

FADE: A Programmable Filtering Accelerator for Instruction-Grain Monitoring

Sotiria Fytraki[†], Evangelos Vlachos[‡], Onur Kocberber[†], Babak Falsafi[†], Boris Grot^{*}

[†]EcoCloud, EPFL

[‡]Oracle Labs

^{*}University of Edinburgh

Abstract

Instruction-grain monitoring is a powerful approach that enables a wide spectrum of bug-finding tools. As existing software approaches incur prohibitive runtime overhead, researchers have focused on hardware support for instruction-grain monitoring. A recurring theme in recent work is the use of hardware-assisted filtering so as to elide costly software analysis.

This work generalizes and extends prior point solutions into a programmable filtering accelerator affording vast flexibility and at-speed event filtering. The pipelined microarchitecture of the accelerator affords a peak filtering rate of one application event per cycle, which suffices to keep up with an aggressive OoO core running the monitored application. A unique feature of the proposed design is the ability to dynamically resolve dependencies between unfilterable events and subsequent events, eliminating data-dependent stalls and maximizing accelerator's performance. Our evaluation results show a monitoring slowdown of just 1.2-1.8x across a diverse set of monitoring tools.

1. Introduction

Software robustness poses a key challenge to application developers as modern systems become increasingly complex, leading to more bug-prone software [22]. Because bugs and security vulnerabilities proliferate, programmer's productivity suffers and security breaches intensify, eventually resulting in catastrophic system failures. Dynamic instruction-grain monitoring is a powerful approach to improve software robustness: by monitoring individual program instructions at runtime [21], dynamic instruction-grain monitoring allows for detection of erroneous behavior, such as bugs [12] and security vulnerabilities [17].

Software dynamic instruction-grain monitoring tools afford high flexibility, but they slow down program execution by up to two orders of magnitude [16]. The high slowdown is due to the numerous monitoring actions taken per application instruction (e.g., perform correctness checks and bookkeeping). As high runtime overhead limits opportunities for deployment, prior work considered trading off flexibility for performance through custom hardware targeting specific monitoring functionality [5, 7, 8, 23]. While both architectural details and targeted monitors vary widely among the various proposals, many tend to employ some form of *filtering* as a task-specific approach for reducing the monitoring load. For instance, HardBound filters out non-pointer application data by accessing and checking the relevant metadata with custom hardware [7]; FlexiTaint employs rule-based filtering to determine whether taint propagation can be performed in dedicated logic avoiding software analysis [23].

The key contribution of this paper is in developing a general filtering accelerator to support a broad range of monitoring tasks with high filtering coverage and low hardware overhead. We generalize and extend earlier observations regarding filterable events by linking them to common application and monitoring activities, such as initializing a stack frame on a function call. We observe that unfilterable events, which require processing by the monitoring software, often contain dependencies on subsequent filterable events, thus lowering filtering efficiency due to stalls. In response, we develop monitor-agnostic architectural extensions that enable concurrent filtering and processing of unfilterable events. We also observe high non-uniformity in the arrival rate of filterable and non-filterable events alike. However, we demonstrate that shallow queues are sufficient to buffer the event bursts.

Using a suite of diverse monitors, we show that:

- The average monitoring load rarely exceeds one event per cycle, indicating that a single-issue filtering accelerator with a throughput of one event per cycle suffices.
- Instruction and stack-update events dominate the monitoring load. Instruction events require fine-grained accesses to monitor's metadata, most of which can be filtered through (1) hardware-executed checks of meta-

[‡]This work was done while the author was at CMU.

^{*}This work was done while the author was at EPFL.

data state against an invariant, and (2) detection and elimination of redundant updates that leave the metadata state unmodified. Stack-update events perform bulk metadata initialization in response to function calls and returns and can be efficiently handled with a simple state machine.

- Maintaining a high filtering rate requires that filtering takes place concurrently with the processing of unfiltered events, a task that is complicated due to dependencies between unfilterable and subsequent filterable events. To decouple filtering and the processing of the unfiltered events, we observe that there is only minimal state that is critical for deciding if a dependent event is filterable. We show that this state can be updated for unfilterable events directly in the accelerator with simple hardware extensions.
- Both filterable and unfilterable events arrive in bursts that must be buffered to reduce stalls due to backpressure. Shallow queues of 16 to 32 events are sufficient for this purpose and allow for decoupling of the filtering accelerator from the core running the application.

Building on the observations above, this paper develops an architecture, along with full microarchitectural support, for a flexible at-speed Filtering Accelerator for Decoupled Event processing, or FADE. Using full-system cycle-accurate simulation, we show that FADE is highly effective, filtering out 84-99% of events that would otherwise be handled in software, thereby reducing the application slowdown to only 1.2-1.8x (versus 1.6-7.4x for unaccelerated execution). In the 40nm technology, FADE requires 0.12mm² of area and 273mW of power at peak.

2. Motivation

Instruction-grain monitoring is a powerful analysis technique with the ability to observe the application actions in fine detail. The monitoring tools observe dynamic application *events* (e.g., instructions, function calls) to identify erroneous, anomalous, and otherwise interesting behaviors. In doing so, *monitors* check whether a predefined program invariant holds. Invariants may specify that every accessed memory location has been allocated, or that the value used as a jump target is not spurious. To assist analysis, monitors maintain bookkeeping information, or *metadata*, about the state of the application memory and registers. Based on the event, the relevant metadata are checked against the invariant and/or updated with a new value.

The powerful analysis enabled through instruction-grain monitoring comes with the downside of high performance overhead. Software-only schemes, such as Valgrind [16], provide flexibility. However, the flexibility comes at a steep performance penalty of up to two orders of magni-

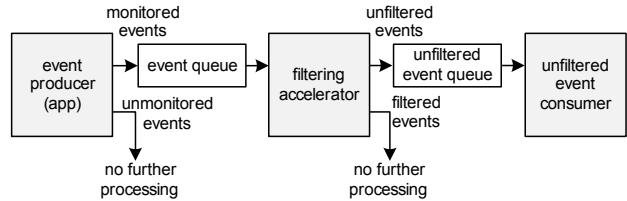


Figure 1. A monitoring system with filtering support.

tude [16], as for each application event, a software handler is dispatched to check and/or update metadata. Several optimizations have been proposed to lower the runtime overhead through analysis-specific optimizations [9], sampling of application activity [10], and hot-path analysis [18], but they either incur considerable slowdown or are not widely applicable.

