EPFL-REPORT-187154

BugSifter: A Generalized Accelerator for Flexible Instruction-Grain Monitoring

Sotiria Fytraki[†], Onur Kocberber[†], Evangelos Vlachos[‡†], Jennifer B. Sartor[*], Boris Grot[†], Babak Falsafi[†]

[†]PARSA, EPFL [‡]CALCM, CMU [*]Ghent University

Abstract

Software robustness is an ever-challenging problem in the face of today's evolving software and hardware that has undergone recent shifts. Instruction-grain monitoring is a powerful approach for improved software robustness that affords comprehensive runtime coverage for a wide spectrum of bugs and security exploits. Unfortunately, existing instruction-grain monitoring frameworks, such as dynamic binary instrumentation, are either prohibitively expensive (slowing down applications by an order of magnitude or more) or offer limited coverage.

This work introduces BugSifter, a new design that drastically decreases monitoring overhead without sacrificing flexibility or bug coverage. The main overhead of instruction-grain monitoring lies in execution of software event handlers to monitor nearly every application instruction to check for bugs. BugSifter identifies common monitoring activities that result in redundant monitoring actions, and filters them using general, light-weight hardware, eliminating the majority of costly software event handlers. Our proposed design filters 80-98% of events while monitoring for a variety of commonly-occurring bugs, delegating the rest to flexible software handlers. BugSifter significantly reduces the overhead of instruction-grain monitoring to an average of 40% over unmonitored application time. BugSifter makes instruction-grain monitoring practical, enabling efficient and timely detection of a wide range of bugs, thus making software more robust.

1 Introduction

Software robustness is a key challenge for application developers because recent shifts in hardware design have led to more complex software [11]. Bugs not only decrease productivity, but also introduce vulnerabilities that can lead to security breaches and catastrophic system failures [2]. Dynamic instruction-grain monitoring [46] is a promising approach to find bugs during application execution by monitoring individual instructions. Flexible instruction-grain monitoring techniques [10, 13, 45, 58] combine the flexibility of software [12, 20, 36, 37, 38, 39, 41, 42] with the low execution overhead of hardware [14, 15, 25, 49, 50, 57, 59].

Unfortunately, in spite of architectural support (e.g., running the program and the monitor on separate cores), state-of-the-art monitoring approaches that support the full flexibility of software [10] slow down execution by an order of magnitude on average. Because monitoring occurs at the granularity of instructions, the monitor performs a number of actions depending on its functionality (e.g., check for null pointers, check for synchronized memory sharing, or perform bookkeeping) and as such incurs high software overhead. Instead, prior work advocates trading off flexibility for performance through custom accelerators targeting specific monitor functionality [9, 15, 44, 50].

Instruction-grain monitors rely on maintaining metadata per word of memory to implement their functionality. In this paper, we make the observation that in the common case, the monitor activity requires either minimal processing (e.g., most operand checks to identify pointers do not find one, or checks for exclusive access rights on local data always succeed) or generalized metadata management (e.g., allocating metadata on the stack upon function calls). Common case metadata activity requiring no action can be found in a diverse set of monitors, ranging from memory access checking (when accessing allocated data) [36, 37] to reference-based garbage collection and memory leak detection (when the operand of an instruction is not a pointer) [4, 32]. For other monitors, common case activity requires a minimal set of actions, as is the case in race [43] and atomicity violation [29] detectors (when accessing thread-local data).

We propose BugSifter, architectural support for metadata caching, lookup, and update that “sifts” metadata in hardware to identify when and what software action is needed, and otherwise eliminates software monitoring overhead. We use cycle-accurate simulation of single- and multi-threaded benchmarks to show that:

- BugSifter is generally applicable, accelerating bug finding for a variety of memory, security, and concurrency bugs;
- By executing metadata checks in simple hardware and eliding redundant metadata updates, BugSifter filters out 80-98% of all software handlers for instruction events;
- Stack updates on function calls and returns account for up to 40% of the monitors’ execution time. A programmable functional unit that performs stack updates can virtually eliminate this overhead;
- A BugSifter design with a 4KB metadata cache enables a state-of-the-art flexible monitoring system, LBA [10], to achieve an average slowdown of only 40%, obviating the need for monitor-specific accelerators.

The rest of the paper is organized as follows: Section 2 presents the background on software bugs and instruction-grain monitoring. In Section 3, we present our observations that motivate the BugSifter design, while Section 4 details the proposed design. Section 5 describes the range of monitors efficiently accelerated by BugSifter and spectrum of bugs that these monitors cover. Section 6 and 7 feature our methodology and experimental results, respectively. Section 8 discusses the related work, and Section 9 concludes the study.

2 Background

2.1 Bugs in Software

Developing effective bug-finding tools first requires an understanding of types of bugs found in existing software. To that end, researchers have analyzed and classified bugs in a variety of applications, concluding that a broad spectrum of bugs falls within three dominant bug categories: semantic, memory, and concurrency [26, 28]. These categories can be summarized as follows:

Semantic: Bugs that arise in code that is inconsistent with the design specification or programmer's intent, examples being missing features and improper exception handling. Due to the dependance on semantic information, this bug category is not well accommodated by automated bug-finding tools.

Memory: Bugs caused by incorrect handling of memory, such as memory leaks, accesses to uninitialized memory, accesses to memory that has been freed (dangling pointers), null pointer dereferences, and double frees. In addition, the bulk of non-semantic **security**-related bugs, which enable attackers to exploit program design flaws to compromise systems (e.g., through buffer overflow and stack smashing attacks), can be classified as a special class of memory bugs [26].

Concurrency: Bugs that occur as a result of interaction of multiple threads. Lu et al. identify three major classes of concurrency bugs, namely (1) data races, (2) atomicity violations, and (3) deadlock-causing bugs [28]. An important property of concurrency bugs is their non-deterministic behavior, which complicates reproducibility and debugging.

