5-10 execution cycles. Furthermore, parallelization of the monitors is also possible in this framework

4 Support for Execution States Extraction

In a multi-core environment, efficiently using one core to monitor the behaviors of the software executing of another requires hardware support. One of the key functionality of such hardware support is to selectively extract information needed by the monitoring core from the monitored core. To avoid stalling the monitored application, information extraction must operate at the speed of the processor. In our previous work, we proposed a programmable *extraction logic* for this purpose, we refer to the architectural support as *Ex-Mon* [**?**]. The *extraction logic* can be programmed with the set of instructions whose runtime behaviors are of interests to the monitor. E.g., the extraction logic can be programmed with a set of instruction addresses; and the results of these instructions will be forwarded. The *extraction logic* monitors the earliest entries of the reorder buffer. When an instruction commits, the *extraction logic* decides whether the results of the instruction is of interest to the monitor; and packs and forwards the necessary information if it is. The monitor software explicitly manages the extraction logic by initiating and updating it at runtime.

Forwarded information is written to a dedicated area of the shared memory, referred to as the *communication queue*. The *communication queue* is a circular buffer, with the head points to the next available slot for the *extraction logic* to write to; and the tail points to the next element to be consumed by the monitor. The head is updated by the extraction logic; and the tail is updated by the monitor. When the queue is full, the monitored program must be stalled, which is a major cause for performance degradation in monitoring.

Although the extraction logic allows programmers to specify events of interests, traffic for some monitor activities can still be excessive. Thus, optimization opportunities to reduce communication traffic must be explored. Consider a monitor that is interested in detecting accesses through dangling pointers. Once a certain memory location is proven to be allocated, all future accesses to this location do not need further verification, until the state of this memory location is changed by system calls to `realloc` and `free`. Consider a loop containing an instruction that sweep through array elements. Once we know the first element accessed, the access pattern is predictable until the end of the loop is reached. To take advantage of these opportinities, we propose hardware and software support to reduce communication traffic.

Eliminating redundant forwarding with hardware support: An auxiliary structure, the *local filter*, is introduced to reduce communication costs. 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 instruction will not be forwarded. The *local filter* is initially empty and is updated by the *extraction logic*, which is in turn managed by the monitoring software. This is achieved by adding two extra bits in the *extraction logic* indicating whether the instruction forwarded should create an entry in the filter, clear the filter or have no effect.

Eliminating redundant forwarding with compiler support: The hardware-based solutions although efficient, is fundamentally limited by the lack of global program information. Compiler can help by only forwarding values that (i) are actually needed by the monitor; and (ii) are difficult for the monitor to derived.

The compiler first identify program state changes that do not affect the verification routines and stop forwarding them. The irrelevant state elimination problem can be formulated as a simple dataflow analysis, where the compiler simply mark all values used in the monitor routines as relevant, and perform a backward analysis to mark all instructions these values depend on. When the algorithm terminates, all unmarked instructions are irrelevant, and should be eliminated.

Communicating values from the monitored core to the monitor can be costly, and is sometimes more expensive than computing the forwarded value locally on the monitor. To identify and explore these opportunities, the compiler must identify a set of communications so that the cost of communication and computation at the monitor is minimized. This optimization can also be formulated as a backward analysis that identified dependent instructions. The backward search terminates when the cost for computing all dependent instructions exceeds the cost of reading from the communication queue.

5 Evaluation with Case Studies

To evaluate the effectiveness and efficiency of architecture and compiler support for execution monitoring, we will illustrate how the Ex-Mon can be utilized to detect memory bugs and track taint propagation.

In the proposed infrastructure, 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 are 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. Currently, the monitor compiler is under development, and the monitor programs used in this paper are developed manually.

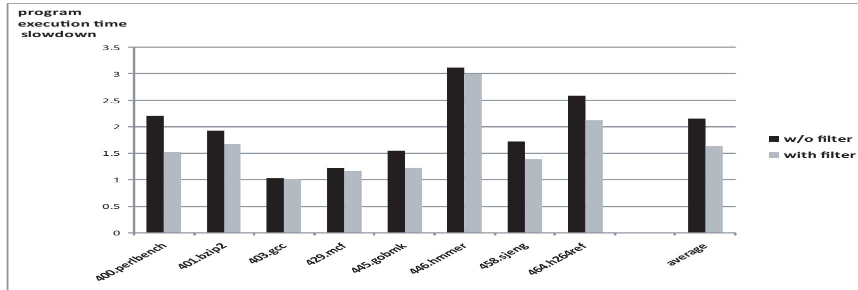
We evaluate the proposed support for software execution monitoring using the Simics [?] simulation environment, a full system simulation platform. We augment the simulator with the Wisconsin GEMS [?] infrastructure for a detailed memory hierarchy simulation. We simulate a multicore system with 8 cores, where 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, with 64Bytes cache lines and 3-cycle access latency. The L2 cache is 8MB in size and 4-way set-associative, with 64Bytes cache lines and 6-cycle access latency. The main memory is 8GB in size with 160-cycle access latency. The extraction table has 1K entries and the local filter has 32 entries. We simulate the SPECINT 2006 benchmarks with the `ref` input set. For a reasonable simulation time, we simulated one billion instructions after skipping the initialization phase of the benchmark.