Filtering of application events that trigger monitoring activity has been proposed as a way to reduce the high runtime overhead. Prior work has identified the potential of filtering and has introduced hardware-based mechanisms to achieve high monitoring performance [2, 7, 19, 23]. However, prior work treats filtering as a trade-off between flexibility and performance. Filtering mechanisms that achieve high efficiency and low runtime overhead are focused on a narrow set of monitoring analyses (e.g., only taint flow analysis [23], or only memory safety analysis [7]). Filtering mechanisms that aim at high flexibility fail to considerably lower the slowdown [2]. Moreover, a number of existing filtering proposals either require intrusive modifications to the core microarchitecture (e.g., a new pipeline stage [23]) or have high resource overheads, needing a dedicated core for the monitoring task [2, 19].

This work makes the observation that filtering does not have to trade flexibility for performance, and can be effective at accelerating a wide range of monitoring tools. Furthermore, filtering can be independent of the underlying system and monitoring architecture while accommodating different design points in terms of the core microarchitecture and the execution substrate for processing of unfilterable events.

3. Design Considerations

Figure 1 shows the main entities involved in the event processing flow of a monitoring system with filtering support. The *application* generates events as instructions retire and enqueues the events of interest (i.e., *monitored events*) in the *event queue*. The rest of the events (i.e., *unmonitored events*) do not require further processing. The *filtering accelerator (FA)* dequeues events from the head of the event queue and checks whether the filtering condition is satisfied. If so, events are *filtered* and no further action is required. As further processing is necessary for

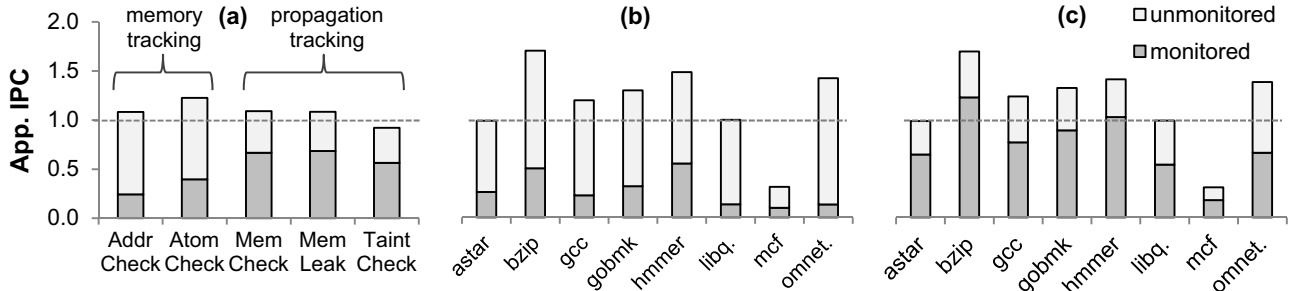


Figure 2. Breakdown of application IPC to monitored and unmonitored: (a) averaged across benchmarks for each monitor, and per-benchmark for (b) AddrCheck and (c) MemLeak.

the rest of the events (i.e., *unfiltered events*), the filtering accelerator places them into the *unfiltered event queue*. Finally, the *unfiltered event consumer* dequeues and handles the unfiltered events completing the monitoring analysis.

3.1. Event Producer

As the application instructions retire, they generate events. However, monitoring analyses do not require all application events to be processed. As a result, software [16] and hardware [2, 6] monitoring frameworks include support to eliminate¹ the unmonitored events. We define *monitoring load* as the ratio of monitored events to all committed instructions.

Based on the types of the monitored instruction events, monitoring analyses can be broadly categorized into two types: *memory tracking*, which process only memory instructions, and *propagation tracking*, which may track any instructions types and propagate a metadata value from the source operand(s) to the destination operand. The exact instruction types being monitored depend on the monitor’s task. For instance, *MemLeak*, which identifies memory leaks [13], monitors instructions that may propagate a pointer value, such as arithmetic and load/store instructions, but eliminates floating-point instructions.

To quantify the load on different monitors, we measure the applications’ monitored IPC on an aggressive 4-way OoO core (we detail the benchmarks and monitors in Section 6). In Figure 2(a), we present the per-monitor results averaged across benchmarks. For instance, for AddrCheck, the average application IPC (including both monitored and unmonitored instructions per cycle) is 1.1, out of which 0.4 (monitored instructions per cycle) require a monitoring action to be taken.

In general, the monitoring load of memory-tracking monitors is lower compared to the monitoring load of

propagation-tracking monitors, because propagation-tracking monitors tend to process more events. As a result, the former have a low monitored IPC (up to 0.4 event per cycle), while the opposite holds for the latter (up to 0.68 event per cycle).

Figure 2(b), shows the per-benchmark results for *AddrCheck*, a memory tracking monitor, which checks whether an access goes to allocated memory [16]. For all benchmarks, the monitored IPC is significantly below 1.0, with an average of 0.24. In contrast, Figure 2(c) shows the per-benchmark results for *MemLeak*, a propagation tracking monitor. While most benchmarks also have a monitored IPC of below 1.0, with an average of 0.68, the monitored IPC of *MemLeak* is 2.8x higher than *AddrCheck*, underscoring the differences in monitoring load.

The monitored IPC indicates the event generation rate of the applications and dictates the rate at which events must be consumed by the filtering accelerator. The presented analysis shows that the monitored IPC is below 1.0 for a range of monitors, even when the event stream is produced by an aggressive OoO core. We thus conclude that *a filtering accelerator with a processing capability of one event per cycle can keep up with the event producer*.

3.2. Event Queue

We next examine the buffering requirements between the event producer and the filtering accelerator. For the purpose of our study, we assume a filtering accelerator that processes one event per cycle and has an infinite event queue. In Figure 3(a, b), we present the cumulative distribution of the event queue’s occupancy for (a) AddrCheck, a memory-tracking monitor, and (b) MemLeak, a propagation-tracking monitor, on an aggressive 4-way OoO core.

For memory-tracking monitors (Figure 3(a)), the monitored IPC is low, resulting in small bursts of events that can be captured in an 8-entry queue. For propagation-tracking monitors (Figure 3(b)), the monitored IPC is considerably higher, resulting in longer bursts. Depending on the benchmark’s monitored IPC, the queuing requirements range from 128 entries (mcf – low monitored IPC) to 8K entries (omnetpp – higher monitored IPC). For

1. The term filtering has been used in prior work [6] to refer to elimination of unmonitored events. We do not use the term filtering in this context because no monitoring task is associated with unmonitored events.