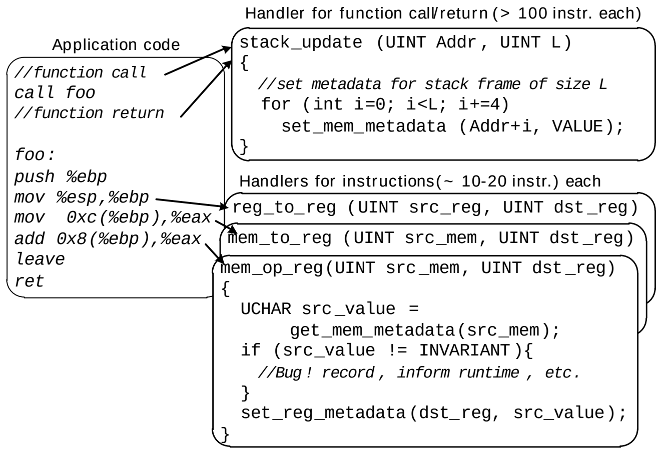
2.2 Instruction-Grain Monitoring

A number of tools have been developed to assist developers in finding bugs. These can be grouped into three general categories [9]: (1) static tools [6, 19, 21] that try to identify problems before the program runs; (2) post-mortem tools [35, 54, 55] that seek to establish the cause of an error after the program has crashed; and (3) dynamic tools [5, 31, 38] that monitor the program at runtime to identify and/or contain the bugs.

While the three categories of tools can be considered complementary, dynamic tools with the ability to monitor at the granularity of individual instructions possess a unique advantage stemming from their access to detailed runtime events, such as memory references and information flow. This capability affords a wide range of powerful bug-finding tools, generally referred to as *monitors*, that span the spectrum from frequently-occurring memory bugs to hard-to-reproduce concurrency bugs. In addition to facilitating bug finding at development time, instruction-grain monitors may be useful in the field by enabling on-the-fly recovery from errors, reducing susceptibility to security exploits, and improving damage confinement.

In general, instruction-grain monitors work by maintaining certain *invariants* and checking that these invariants hold for each application *event of interest*. Invariants might specify that every accessed memory location has been allocated and initialized, or that the value used as a jump target is not tainted. Events of interest may include instructions, memory accesses, function calls and returns, as well as high-level operations (e.g., malloc and free). To assist analysis, monitors maintain bookkeeping information, or *metadata*, about application memory and registers. Depending on the event, the relevant metadata are checked against the invariant and/or updated with a new value.

A code snippet for a representative monitor, along with a slice of monitored application code, is shown in Figure 1. The monitor performs propagation-based analysis used by a number of bug-finding tools (e.g., MemCheck, which checks whether every referenced memory location has been initialized). In the example, each application instruction triggers a software handler associated with the monitor. For each of the instruction's source operands that are in memory, the handler accesses and checks the metadata. If the metadata value differs from the invariant (e.g., a referenced memory location has not been allocated or initialized), an action is taken to inform the user and/or the runtime. The handler also updates the metadata for destination operands based on the metadata state of the source operands.


Application code
Handler for function call/return (> 100 instr. each)
Handlers for instructions(~ 10-20 instr.) each

```
//function call
call foo
//function return

foo:
push %ebp
mov %esp,%ebp
mov  0xc(%ebp),%eax
add 0x8(%ebp),%eax
leave
ret
```

```
stack_update (UINT Addr, UINT L)
{
  //set metadata for stack frame of size L
  for (int i=0; i<L; i+=4)
    set_mem_metadata (Addr+i, VALUE);
}
```

```
reg_to_reg (UINT src_reg, UINT dst_reg)
mem_to_reg (UINT src_mem, UINT dst_reg)
mem_op_reg(UINT src_mem, UINT dst_reg)
{
  UCHAR src_value =
       get_mem_metadata(src_mem);
  if (src_value != INVARIANT){
    //Bug! record, inform runtime, etc.
  }
  set_reg_metadata(dst_reg, src_value);
}
```

Figure 1. A simple instruction-grain monitor

2.3 Existing Approaches to Instruction-Grain Monitoring

A number of projects have targeted effective instruction-grain monitoring. Broadly speaking, the approaches can be classified as software-only or hardware-assisted. Here, we briefly summarize the chief attributes of these schemes; a comprehensive discussion of existing techniques is provided in Section 8.

Software-only schemes, such as Valgrind [38], rely on dynamic binary instrumentation and provide full generality in terms of the types of monitors they support. However, this flexibility comes at a steep performance penalty of 10-100x [36, 38, 48], since for each application event, a software handler is dispatched to check and/or update metadata. Researchers have proposed monitoring tools that leverage algorithm-specific optimizations to reduce the performance overhead, yet even in those cases the slowdown is substantial (e.g, 8x for a software race detector [20]).

To mitigate the performance bottleneck, researchers have investigated hardware-assisted solutions. The most general hardware-assisted scheme is LBA [10], which supports unmodified applications and provides hardware acceleration for dispatching software handlers, thus obviating the instrumentation overhead. However, the actual monitoring functionality is not accelerated and is fully executed in software. Hardware-assisted schemes are either customized for a specific monitoring algorithm (e.g., [18, 22]), or support a limited range of monitors (e.g., [9, 50]). In addition, hardware-assisted monitors commonly modify the application so as to allocate their metadata in the application's address space, necessitating extra actions to ensure that metadata are protected (i.e., from a potential security attack or an overwrite by the application code).

2.4 Summary

Instruction-grain monitoring holds the promise of enabling highly effective on-the-fly bug finding for a broad class of bugs, including hard-to-reproduce concurrency bugs and malicious security exploits. Today's approaches to instruction-grain monitoring force a compromise between performance and generality. Meanwhile, increasing software complexity demands (a) full generality to cover a broad range of bugs and support a variety of algorithms within a common bug-finding framework, and (b) good performance to enable continuous monitoring in development, and ideally in the field, to maximize coverage, resilience, and security.