In this paper, we evaluate the performance overhead of two execution monitor: memory-bug detection and taint propagation, using SPECINT 2006 [?] benchmark suite. We manually developed both monitoring softwares. The memory bug detection monitor detects a set of well-known memory bugs, including double free, memory leak, dangling pointer, and uninitialized load dynamically. The taint propagation monitor tracks flow of information and signals an error if tainted data takes control of program execution. In our implementation, both monitors simply log error states when faults are detected, and reported the faults at the end of the execution.

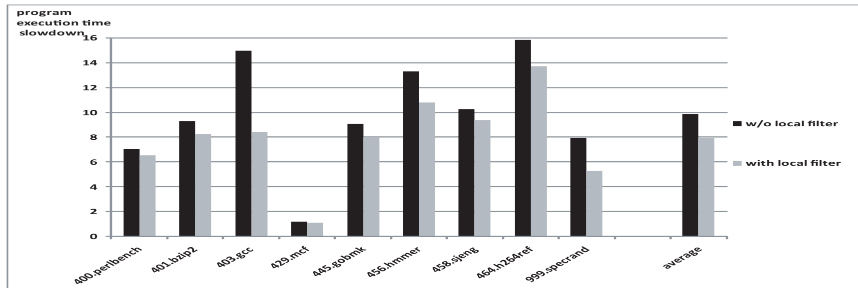
5.1 Verifying Memory Bugs

Memory bugs include memory leak, dangling pointers, loads to uninitialized memory locations and double free. In the rest of this section, we will first provide a brief description of the monitoring software; and then show how the monitor program works with the proposed hardware supports; and finally show the performance of monitoring software with hardware support.

The dispatching routine for memory bug detection 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



(a) Memory bug detection.



(b) Taint propagation.

Fig. 2. Execution time slowdowns due to monitoring for memory bugs and taint propagation.

the termination of the monitored program, will lead to the termination of the dispatching routine.

For memory bugs detection, the forwarded information is the commit of memory access instructions that accesses the heap, and all instructions related to calls of memory management functions, such as `malloc, free, realloc` and `calloc`. The extraction table is thus initiated with instructions that setup and invoke memory management routines; and a memory range that correspond to the heap space.

At runtime, the monitoring software parses incoming events and maintains allocation information for each and every memory block. Based on the allocation information, the monitor is able to verify whether a memory error has occurred. 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.

In memory bug detection, we can use the local filter to reduce communication. In this case, all heap references are entered to the local filter to avoid repeated

forwarding. Commits of call instructions to memory management functions will clear the local filter, since these functions can change the memory allocation.

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, as shown in Figure 2(a). For a 32K communication queue, the performance overhead is 110% on average. However, for some benchmarks, such as 403.GCC, the performance overhead is negligible, while for some other benchmarks, such as 446.HMMER the performance overhead is nearly 200%. Utilizing the local filter to reduce the number of forwarded through the communication queue has a significant performance impact, as shown by the `with local filter` bars in Figure 2(a). With a 32-entry local filter, all benchmarks are able to benefit significantly—on average, we are able to achieve 20% performance improvement.

5.2 Tracking Taint Propagation

Taint propagation is the foundation for building many security-enhancing software monitors. In taint propagation, each data item, i.e., every memory location and register, is tagged with a taint tag indicating whether the value stored is tainted or not. At runtime, the taint tags are propagated by the instructions that manipulate these data. For example, we can mark all data from unsafe sources, such as the internet, as tainted, and then keep track of taint propagation to ensure that unsafe data do not take control of the program.

From the perspective of taint propagation, two types of instructions requires monitoring: data movement instructions (load/store, mov,etc) that propagate taint status of the source operand to its destination; and an arithmetic instruction that taint the destination operant if any of its sources is tainted. Since these two types of instructions contribute to a significant potion of dynamic execution, taint propagation creates a burden on the communication capability between the monitoring core and the monitor. At runtime, the monitor parses incoming events and maintains taint information for each and every memory and register. Based on the allocation information, the monitor is able to keep track of taint information. In taint tracking, we can use the local filter to reduce communication.

Results The workload of the monitor is almost always higher than that of the monitored program, and thus the application stalls often, causing performance degradation. This is the main performance penalty evaluated in our work, as shown in Figure 2(b). With a 32K communication queue, the performance overhead is 8x on average. For some benchmarks, such as 456.HMMER and 464.H264,

the overhead is as high as 10-13x. These benchmarks have larger percentage of data-moving and arithmetic instructions, and thus impose heavy workload on communication and taint tracking. 429.MCF, on the other hand, only incurs an 13% overhead. This is because 429.MCF has low IPC due to cache misses, thus stalls caused by monitoring is relatively small.

Utilizing the local filter to reduce the number of data items forwarded through the communication queue has a significant performance impact, as shown by the `with local filter` bars in Figure 2(b). With a 32-entry local filter, all benchmarks are able to benefit significantly—on average, we are able to achieve 20% performance improvement. For 403.GCC, the performance improvement is as much as 46%.

6 Conclusions

Our central hypothesis is that by leveraging the rapid emergence of multi-core processor architectures, we can achieve a non-intrusive, predictable, fine-grained, and highly flexible general purpose monitoring framework through *monitoring-aware* compilers coupled with *novel architectural enhancements* to the multi-core architectures.

In this paper we describe our initial steps in this direction and provide some preliminary performance results achieved with this new multi-core architecture. We use separate cores for the execution of the application to be monitored and the monitors. We augment each core with identical programmable extraction logic that can observe an application executing on the core as its program state changes. We experimented our hardware support with two well-known monitors: memory bug detection and taint propagation, and found that with adequate hardware support, performance penalty can be reduced significantly compare to instrumentation-based approaches.