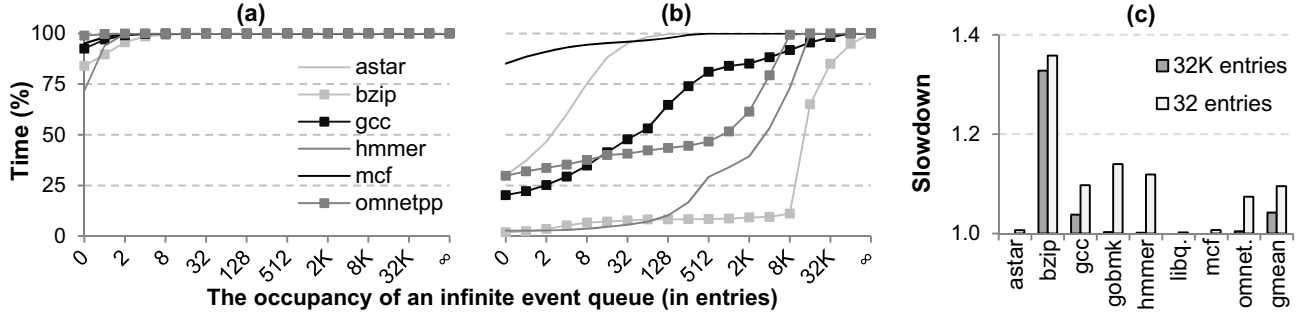


Figure 3. The occupancy of an infinite event queue for (a) AddrCheck and (b) MemLeak. (c) The effect of event queue size on performance for MemLeak.

benchmarks with a monitored IPC greater than one, such as bzip, queueing cannot help, as the filtering rate (1.0 event per cycle) is below the event generation rate (1.2 events per cycle).

We next compare the performance loss stemming from finite queues over an infinite event queue. We evaluate two queue sizes: (1) 32K entries, which can accommodate the bursts based on our analysis, and (2) 32 entries, which is a practical-sized queue. In Figure 3(c), we present results for MemLeak, a monitor that exerts the greatest pressure on the queue due to its high monitored IPC. We observe that the 32K-entry queue can fully accommodate the bursts (resulting in no slowdown) for all benchmarks but bzip and gcc, corroborating the burstiness analysis in Figure 3(b). Meanwhile, a much smaller queue of only 32 entries results in a slowdown that ranges from none (mcf, astar, libq.), to 1.17x (gombk). Queueing cannot help with bzip (monitored IPC over 1.0) resulting in a 1.33x slowdown for a 32K-entry queue and a 1.36x slowdown for a 32-entry queue. For gcc, queueing reduces the slowdown from 1.1x (32-entry queue) to 1.04x (32K-entry queue). We conclude that *a small (e.g., 32-entry) event queue allows for insignificant slowdown caused by bursts.*

3.3. Filtering Accelerator

The filtering accelerator aims at reducing the overhead of common monitoring activities, which mainly happen in response to two categories of application events: (1) instructions, (2) function calls and returns. The monitors also process high-level events (e.g., malloc, fopen, mmap). The filtering accelerator does not target high-level events, as they are infrequent and require complex handling.

The vast majority of monitoring activity is due to instruction events requiring accesses, checks, and updates to the metadata of the instruction operands. Nearly all remaining monitoring activity is due to function calls and returns. At each function call (return), a frame is allocated (deallocated) on the application stack. We refer to both types of activity as a *stack update*. Stack updates must be shadowed by the monitor to properly track which portion

of the application memory has been allocated. Therefore, the monitor sets a region of metadata memory to a known value (e.g., *allocated and uninitialized* on a call, *unallocated* on a return).

Figure 4(a) breaks down the monitors’ execution time into instruction (classified into RU and CC, explained later) and stack-update handling. While instructions dominate the execution profile, in two out of five studied monitors *stack updates consume up to 17% of the execution time and represent an attractive acceleration target.*

3.4. Unfiltered Event Queue and Consumer

Events that cannot be handled by the filtering accelerator (i.e., unfiltered events) require further processing by the monitoring system. An ideal unfiltered event consumer should be able to support a wide variety of monitoring tools for comprehensive bug coverage. This requirement argues for a programmable substrate, such as a general-purpose core (e.g., LBA [2]) or a reconfigurable fabric [6].

Nearly all unfiltered events arise as a result of (1) memory allocation, deallocation, or initialization; and (2) traversals of tainted data structures or files in taint-tracking monitors. In general, these actions involve multiple memory words and, as a result, trigger a burst of metadata updates that cannot be filtered.

Figure 4(b) plots the distance, as a cumulative distribution, between unfiltered events for MemLeak. Results are similar for other monitors. The distance is measured in events. We observe that two unfiltered events are typically separated by up to 16 filterable events. Based on this analysis, we define an *unfiltered burst* as a sequence of unfiltered events, each of which is separated by at most 16 filterable events. Figure 4(c) shows the average burst size (measured in unfiltered events) for each monitor and benchmark pair. We observe that the bursts are small, with an average size of 16 or fewer unfiltered events for the majority of benchmarks and monitors. We thus conclude that *a small (e.g., 16-entry) unfiltered event queue is effective at accommodating the bursts.*

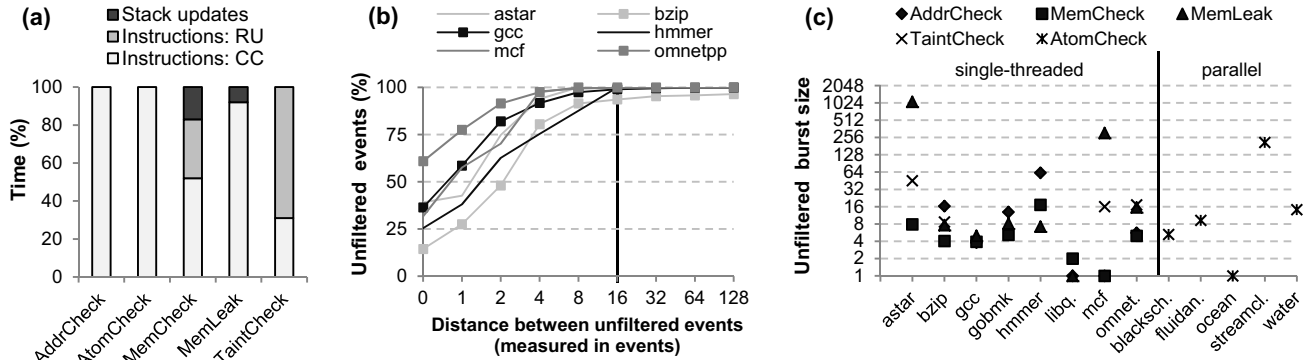


Figure 4. (a) Time breakdown in instruction-grain monitoring. (b) Cumulative distribution of distances between unfiltered events for MemLeak. (c) Unfiltered burst size for all monitors and benchmarks.

An important implication of our analysis is that because filterable events are interleaved between pairs of unfilterable event, it is essential to perform filtering concurrently with the processing of unfiltered events. However, inter-event dependencies mandate in-order event processing, forcing a naïve filtering accelerator design to stall when an unfiltered event is processed by the unfiltered event consumer.