3 BugSifter Motivation

BugSifter targets fully generalized monitoring with low runtime overhead. To motivate the BugSifter design, we first identify

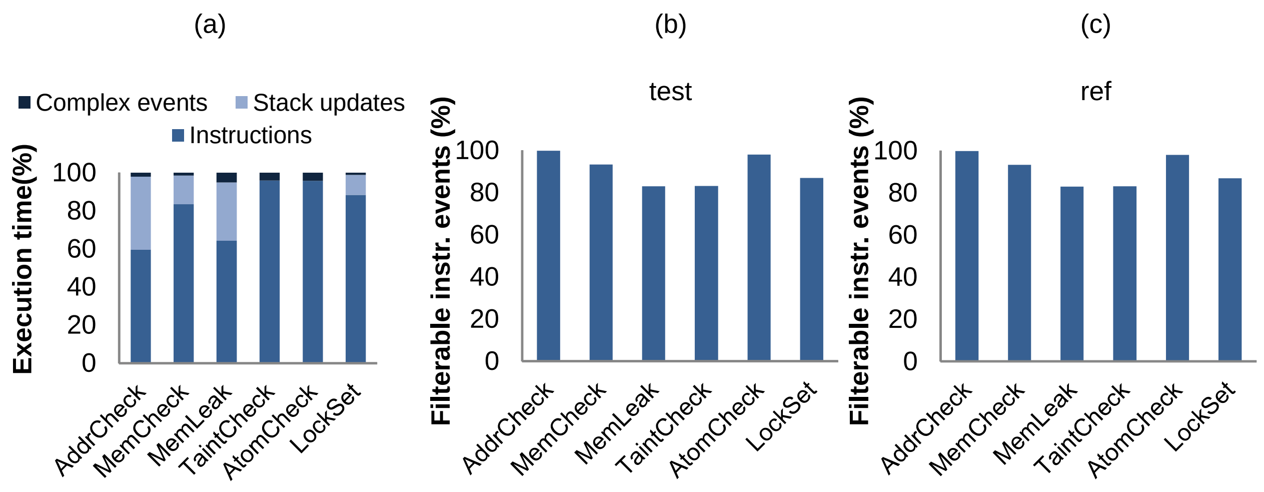


Figure 2. (a) The sources of instruction-grain monitoring slowdown for instruction and stack-update events. The opportunity to filter instruction events for (b) *test* input (c) *ref* input

common high-level functionality inherent in a wide range of monitors. Next, we provide the intuition for how simple and fully general mechanisms can eliminate much of the run-time overhead incurred through commonly occurring monitoring activities.

3.1 Generalized Monitor Functionality

The existence of different bug types dictates that monitors should be specialized for each particular type of a bug. Moreover, for a given bug type, several bug-finding algorithms may exist that differ in their coverage guarantees, resource requirements, implementation complexity, etc. Despite the resulting diversity of bug-finding tools and algorithms, we find that virtually all monitors have functionally-similar characteristics at a high level. These can be summarized as follows, with Figure 1 serving as an illustrative example:

- **Simple checks and updates:** The bulk of monitoring activity in response to individual application instructions involves some combination of metadata accesses, metadata checks against an invariant, and metadata updates. As most instructions in ISAs of contemporary general-purpose processors operate on one or two source operands and update one destination operand, the per-instruction handlers typically manipulate three small pieces of metadata or less, with each metadata item associated with a given application register or memory location.

- **Stack updates:** Software engineering practices call for abstraction and encapsulation of functionality, leading to software with many short functions and frequent function invocations at execution time. 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 what memory has been allocated to an application. As a result, each function call and return event in the application triggers a handler in the monitor that sets a region of metadata memory to a known value (e.g., *allocated+uninitialized* upon a call, *unallocated* upon a return).

- **Complex or uncommon functionality:** Occasionally, monitors invoke functionality that is different from the two cases above. This happens whenever the application executes an uncommon instruction (e.g., containing more than two source operands), performs a high-level event of interest (e.g., malloc or free), initializes the metadata, or when an invariant check fails on the monitor side.

Figure 2(a) breaks down the monitors' execution time into (i) simple metadata checks and updates for applications' instructions, (ii) metadata stack updates in response to applications' stack frame allocations and deallocations, and (iii) all other activities. The six monitors presented in the figure cover the majority of non-semantic bugs described in Section 2.1. Details related to monitor functionality can be found in Section 5.1. Our workloads, described in Section 6, consist of SPEC2000 integer benchmarks using the full test input and 10 billion instructions of the reference input; for LockSet and AtomCheck, we use a set of multithreaded benchmarks.

As the figure shows, nearly all of the monitors' execution time is spent on processing simple instructions and stack updates. Complex instructions and uncommon events together account for less than 5% of the run time for all monitors. Monitoring of simple instructions dominates the execution profile; however, stack updates consume over 30% of the execution time in two out of six monitors due to a large number of instructions (over 100, on average) committed by the stack update handlers iterating through a memory region.

3.2 Accelerating the Common Events

BugSifter accelerates monitor execution through light-weight hardware support for the two common application event types: simple instructions and stack updates. BugSifter also facilitates rapid access to metadata for all monitoring activities, both accelerated and software-handled. Here, we explain the intuition behind the accelerators and leave the detailed description of the BugSifter design for Section 4.

Accelerating instruction events: BugSifter uses a simple event filtering mechanism that relies on two observations to elide the execution of costly software handlers. First, most of the time applications behave correctly and the metadata match the expected invariant (e.g. memory accesses reference memory that has been allocated and initialized). We refer to these events as *clean checks*, which are performed in hardware, avoiding handler execution in case the check succeeds. Second, propagation event handlers that copy metadata values from source to destination operands commonly update the metadata with the same value, because metadata are stable (e.g., memory that has been initialized remains initialized while the actual value in application memory may change). We call these events *redundant updates* and filter the associated handlers as they do not affect the metadata state.