3.5. Summary

Our study of a broad range of applications and monitors shows that the monitoring load rarely exceeds one event per cycle even with an aggressive OoO core producing events. While instructions dominate the event stream, stack updates also contribute to the monitoring load. Event production is bursty, mandating queuing for pending events; however, a small queue is sufficient for good performance. Unfiltered events are also bursty and are sparsely spaced within an otherwise filterable event stream.

These results point to a programmable filtering accelerator able to keep up with an average monitoring load of one event per cycle, capable of filtering concurrently with unfiltered event processing, and loosely coupled through shallow queues to both application and monitoring systems. The next two sections present our design for such an accelerator. We first present a design that does not support filtering concurrent with the processing of unfiltered events (Section 4), and then extend it to support *Non-Blocking Filtering* (Section 5).

4. Baseline Filtering Accelerator

We introduce our baseline Filtering Accelerator for Decoupled Event processing, or FADE. FADE is composed of two building blocks: (1) the *Filtering Unit*, which filters instruction events (Section 4.1), and (2) the *Stack-Update Unit*, which accelerates stack-update events (Section 4.2). Without loss of generality, we assume that

unfiltered events are processed in software on a general-purpose core.

4.1. Filtering Unit

To elide software execution, the Filtering Unit supports two filtering actions, *clean checks (CC)* and *redundant updates (RU)*. Clean checks are based on the observation that most of the time applications behave as expected and the metadata match the expected invariant (e.g., memory references are to initialized memory). Redundant updates are based on the observation that metadata are stable as propagation handlers commonly update the metadata with the same value (e.g., initialized memory remains initialized even when the actual value in application memory changes). Figure 4(a) breaks down the execution time of instruction events into clean checks and redundant updates.

The Filtering Unit handles an instruction event either as a clean check or as a redundant update. To maximize flexibility and applicability, the Filtering Unit implements three modes of operation: (1) *Single-shot filtering* either performs a clean check or identifies a redundant update, (2) *Multi-shot filtering* chains multiple single checks together to determine whether an event is filterable, (3) *Partial filtering* filters a part of the software handler functionality in hardware, thus reducing the handler’s length.

FADE’s hardware is fully programmable and allows for per-event definition of the filtering rules. In FADE, programmability is achieved by configuring two structures: (1) the *event table*, which includes per-event filtering rules, and (2) the Invariant Register File (*INV RF*), which keeps invariant values related to the monitoring task (e.g., *unallocated*, *allocated*, and *initialized* states for Mem-Check). These structures are memory-mapped and programmed on a per-application basis.

Figure 5 shows the baseline filtering pipeline, which consists of four stages. Note that striped structures, including the *Metadata Write* stage, are only for Non-Blocking Filtering as discussed in Section 5. The pipeline works as

(a)		(b)																								
field	bits	eventID/ next entry			valid			mem			MD bytes			mask			CC	INV id			RU	Non-Block.	MS	next entry	P	handler
event ID	6	s1	s2	d	s1	s2	d	s1	s2	d	s1	s2	d	s1	s2	d		s1	s2	d		INV id	Op			PC
src1 reg	5	1	1	1	1	1	1	1	1	1	0xff	0xff	1	2	2											PC ₁
src2 reg	5	1	1	1	1	1	1	1	1	1	0xff	0xff				1										PC ₃
dest reg	5	1			1			1			0x80		1	2									1	A		
app addr	32																									
app PC	32	A	1		1			1			0x7f		1	3												PC ₂

Figure 6. (a) Event entry format. (b) Event table entries. The size of an event table entry is 96 bits.

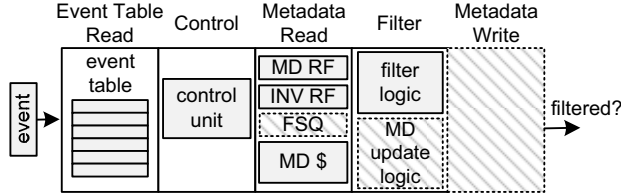


Figure 5. Filtering unit pipeline. Striped structures show pipeline extensions for Non-Blocking Filtering.

follows. First, the filtering rules are read from the event table. Next, the *control unit* uses the event information and the filtering rules to produce the control signals for subsequent stages. Then, the Filtering Unit accesses the metadata register file (*MD RF*) and a dedicated metadata cache (*MD cache*) to obtain metadata. The Filtering Unit may also access the *INV RF* to obtain monitor-specific invariants, if necessary. Finally, in the *Filter* stage, the *filter logic* checks whether the filtering condition is satisfied.

Stage 1: Event Table Read. The filtering accelerator dequeues an event (Figure 6(a)) from the event queue and accesses the event table with the event ID to obtain the event’s filtering rules. An event table entry (Figure 6(b)) includes the following information for each operand (i.e., *s1*, *s2* and *d*): (1) the *valid* bit and the *mem* bit to denote the evaluated operands and the memory operands, respectively; (2) the number of *MD bytes* to be evaluated; (3) a *mask* to extract the appropriate bits. Each entry also includes the *PC* of the software *handler* to be invoked for unfiltered events.

Each entry includes the *CC* bit and the *INV id* for clean checks, and the *RU* field for redundant updates. The *INV id* indicates the invariant registers (one for each operand) to be used upon a clean check. The *RU* field encodes three options. In case of one source operand, the source metadata are directly compared to the destination metadata. In case of two source operands, the source metadata are composed using either OR or AND and then compared to the destination metadata. The rest of the fields are described later.

Stage 2: Control. The control unit processes the information obtained from the event table and uses combinational logic to generate control signals for subse-

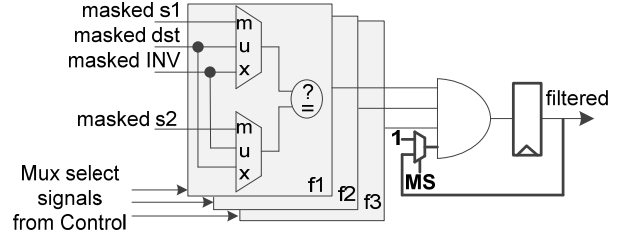


Figure 7. The internals of filter logic.

quent stages (e.g., filter logic mux controls, selects and enables for *MD RF*).

Stage 3: Metadata Read. The Filtering Unit accesses the *MD RF*, the *INV RF* and the *MD cache*, to obtain metadata and invariants values. As application and monitor processes use different address spaces (a desirable feature that enhances system security and reliability), metadata accesses necessitate a translation from the application to the monitor address space. We fold the address translation into the *MD cache* access. The TLB of the *MD cache*, similar to M-TLB [2], contains the translation from a virtual application page to the physical page that contains the associated memory metadata. The M-TLB misses are serviced in software.

Stage 4: Filter. The Filtering Unit supports three modes of operation to filter events.

Single-shot Filtering. In a single cycle, the Filtering Unit compares up to three distinct operand metadata to an invariant (clean check), or compares the operand metadata to each other (redundant update).

Examples of single-shot filtering are shown in the first two entries of Figure 6(b). The first event table entry corresponds to a load instruction for MemLeak. FADE handles the event as a clean check ($CC=1$) and filters the event when both operands are not pointers. In doing so, the metadata of the event operands (i.e., the memory operand *s1* and the register operand *d*) are compared to the *non-pointer* invariant, which is stored in the third entry of the *INV RF* ($INV id=2$). The evaluated metadata are one byte ($MD bytes=1$). The second event table entry corresponds to a load instruction that is handled as a redundant update.

Figure 7 details the *filter logic*, which is organized as three identical two-operand comparison blocks (labeled

f1, f2, and f3 in the figure). Each block can compare any one of three event operands (i.e., s1, s2, and d) to another operand or to an invariant. Together, the three blocks allow for a single-cycle evaluation of the most complex single-shot condition (i.e., comparing each of the three operands – s1, s2, and d – to a different invariant).

Multi-shot Filtering. To accommodate complex monitors that require multiple checks to determine whether an event is filterable, FADE supports multi-shot filtering. The Filtering Unit processes multi-shot events in multiple cycles by performing one check per cycle, and maintains one entry in the event table per check, thus keeping each entry simple. To encode multi-check events, each event table entry requires two additional fields (shown in Figure 6(b)): (1) the *next entry* field, which contains a pointer to the next entry in the event table; and (2) the *multi-shot bit* (MS), which enables multiple checks to be chained by allowing the previous filtering outcome to be considered in the final filtering outcome. As shown in Figure 7, the associated circuit (in bold) includes a clocked register and a multiplexer, controlled by the MS bit.

Partial Filtering. Partial filtering affords a part of the handler functionality to be executed in hardware, reducing the length of the software handler. A software handler may first perform a check and based on the check’s outcome, executes either an update or a more complex routine including multiple checks and updates. FADE accelerates such cases by performing the initial check in hardware. To support partial filtering, each event table entry includes a *partial* bit (*P*) (shown in Figure 6(b)), which drives the selection of the handler PC.

An example of partial filtering appears in AtomCheck, where the filter logic checks whether a shared memory location was last referenced by the same thread. Commonly, the check succeeds, and a simple software handler is dispatched to update metadata. Otherwise, a complex handler runs to check whether there is a potential atomicity violation. While both cases require software execution, the hardware check eliminates the code associated with the check itself, control flow, and register spills and fills.

4.2. Stack-Update Unit

Stack-update events, which set consecutive metadata addresses to a predefined value in response to function calls and returns, are handled in FADE via a dedicated Stack-Update Unit (SUU). The SUU implements a finite state machine that takes the stack frame’s starting address and length as parameters to calculate the address(es) of the metadata block(s) covered by the stack frame. The SUU issues writes to the MD cache to set the target range of addresses to one of two predefined values (one value on function calls and another on function returns), which are stored in the INV RF.

5. Non-Blocking FADE

5.1. Observations

Due to true dependencies between monitored instructions, baseline FADE must stall filtering when an unfiltered event is encountered. Filtering resumes when the monitoring system completes the unfiltered event processing and the updated metadata become available. This organization penalizes performance because filtering and execution of unfiltered event handlers cannot overlap.

To overcome the serial processing of unfiltered events and subsequent dependent events, we make a critical observation: while monitors often maintain detailed metadata to support complex monitoring analyses, there is a subset of metadata, which we call *critical*, that includes sufficient information to decide whether a subsequent dependent event is filterable. Importantly, this critical state can be updated for unfilterable events directly in hardware in the Filtering Unit. These updates are non-speculative and are based on predefined rules that can be implemented in simple hardware.

For instance, for MemLeak, which performs reference counting to identify memory leaks, an event is filterable when its operands are not pointers. Therefore, just checking the *pointer/non-pointer* status of a memory location or a register suffices to make the filtering decision. For example, in case of a load instruction, if the source memory location has a pointer status, the destination register obtains a pointer status as well. However, to perform reference counting, MemLeak maintains additional metadata per register and memory location, which consist of a pointer to the context (explained in Section 6) of the corresponding malloc. While fundamental to MemLeak’s monitoring algorithm, these additional metadata are non-critical from the perspective of the filtering task.

Overall, we observe that (1) there is critical (minimal) state that can be checked to determine the filtering outcome in a non-speculative way, and (2) this state can be updated in simple hardware based on simple pre-defined rules. Based on these observations, the filtering decision and the handling of unfiltered event can be decoupled, thus enabling the design of a *Non-Blocking* filtering unit that can continue filtering past an unfiltered event.

5.2. Extensions to the Baseline Pipeline

Figure 5 shows the pipeline extensions (striped) to support Non-Blocking Filtering. We introduce two new structures; the *metadata (MD) update logic*, which performs updates to the filtering-critical metadata for unfilterable events, and the *filter store queue (FSQ)*, which stores the updated memory metadata. We also introduce a new pipeline stage, *Metadata Write*, where updates to metadata take place.

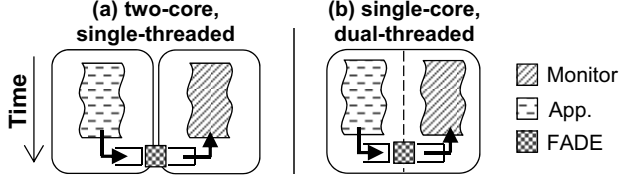


Figure 8. Evaluated systems.

Processing of instruction events. Consider an unfilterable event that just enters the pipeline. The processing in the first three stages (Event Table Read, Control, and Metadata Read) is the same as in the baseline pipeline. In the Filter Stage, while the filtering condition is evaluated, the MD update logic computes the new value for the filtering-critical metadata. The new metadata value is subsequently used only if the filtering condition evaluates to false, indicating an unfilterable event. Otherwise, the new metadata value is discarded.

To determine the logic for critical metadata updates, we observed that critical metadata have minimal state and their propagation follows simple rules. Based on the studied monitors, we provide support for the following rules: (1) propagating the source metadata (s1 or s2) to the destination; (2) composing the new destination metadata from the two source metadata using OR or AND; (3) setting the destination metadata to a constant value, which is stored in an INV register denoted by the *Non-Blocking/INV id* field in the event table (see Figure 6(b)); and (4) conditionally performing one of the above actions after comparing the source operands to each other, to the destination, or to a constant.

In the *Metadata Write* stage, the Filter Unit commits updated metadata to the MD RF (for register) or to the FSQ (for memory). Subsequent events with a true dependence on the updated metadata can then obtain them from the MD RF or the FSQ in Metadata Read stage. For memory metadata, the FSQ is searched in parallel with the MD cache. If a matching FSQ entry is found, it is used to satisfy the dependence; otherwise the metadata from the cache are used. To accommodate back-to-back dependencies, forwarding from the Metadata Write stage to the Filter stage is supported.