Figures 2(b) and 2(c) show the percentage of instruction event handlers that can be elided as they fall into either the clean check

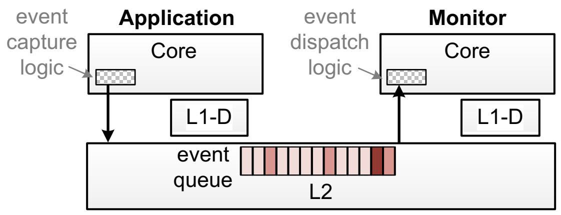


Figure 3. Baseline system

or redundant update category. For AddrCheck, we can filter almost all instruction events due to clean checks, because applications access allocated memory. BugSifter filters 87% of instruction events for MemCheck due to both clean checks and redundant updates. The filtering rate for MemCheck is lower than that for AddrCheck due to metadata initialization. MemLeak has a filtering rate of 80% because most of the applications data are not pointers. TaintCheck has a filtering rate of 80% because metadata are stable and clean. As most application data are accessed by a single-thread, BugSifter filters 87% of events for LockSet. AtomCheck partially filters 98% of instruction events due to clean checks (we explain partial filtering in Section 4.2.1). Overall, we show that 80-98% of instruction event handler dispatch can be eliminated, thereby virtually eliminating monitoring overhead.

Accelerating stack update events: Stack update events, namely function calls and returns, contribute up to 40% of the monitoring execution time, as shown in Figure 2(a). These handlers set a large range of metadata to a predefined value and do not check for bugs directly. BugSifter efficiently accommodates stack update events through a dedicated hardware unit that performs multi-block writes, thereby eliding the associated event handler dispatch which would unnecessarily add more overhead. According to our profiling results, most stack update events (> 97%) can be handled in hardware, virtually eliminating the overhead of the respective software handlers.

Accelerating metadata accesses: Nearly all monitoring activities involve accesses to the metadata. We accelerate accesses to metadata through a dedicated metadata cache, similar to prior proposals [49, 50].

4 BugSifter Design

We detail the BugSifter design in the context of a generic monitoring framework by first presenting the baseline system organization, followed by the set of extensions necessary to enable BugSifter's functionality.

4.1 Baseline System

Our baseline design is shown in Figure 3. While the application is executing, the *event capture logic* creates an event stream queue, or log, from the committed instruction stream, such as in Log-Based Architectures (LBA) [10]. The event queue is memory-mapped into the L2 cache, and does not require dedicated storage. Each entry of the event stream consists of information such as the event category, program counter, input/output operand identifiers and data addresses, and function arguments. Event stream entries are compressed to an average size of less than one byte [10].

A separate monitor thread removes and processes the events from the head of the event queue. Each event triggers an appropriate software handler to be executed by the monitor. Event handler dispatch is assisted by light-weight logic that communicates with the core to dispatch the correct event handler using a jump table of function pointers [10].

The application (producer) and monitor (consumer) threads are not synchronized and the communication between them is restricted to the event queue. To contain the effects of bugs and prevent the propagation of errors beyond the application level, the OS stalls the application thread at system call boundaries, allowing the monitor to catch up with the application thread by processing all queued events.

While Figure 3 shows the application and monitor threads running on separate cores, neither the baseline system nor the proposed BugSifter extensions are restricted to this design point. Nonetheless, in order to limit the scope of the design and evaluation, the two-core organization is assumed from here on and analysis of a single-core system is left as future work. Similarly, both the design and evaluation are performed in the context of the IA32 ISA, which is neither a limitation nor an advantage.

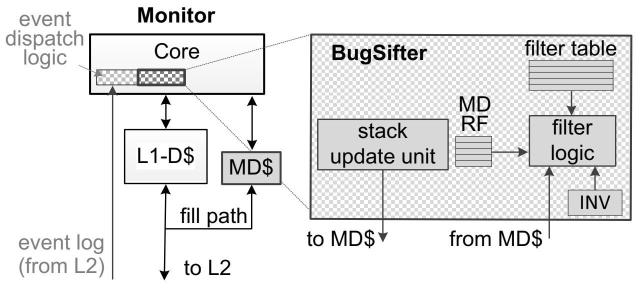


Figure 4. BugSifter design

4.2 BugSifter Architecture

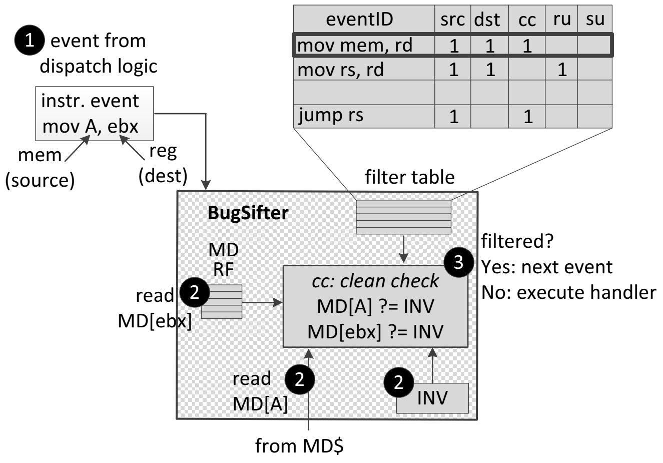
Figure 4 shows BugSifter extensions to the baseline design described in Section 4.1. To enable filtering of instruction events, BugSifter uses a configurable *filter table* and light-weight *filter logic*. For accelerating stack updates, BugSifter introduces a *stack update unit*. Finally, to speed up all aspects of monitor execution, including the filtering, stack updates, and software handlers, BugSifter allows for fast metadata access via a *metadata cache* and metadata register file (*metadata RF*). We next explain how these simple hardware extensions support the common behavior of both applications and monitors to accelerate bug finding while maintaining generality and flexibility on the monitoring side.

4.2.1 Handling instruction events

When BugSifter receives an event, it accesses the filter table to check if the event is filterable. The table has an entry for each filterable event type that indicates the appropriate filtering action: clean check, redundant update, or stack update. In the evaluated design, the table has 30 entries that cover the heavily-used subset of the IA32 ISA. Because in IA32 most instructions have up to two operands (register or memory), events that have more than two operands (for source and destination combined) are classified as uncommon and handled in software.

To facilitate filtering, BugSifter uses a small set of programmable registers that store values of monitor-specific invariants as well as the metadata factor, defined as the ratio between the application word size (e.g., four bytes) and the monitor metadata item size (e.g., one byte). The filter logic uses the metadata factor for accessing and selecting the correct words in the metadata cache.