Eventually, the unfiltered event handler executes and updates both the critical and the non-critical metadata for registers and memory. Once the handler completes, the MD cache contains the updated value for the critical memory metadata (if any) and the corresponding FSQ entry is discarded. Subsequent accesses to these metadata are served by the MD cache.

Processing of stack-update events. As stack updates change the metadata state, filtering must stop upon a stack-update event to allow the SUU to set the stack frame metadata. Moreover, as pending unfiltered events may ref-

Table 1. Simulation setup.

Parameter	Value
Core type	in-order 1-way
	lean OoO 2-way/48-entry ROB
ISA	aggr. OoO 4-way/96-entry ROB
	SPARC v9
L1 Caches	32KB, 2-way, 64B block 2-cycle latency
Shared L2	2MB, 16-way, 64B block, 10-cycle latency
DRAM	90-cycle latency

erence stack frame-related metadata, the unfiltered event queue must be drained by the consumer prior to stack-update processing.

6. Methodology

Evaluated designs. We evaluate two FADE-enabled systems, shown in Figure 8. The *two-core monitoring system* (Figure 8(a)) executes the application and monitor threads on separate cores to maximize concurrency [2]. Filtering takes place next to the monitor core. The *single-core monitoring system* (Figure 8(b)) is based on a fine-grained, dual-threaded core with a dedicated hardware thread for the application and monitor processes. This design point minimizes resource requirements, but results in higher slowdown because the core resources are shared between the application and monitor.

We also evaluate two unaccelerated systems, similar to the single- and two-core systems presented in Figure 8 but without FADE. In these systems, the application and the monitor communicate through a single queue.

System configuration. Table 1 summarizes the configuration of the evaluated systems. Additionally, FADE-enabled systems have a 4KB, two-way MD cache with one-cycle access latency, and a 16-entry Metadata TLB. A sensitivity analysis for these two structures (excluded due to space limitations) shows that this design point offers the best cost-performance ratio. The event table has 128 entries, covering the heavily used subset of the modeled ISA (SPARC). The event queue and the unfiltered event queue is 32 and 16 entries, respectively. Unless, otherwise specified, experiments use Non-Blocking FADE.

Simulation. We use Flexus [26] for cycle-accurate full-system simulation. Flexus extends Simics with timing models of multithreaded cores, caches, and interconnect. For our evaluation, we use the SMARTS sampling methodology [27]. Our samples are drawn over one billion instructions of the monitored application. As our benchmarks are organized as a collection of loops, we sample over an execution interval that covers multiple iterations. For the parallel benchmarks, we follow the same approach to cover a representative part of the benchmark’s parallel

section. For each measurement, we launch simulations from checkpoints with warmed caches (including the MD cache), and run 100K cycles to achieve a steady state of detailed cycle-accurate simulation before collecting measurements for the next 50K cycles.

Power and Area. To estimate FADE’s area and power, we synthesize our VHDL implementation with Synopsys Design Compiler. We use TSMC 45nm technology (core library: TCBN45GSBWP, V_{dd} : 0.9V) scaled down to 40nm half node, and target a 2GHz clock frequency. For the MD cache, we estimate area, power, and latency with Cacti 6.5 [14].

Monitors. To demonstrate FADE’s generality, we use a suite of five diverse monitors that cover a range of memory, security, and concurrency bugs.

AddrCheck [16] checks whether memory accesses are to an allocated region. The critical metadata encode two states (*allocated or unallocated*) per memory location, while the non-critical metadata include book-keeping information for bug reporting. FADE filters accesses to allocated data through clean checks.

MemCheck [16] extends *AddrCheck* to detect the use of uninitialized values, and *TaintCheck* [17] detects overwrite-related security exploits. For critical metadata, *MemCheck* has three metadata states (i.e., *unallocated*, *uninitialized*, and *initialized*) and *TaintCheck* has two metadata states (i.e., *untainted* and *tainted*). Non-critical metadata may include information related to origin tracking [1] or other bookkeeping information. FADE performs clean checks for legitimate accesses and filters redundant updates when metadata remain unchanged.

MemLeak [13] identifies memory leaks through reference counting. The critical metadata consist of the *pointer/non-pointer* status of each register and memory word. Non-critical metadata consist of a pointer to the corresponding malloc’s context and a null value otherwise. The context includes a unique ID, PC, and a reference counter. FADE performs clean checks to filter events with non-pointer operand values.

AtomCheck [12] detects atomicity violations by checking access interleavings. For this purpose, it keeps track of the last access by each thread to each application memory location. *AtomCheck* maintains one byte of critical metadata per application word with the *thread status bit* and the *thread id*. Furthermore, it keeps non-critical metadata including the type (Read/Write) of the last access by each thread in local per-thread tables. *AtomCheck* is accommodated by Partial filtering, as explained in Section 4.1.

Benchmarks. For all monitors, except *AtomCheck*, we use the SPEC2006 integer benchmarks with reference inputs. These CPU-intensive benchmarks stress the monitoring system with a high event generation rate. For *TaintCheck*, we use the benchmarks (astar, bzip, mcf,

Table 2. FADE’s filtering efficiency.

AddrCheck	AtomCheck	MemCheck	MemLeak	TaintCheck
99.5%	85.5%	98.0%	87.0%	84.0%

omnetpp) that have tainting propagation and we exclude the rest. For *AtomCheck*, we use five multithreaded benchmarks: water and ocean from the SPLASH suite; and blackscholes, streamcluster, and fluidanimate from the PARSEC suite. Each benchmark has four threads that run on one core in a time-sliced manner. All benchmarks use 32-bit binaries.

7. Evaluation

7.1. Filtering Efficiency

Table 2 shows that FADE filters 84-99% of all instruction event handlers. *AddrCheck* has the highest filtering ratio because nearly all instruction events can be filtered via clean checks as the applications access allocated memory. In contrast, *TaintCheck* has the lowest filtering ratio of 84%, as it performs value propagation that results in long propagation chains with a higher frequency of metadata updates.

7.2. FADE versus Unaccelerated System

Figure 9 depicts the performance of FADE versus the unaccelerated monitoring system. In both systems, application and monitor tasks execute in dedicated hardware threads of a dual-threaded 4-way OoO core. Performance is normalized to an unmonitored (application-only) system.

In general, for the unaccelerated systems, we observe an average slowdown of 4.1x, across monitors. For memory-tracking monitors (*AddrCheck*, *AtomCheck*), the average slowdown is 2.5x, while for propagation-tracking monitors (*MemCheck*, *MemLeak*, *TaintCheck*), the slowdown is 5.8x. FADE reduces the slowdown significantly for all monitors, with an average slowdown of 1.5x. FADE’s slowdown is 1.3x and 1.6x for memory- and propagation-tracking monitors, respectively.