Figure 5 shows the steps in the filtering process. For clean checks and redundant updates, BugSifter reads metadata from the metadata cache and metadata RF based on the instruction source and destination operands (steps 1 and 2). To enable one-cycle metadata access, the metadata RF has two ports and the metadata cache has one port. For instructions with two memory operands, the metadata cache is accessed twice over two consecutive cycles.



eventID	src	dst	cc	ru	su
mov mem, rd	1	1	1		
mov rs, rd	1	1		1	
jump rs	1		1		

Figure 5. BugSifter filtering example. *MD* stands for metadata, *INV* stands for invariant, and *cc*, *ru*, and *su* stand for clean check, redundant update, and stack update, respectively

Once the operand metadata is available, BugSifter evaluates the filter condition in the filter logic (step 3). For clean checks, the logic compares the operands to the invariant value (read in step 2). For redundant updates, the logic compares the source and destination operand metadata, and filters the event if their values match.

Events that cannot be filtered are sent to the event handler dispatch. These events include uncommon instruction events (<5% of the total dynamic instructions), events that update the metadata (e.g., on initialization), and high-level events, such as malloc and free, which contribute less than 3% to the monitor execution time. Stack update events are handled by a dedicated hardware unit described below.

In some cases, an instruction event first performs a check and based on the check's outcome, it executes either a sole update or a more complex routine that includes multiple checks and updates. BugSifter efficiently supports such cases by filtering one of the actions in hardware (e.g., the first check) and consequently dispatching a simplified software handler. For instance, in the AtomCheck monitor, BugSifter uses the filter logic to check if a memory location was last referenced by the same thread. If the check succeeds, which is the common case, a simple software handler is dispatched to update the metadata; if the check fails, a complex handler runs to check whether there is a potential atomicity violation. While both cases require software execution, the hardware check eliminates more than 2/3 of the instructions executed by a software-only handler for the common case. The eliminated instructions include the code to perform the check, control flow instructions, and register spills and fills. We refer to this type of filtering as *partial* filtering and present more use cases of it in Section 5.2.

4.2.2 Handling stack-update events

Stack update handlers (i.e., function calls and returns) set consecutive metadata addresses to a predefined value. As Figure 2(a) shows, four out of the six studied monitors handle stack updates, which constitute up to 40% of the monitor execution time. Unlike all prior proposals for hardware-supported monitoring systems that ignored stack updates, BugSifter accelerates the majority of stack update handlers.

BugSifter introduces a configurable stack update unit to accelerate the execution of stack update events. The stack update unit takes the stack frame's starting address (*esp*, in IA32) and length as parameters. Because the metadata cache is indexed by the application virtual address, the stack update unit can use the stack pointer (*esp*) to directly calculate the address(es) of the metadata block(s) covered by the stack frame.

Using a simple finite state machine (FSM), the stack update unit issues writes to the metadata cache to set the target range of addresses to one of two predefined values (one value on function calls and another on function returns). The stack update unit can use either the block-wide or sub-block interface of the metadata cache to minimize the number of write operations; this interface is similar to that in contemporary processors that support write combining to reduce write activity in L1-D. The number of metadata cache block writes is dictated by the length of the stack frame, encoded in the event descriptor.

In addition to performing bulk writes, the stack update unit is general enough to handle simple stack-updating instructions, such as push and pop in IA32. For both range and individual stack address updates, BugSifter handles stack events in hardware by (1) configuring the filter table to recognize them, (2) forwarding them directly to the stack update unit, and (3) executing them using the range update FSM, thus completely eliding software intervention.

In order to minimize hardware complexity, stack events that update a region whose size exceeds that of the metadata cache are not filtered and are handled in software instead. Whenever such an event is encountered, the dispatched handler first invalidates the contents of the metadata cache (a performance optimization), and then processes the stack update event in software. Stack frames whose size exceeds the metadata cache size are rare (<3%). Moreover, such large frames typically correspond to long-running functions, which amortize the cost of the software stack update handler.

4.2.3 Metadata cache

BugSifter provides low-latency metadata lookups and updates through a dedicated cache, which accelerates metadata accesses for all monitoring activities [49, 50]. In principal, a dedicated cache for metadata is not required, and the metadata could be cached in the L1-D. However, we find that a small dedicated cache improves filtering performance by enabling single-cycle lookups and reduces the pressure on the L1-D. The filtering hardware accesses the metadata cache directly. To enable the software handlers to leverage the metadata cache, we extend the ISA with new Load Metadata and Store Metadata instructions. Conventional loads and stores access the L1-D unaffected.

4.2.4 Monitoring of parallel applications

BugSifter's single core/process approach is orthogonal to parallel monitoring techniques and platforms. If application threads are time-sliced on the same core, only one BugSifter instance is sufficient. In monitoring systems where the application threads run on different cores [24, 52], each core would need one BugSifter instance. In this case, the metadata caches should be maintained coherent. A simple way to support coherence, given BugSifter's virtually-addressed metadata caches, is to make L1-D inclusive to the metadata cache and maintain backpointers in L1-D [7].

Prior work [24, 52] has also proposed solutions to ensure metadata atomicity and enable parallel monitoring under different memory ordering models. The proposed mechanisms are applicable to BugSifter; however, parallel monitoring is outside the scope of this paper.