Figure 9(a) shows *AddrCheck*’s performance, which is generally good on both systems as the monitor just processes non-stack memory instructions. In the unaccelerated system, *AddrCheck*’s slowdown ranges from 1.2x to 2.9x, with an average of 1.6x. FADE reduces the slowdown to an average of 1.2x by filtering out nearly all monitored events.

Figure 9(b) shows the results for *MemLeak*, a heavy-weight propagation-tracking monitor. In the unaccelerated system, we observe slowdown ranging from 3.4 to 11.5x, with an average of 7.4x. We note that the benchmarks with a high *monitored* IPC (e.g., 1.2 for bzip) generate events faster than those with a low *monitored* IPC (e.g., 0.2 for

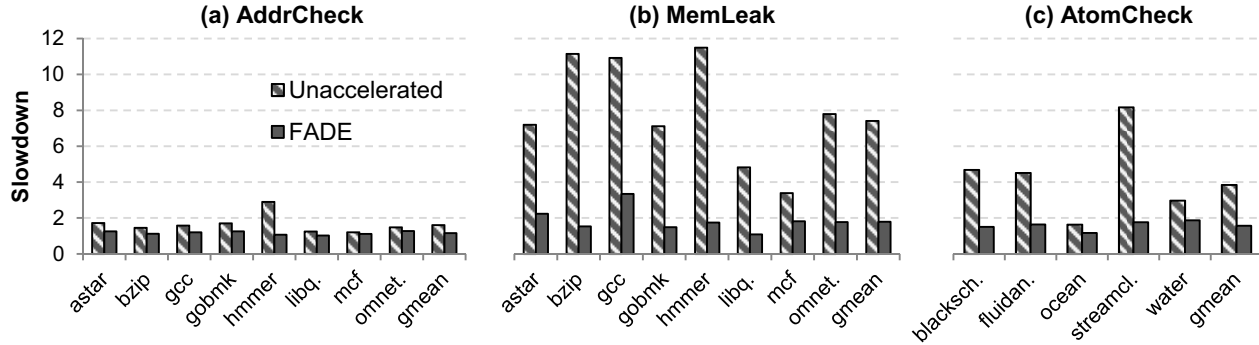


Figure 9. Performance of FADE compared to the unaccelerated system.

mcf), resulting in higher slowdown due to the increased pressure on the monitor. FADE significantly reduces the slowdown to an average of 1.8x, thanks to its high filtering ratio and the hardware-accelerated stack-update unit. The highest slowdown is observed on *astar* (2.2x) and *gcc* (3.3x), which are characterized by a low filtering ratio (70%) and must frequently drain the unfiltered event queue at function call/return boundaries (Section 5.2).

Figure 9(c) presents results for *AtomCheck*. Although *AtomCheck* is a memory-tracking monitor with a low event generation rate, it has an average slowdown of 3.9x (8.2x max) in the unaccelerated system because the events are costly due to numerous monitoring actions. In contrast, FADE benefits from a high filtering ratio, resulting in an average slowdown of 1.6x (1.9x max).

Finally, FADE reduces the slowdown to an average of 1.4x for *MemCheck* (similar to *MemLeak*) and 1.6x for *TaintCheck* (similar to *AtomCheck*). Detailed results for these monitors are omitted due to space limitations. Across the five evaluated monitors, FADE reduces the monitoring slowdown to an average of 1.5x, versus 4.1x for the unaccelerated system.

7.3. Performance for Different Core Types

To better understand the effects of core microarchitecture on monitoring performance, we evaluate the unaccelerated and FADE-enabled systems with different core types. Figure 10 summarizes the performance for three core microarchitectures; in-order, 2-way OoO, and 4-way OoO averaged across all benchmarks.

For the unaccelerated monitoring systems (dashed bars), we observe a reduction in performance ranging from 7% to 51% for simpler core microarchitectures as compared to the 4-way design. Although the applications generate up to 2x fewer events per cycle on the in-order core than on the 4-way OoO core, each event handler executes up to 3x faster on 4-way OoO because event handlers consist of instruction sequences with high cache locality, resulting in high IPC on aggressive cores. Thus,

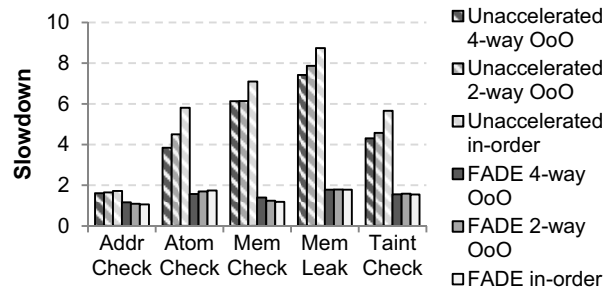


Figure 10. Performance of the single-core monitoring system.

we conclude that monitors are sensitive to the core microarchitecture.

In the FADE-enabled system (solid bars), performance is less dependent on the core type. For example, *MemCheck* performs marginally better on the simple microarchitecture (average slowdown of 1.2x on in-order versus 1.4x on 4-way OoO), showing that filtering leaves little work for the monitor core and the core microarchitecture is less important.

7.4. Single-Core versus Two-Core System

Prior work [2, 25] has suggested utilizing otherwise idle cores to accelerate the monitoring task. We next evaluate this design point in the context of FADE.

Figure 11(a) compares the performance of single-core (dual-threaded) and two-core monitoring systems. Both are FADE-enabled and feature a 4-way OoO microarchitecture. The results indicate that the two-core design outperforms the single-core option by 15% on average (28% max) by eliminating resource contention between monitor and application threads. As the second core is expected to provide a theoretical speed-up of 2x over the single core, we investigate the reason for the limited benefit of the second core.