4.3 BugSifter Design Summary

BugSifter uses simple, configurable hardware mechanisms to efficiently accelerate a range of monitors by (1) filtering instruction events, (2) eliminating stack update overheads, and (3) minimizing delays on metadata accesses across all monitoring activities. BugSifter keeps hardware complexity low through the use of a compact 4KB metadata cache, a metadata RF with eight 32-bit registers, a configurable filter table with 0.2KB of state, and

Table 1. Monitor functionality and how BugSifter filters instruction events (i.e., clean checks and redundant updates). M stands for metadata of memory/registers. The events are based on IA32. Each enumerated monitor functionality on the left maps to the corresponding enumerated BugSifter action on the right

		Monitor	BugSifter
Memory bugs	AddrCheck	*Purpose: Check whether every memory access goes to an allocated region of memory* 1)The metadata are checked for unallocated memory accesses	*Filterable action: checks* 1) BugSifter action: clean check Example: `mov mem(saddr), %rd` filtered if (M[saddr] == *allocated*)
	MemCheck	*Purpose: Extend AddrCheck to detect the use of uninitialized values* 1) Metadata values are propagated through instructions, such as `mov` 2) An error occurs when uninitialized data are used in critical ways, such as pointer dereference [37, 38] 3) After an error, the operands are set to *initialized* to avoid error cascading	*Filterable actions: checks & updates* 1) BugSifter action: redundant update Example: `mov %rs, %rd` filtered if (M[rs] == M[rd]) 2) BugSifter action: clean check Example: `mov mem(saddr), %rd` filtered if (M[rd] == M[saddr] == *initialized*) 3) BugSifter action: event handler dispatch
	MemLeak	*Purpose: Perform reference counting to identify memory leaks* 1) Metadata values are propagated through instructions, such as `mov`. 2) When propagating a pointer or overwriting one with Non-Pointer value, MemLeak updates a reference counter if the destination pointer does not reside on the stack [32]	*Filterable actions: checks & update* 1a) BugSifter action: clean check Example: `mov mem(saddr), %rd` filtered if (M[rd] == M[saddr] == non-pointer) 1b) BugSifter action: redundant update Example: `mov %rs, %rd` filtered if (M[rs] == M[rd]) 2) BugSifter action: event handler dispatch
Security bugs	TaintCheck	*Purpose: Detect overwrite-related security exploits (e.g., format string vulnerabilities)* We configure BugSifter according to the policies in [39] 1) Metadata values are propagated through instructions, such as `mov` 2) An error occurs when tainted data are used in critical ways, such as jump target 3) Certain instructions need to be handled in a special way	*Filterable actions: checks & updates* 1) BugSifter action: redundant update Example: `mov %rs, %rd` filtered if (M[rs] == M[rd]) 2) BugSifter action: clean check Example: `jne %rs (indirect jump)` filtered if (M[rs] == *untainted*) 3) BugSifter action: event handler dispatch. Example: xor an operand with itself resets the operand to untainted
Concurrency bugs	AtomCheck	*Purpose: Identify atomicity violations based on the interleavings of accesses by different threads* 1) Upon a memory access, AtomCheck first checks if the current and the previous access have been performed by the same thread (accessing a global table) True: just update the thread-local metadata table False: obtain the necessary information from the thread-local metadata tables, and check the interleaving of the last three memory accesses to a memory location	*Partially filterable actions: checks* 1) BugSifter action: clean check (i.e., partially filter an accesses to a memory location that was last referenced by the same thread) Example: `mov mem(saddr), %rd` filtered if (M[saddr] == *current-thread*) True: execute a simple handler in software False: execute a complex handler in software
	LockSet	*Purpose: Detect data races by checking whether accesses to shared memory locations are protected by consistent sets of locks* 1) Upon a memory access, the memory metadata are checked to detect if the memory location is protected by a consistent set of locks	*Filterable actions: checks* 1) BugSifter action: clean check (i.e., filter accesses to thread-local data) Example: `mov mem(saddr), %rd` filtered if (M[saddr] == *thread-local*)

some trivial logic. Finally, BugSifter maintains full flexibility through software event handlers for dealing with rare instructions and infrequent events.

5 Monitors

To demonstrate BugSifter's generality, we use a suite of six diverse monitors that target a broad spectrum of bugs. We also discuss a number of other bug-finding tools that can be effectively accelerated by BugSifter. Together, these tools cover the vast majority of non-semantic bugs described in Sections 2.1.

5.1 Targeted Monitors

We currently use six monitors in our evaluation of BugSifter to demonstrate bug finding across a wide range of erroneous application behavior. These monitors effectively cover a number of memory, security, and concurrency bugs described in Sections 2.1. The monitors are as follows: (i) AddrCheck checks whether every memory access is to an allocated region of memory [36]; (ii) MemCheck extends AddrCheck to detect the use of uninitialized values [37, 38]; (iii) MemLeak observes the use of allocated memory and identifies memory leaks through reference counting [32]; (iv) TaintCheck detects overwrite-related security exploits (e.g., due to format string vulnerability) [39]; (v) AtomCheck detects atomicity violation by checking access interleavings [29]; and (v) LockSet detects data races by checking whether accesses to shared memory locations are protected by a consistent set of locks [42]. Table 1 summarizes the monitors and how they are supported in BugSifter.

AddrCheck sets ranges of metadata in response to memory management events (e.g., malloc) and checks that accesses go to allocated memory regions, which is the common case. AddrCheck has a metadata factor of eight, keeping one bit of metadata per application byte to encode two states (*allocated* or *unallocated*).

MemCheck and TaintCheck propagate metadata values from an instruction's source operand(s) to its destination operand and perform checks to ensure correctness. Commonly, these actions do not change metadata (source and destination metadata are the same) and the checks simply confirm that an access is legitimate (e.g., use of initialized values for MemCheck, non-malicious use of data for TaintCheck). MemCheck has three metadata states (i.e., *unallocated*, *uninitialized*, and *initialized*) and TaintCheck has two metadata states (*untainted* and *tainted*). For these two monitors, we use two bits of metadata per application byte, or a metadata factor of four. This ratio enables us to access 1-byte of metadata for common 4-byte operands, avoiding sub-byte update costs. This optimization is not necessary for AddrCheck as it rarely updates the metadata.

MemLeak performs reference counting to identify memory leaks. MemLeak maintains one metadata word per application word, which is a pointer to the context of the corresponding malloc and a null value otherwise. The context includes a unique ID, the PC, and a reference counter. At the end of the program's execution, non-freed heap objects having a zero value counter are identified as memory leaks. In addition to the main metadata map, BugSifter uses an auxiliary map that just keeps *pointer/non-pointer* information, requiring one bit of storage per application word. BugSifter caches the auxiliary map to filter out instructions with no pointer operands, while the full map is maintained in software.

LockSet identifies accesses to data that are not protected by a consistent set of locks. LockSet's metadata include information about the state of an application word (*uninitialized, thread-local, read-shared*, and *write-shared*), and a pointer to the corresponding lockset for shared memory locations. Overall, LockSet keeps one metadata word per application word (i.e., metadata factor of one).

AtomCheck detects atomicity violation by checking access interleavings. For this purpose, AtomCheck maintains information about the last access by each thread to each memory location. The main structures are (i) a global table to keep the id of the thread that last referenced each memory location, and (ii) local per-thread tables to keep the type (Read/Write) of the last access by each thread. AtomCheck encodes the thread id in one byte, and maintains one piece of metadata per application word (i.e., metadata factor of four). For each memory access, AtomCheck first checks the global table and the current thread id. If they match (i.e., the memory location was previously referenced by the same thread), which happens in the common case, a simple software handler is dispatched to update the metadata of the local table for the current thread. Otherwise, a complex handler is dispatched to check if there is a potential atomicity violation. BugSifter performs *partial* filtering for AtomCheck by caching the global table and checking if an access falls into the common case.

5.2 Additional Monitors

The above monitors cover a range of bug types while keeping the evaluation tractable and focused. To further stress the generality of BugSifter, we now describe several additional bug-finding tools that can be efficiently accelerated by the proposed design.

Pointer-related monitors perform checks to confirm the correct use of pointers. For example, Hardbound [17] proposes a bounded pointer primitive to identify illegitimate accesses due to mistakes in pointer arithmetic and array indexing. To monitor heap objects, it is sufficient to intercept malloc-family calls. Compiler support is necessary to initialize pointer metadata for the non-heap objects. BugSifter can filter accesses to non-pointer data, which correspond to the 80% of the application events.

Type safety monitors perform a type check upon a memory access. Metadata are updated when a memory location is written. Compiler support is required to insert type annotations in the application code [27]. BugSifter filters accesses to initialized data with the same type. The filtering rate is expected to be comparable to MemCheck (around 87%), because type-safety monitors follow identical propagation rules, and check if application accesses initialized data that have the same type.

Table 2. Simulation Setup

Simulator Parameters	
Private L1-I Private L1-D	16KB, 64B line, 2-way assoc., 1-cycle access lat.
Shared L2	512KB, 64B line, 8-way, 10-cycle access lat., 4 banks
Main memory	200-cycle latency
Log buffer	64KB, 1B per compressed record

Race detectors that use vector-clock based algorithms (e.g., ThreadSanitizer [43], FastTrack [20]) can benefit from BugSifter, which can identify accesses to thread-local and/or lock-protected data. On such accesses, BugSifter dispatches simpler handlers, which just update the (Read/Write) vector clock for the accessed memory location. This partial filtering, which is similar to AtomCheck, elides the expensive full check of vector clocks. The filterable accesses account for 80% of the application memory accesses according to our results for LockSet, which corroborate prior work [22].

6 Methodology

Without any loss of generality, we evaluate BugSifter by extending the baseline LBA system. We perform our experimental evaluation with both (i) the baseline LBA system and (ii) the baseline LBA enhanced with BugSifter. We implement the baseline LBA by extending the Simics [51] full-system simulator with event capture and event dispatch support. Table 2 describes our simulation setup. We model a dual-core IA32 system, running the application on one core and the monitor on the other core. BugSifter uses a 4KB, two-way associative metadata cache, and an 8-entry metadata RF, each with one-cycle access latency. To estimate the energy implications of BugSifter's metadata cache optimizations, we use Cacti v.5.3 [1] to model the SRAM structures in 45nm.

Our studies are performed on the six monitors described in Section 5.1. For LockSet, we perform stack updates and check all memory accesses (stack/non-stack) to provide end-to-end coverage. For AtomCheck, stack updates (i.e., due to function calls/ returns) do not need to be processed because AtomCheck keeps information for the last access (in the stack) locations across function calls. We consider two variants of AtomCheck that differ in the monitoring policy of the stack accesses. The first variant identifies the presence of potential atomicity violations in the stack, while the second variant can provide detailed information regarding the accesses involved. We only present results for the first variant, and note that the performance trends of the second variant are identical.

To evaluate performance, we use the SPEC2000 integer benchmarks for all the monitors except AtomCheck and LockSet. We choose these benchmarks because they are CPU-intensive and stress instruction-grain monitoring. In order to keep the simulation time tractable, we present results for the test inputs, as we find that BugSifter achieves the same filtering rate for the test and ref input (Figures 2(b) and 2(c)). For AtomCheck and LockSet, we use five multithreaded benchmarks: bzip uncompression [40], water from the SPLASH suite [53], and blackscholes, streamcluster, and swaptions from PARSEC [3]. Each benchmark is restricted to run on one core in a time-sliced manner. Our results present the slowdown as the CPI of the monitored application normalized to the