Figure 11(b) breaks down the utilization of the two-core system into three categories: cycles in which (1) the application core is idle because the event queue is full, (2) the monitor core is idle because FADE filters all events,

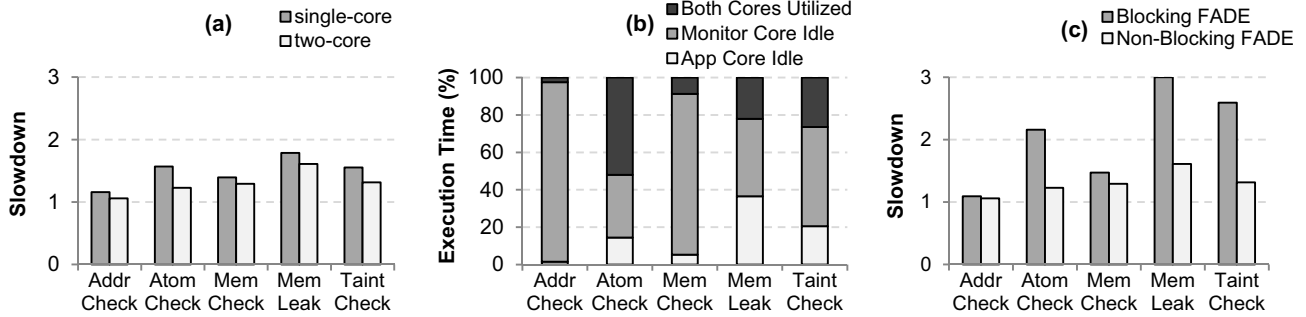


Figure 11. (a) Performance of the single- versus two-core monitoring systems with FADE. (b) Core utilization in the two-core system. (c) Performance benefits of Non-Blocking Filtering.

and (3) both application and monitor cores are utilized. As the figure shows, 48% to 97% of the time, one of the two cores is idle, as FADE either filters the incoming event stream (idling the monitor core), or the monitor core processes unfiltered events (backpressuring the application core). With both cores utilized only 22% of the time, on average, the benefit of the second core is clearly limited.

7.5. Benefits of Non-Blocking Filtering

To show the benefit provided by Non-Blocking Filtering, Figure 11(c) compares the performance of Non-Blocking FADE (used in the studies above) to the baseline FADE that stalls on each unfiltered event.

We observe that Non-Blocking Filtering improves the performance by 2x for AtomCheck, MemLeak and TaintCheck, which have relatively low filtering ratios (<87%), and by 1.1x for AddrCheck and MemCheck, whose filtering ratio is high (>98%). The benefit of Non-Blocking FADE comes from overlapping the filtering actions with the unfiltered events processing.

7.6. Area and Energy Efficiency

To model FADE’s area and power costs, we synthesized our RTL design in TSMC 40nm technology. Our design includes a 128-entry event table, a 32-entry event queue, and a 16-entry unfiltered event queue. Synthesis results show an area of 0.09mm² and a peak power consumption of 122mW. To estimate the area and power requirements of the 4KB MD cache, we use CACTI. We find the area cost of the cache to be 0.03mm², peak power of 151mW, and an access latency of 0.3ns.

8. Related Work

Prior work proposed hardware support for instruction-grain monitoring. Early proposals sacrifice flexibility by hardwiring the monitoring policy [4, 21]. Other proposals [5, 23, 24] allow for a number of different monitoring policies but their pipelines can only accommodate fixed-size metadata. Monitor-specific proposals include support for

race detection [8], and spatial memory safety for C/C++ programs [7, 15]. DISE [3] instruments the instruction stream on-the-fly by injecting the instrumentation code directly into the pipeline. Another class of tools is based on watchpoints [11, 20, 28]; however, it cannot support certain monitors (e.g., propagation trackers) [28], and can degrade performance for certain metadata layouts [11].

Event filtering has been considered in prior work as a way to accelerate monitoring; however, prior proposals only considered filtering for a narrow range of behaviors by (1) targeting only a specific monitor [7, 19], (2) supporting only metadata of specific size [23], or (3) sacrificing bug coverage [2]. This work advances the state-of-the-art by (1) providing generalized support for filtering including partial and multi-shot filtering, (2) accelerating bulk metadata management (i.e., stack updates), and (3) proposing Non-Blocking Filtering.

9. Conclusions

This work introduced FADE, a Filtering Accelerator for Decoupled Event monitoring. The proposed design exploits common behavior across monitors to provide simple, programmable hardware for handling common application events while delegating infrequent complex events to software for maximum flexibility. To maximize throughput and avoid stalls in the presence of unfiltered events, FADE employs Non-Blocking Filtering — a hardware-assisted mechanism for concurrent processing of filterable and unfiltered events. Our results showed that FADE can filter 84-99% of application events across a range of monitors, resulting in an average slowdown of only 1.2-1.8x, thereby making instruction-grain monitoring practical.

10. Acknowledgements

We would like to thank Stavros Volos, Djordje Jevdjic, Cansu Kaynak, Almutaz Adileh, Javier Picorel, and the anonymous reviewers for their insightful feedback on earlier drafts of the paper. This work was partially supported

by Swiss National Science Foundation, Project No. 200021_140551/1.

References

- [1] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: Reporting the origin of null and undefined value errors. In *Conference on Object-oriented Programming Systems and Applications*, 2007.
- [2] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *International Symposium on Computer Architecture*, 2008.
- [3] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A programmable macro engine for customizing applications. In *International Symposium on Computer Architecture*, 2003.
- [4] J. Crandall and F. Chong. Minos: Control data attack prevention orthogonal to memory model. In *International Symposium on Microarchitecture*, 2004.
- [5] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *International Symposium on Computer Architecture*, 2007.
- [6] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *International Symposium on Microarchitecture*, 2010.
- [7] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.
- [8] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. RADISH: Always-on sound and complete race detection in software and hardware. In *International Symposium on Computer Architecture*, 2012.
- [9] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Conference on Programming Language Design and Implementation*, 2009.
- [10] J. L. Greathouse, I. Wagner, D. A. Ramos, G. Bhatnagar, T. Austin, V. Bertacco, and S. Pettie. Testudo: Heavyweight security analysis via statistical sampling. In *International Symposium on Microarchitecture*, 2008.
- [11] J. L. Greathouse, H. Xin, Y. Luo, and T. Austin. A case for unlimited watchpoints. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.
- [12] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.
- [13] J. Maebe, M. Ronsse, and K. D. Bosschere. Precise detection of memory leaks. In *International Workshop on Dynamic Analysis*, 2004.
- [14] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *International Symposium on Microarchitecture*, 2007.
- [15] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *International Symposium on Computer Architecture*, 2012.
- [16] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Conference on Programming Language Design and Implementation*, 2007.
- [17] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security*, 2005.
- [18] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing dynamic information flow tracking. In *Symposium on Parallelism in Algorithms and Architectures*, 2008.
- [19] 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 of Research and Development*, 50(2), 2006.
- [20] A. Shriraman and S. Dwarkadas. Sentry: Light-weight auxiliary memory access control. In *International Symposium on Computer Architecture*, 2010.
- [21] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.
- [22] The MITRE Corporation. Common vulnerabilities and exposures (CVE). <http://cve.mitre.org>.
- [23] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *International Symposium On High Performance Computer Architecture*, 2008.
- [24] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. MemTracker: Efficient and programmable support for memory access monitoring and debugging. In *International Symposium On High Performance Computer Architecture*, 2007.
- [25] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry. ParaLog: Enabling and accelerating online parallel monitoring of multithreaded applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.
- [26] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4), 2006.
- [27] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *International Symposium on Computer Architecture*, 2003.
- [28] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architectural support for software debugging. In *International Symposium on Computer Architecture*, 2004.