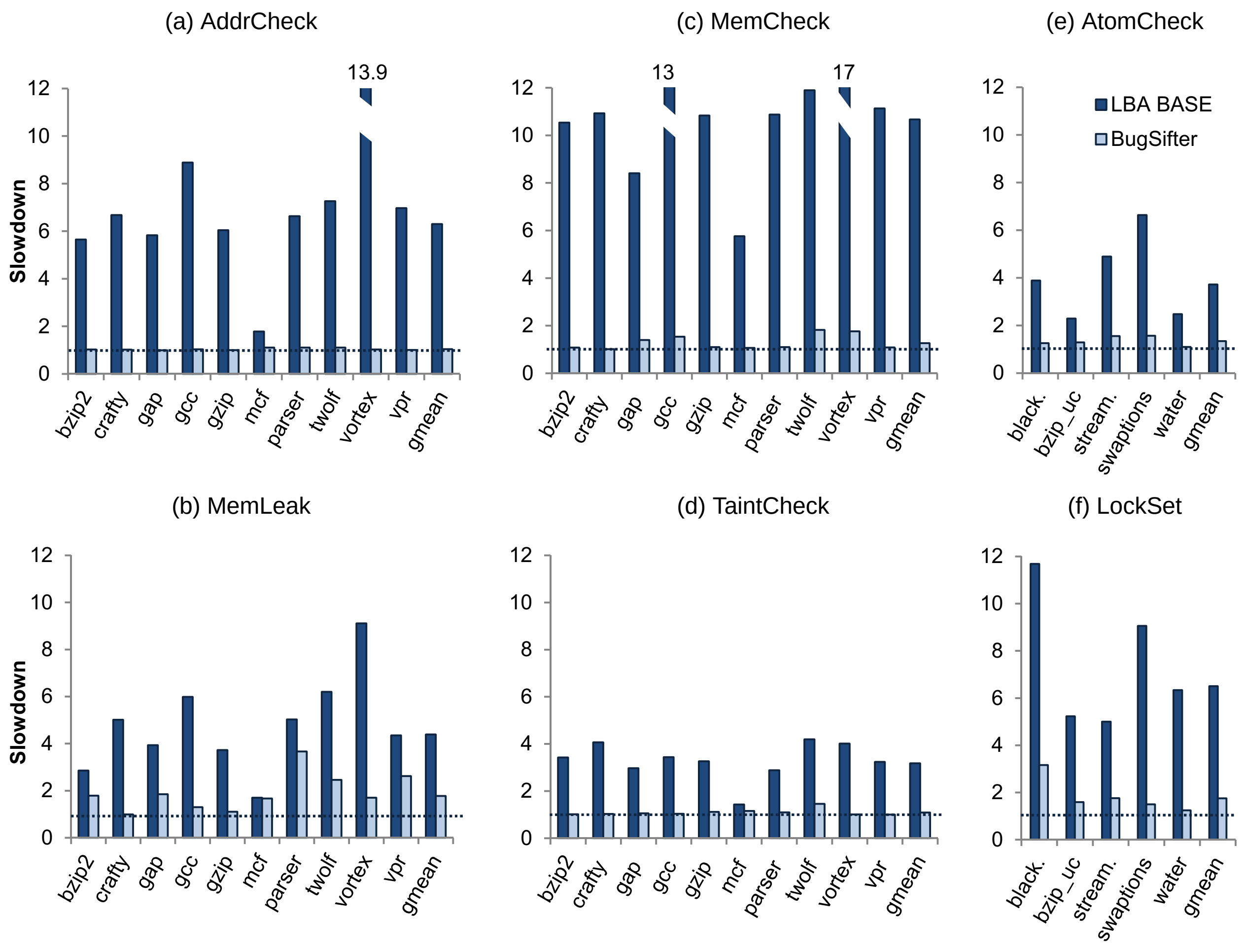


Figure 6. Performance benefits of BugSifter compared to baseline LBA

unmonitored application's CPI. A slowdown of one corresponds to zero monitoring overhead.

We also compare BugSifter with LBA+3ACC [9], an LBA system enhanced with three accelerators. The first accelerator, unary inheritance tracking (IT), targets the overhead of TaintCheck and MemCheck. Unary inheritance tracking uses hardware to accelerate only instructions with one source operand, adding extra overhead or sacrificing bug coverage for two-source operand instructions. The second accelerator, idempotent filter (IF), stores addresses to filter redundant metadata checks, but requires frequent storage flushing (upon stack-update, malloc, free, etc.). The third accelerator, a metadata TLB, maps an application address to a memory metadata address in hardware. The three accelerators reduce the slowdown of the six monitors described above, but may sacrifice bug coverage. For our experimental setup, we assume an 8-entry IT table, a 32-entry fully-associative IF, and a 1-cycle latency for the load metadata address instruction.

7 Results

In this section, we present the experimental evaluation of BugSifter, comparing to the baseline LBA. We present a metadata cache sensitivity analysis and show that a 4KB metadata cache is sufficient to significantly reduce monitoring overhead. We also show that BugSifter reduces the energy spent to access L1-D in comparison to the baseline LBA. Finally, we compare BugSifter with LBA+3ACC. In comparison to previous work, which has an overhead of 1-10x, we consistently reduce the monitor slowdown to less than 1.8x for all studied monitors.

7.1 Slowdown Reduction over Baseline LBA

Figure 6 shows the application slowdown for each benchmark for both the baseline LBA system and BugSifter. The y-axis shows the CPI of the monitored application normalized to the CPI of the unmonitored application. For all benchmarks and all monitors, BugSifter reduces slowdown significantly. AddrCheck and LockSet overheads are over 6x in the baseline LBA, which is reduced to 1.04x and 1.75x, respectively, in BugSifter. We reduce MemCheck's slowdown from over 10x to 1.27x, and TaintCheck from over 3x to only 1.09x. TaintCheck has a lower slowdown in the baseline because, it does not need to process stack-update events. BugSifter reduces the slowdown of MemLeak from 4.4x to 1.78x. For parser, twolf, and vpr, the slowdown is above the average, because they have the lowest filtering rate (50-60%). AtomCheck has a slowdown of 3.7x in the baseline LBA which is reduced to 1.34x with BugSifter's acceleration. In the baseline LBA, the mcf benchmark has the lowest slowdown, and vortex often has the highest because of their high and low CPI, respectively. Vortex produces events at a faster rate compared to the rest of the benchmarks, and hence imposes more stress on the monitor.

7.2 Cache Sensitivity Analysis

We present a sensitivity analysis of the metadata cache size, which shows that 4KB is, in fact, the sweet spot. Figure 7(a) shows the effect of varying the cache size on the metadata cache miss rate across the monitors. The miss rate is averaged across all benchmarks and is calculated as the ratio of events that resulted in a miss to the total number of events. As we increase the cache size to 4KB, the miss rate of all monitors drops below 5%.

Figure 7(b) shows the slowdown of BugSifter for the six monitors averaged across all benchmarks for different cache sizes. For the LockSet monitor, smaller metadata caches result in higher overheads, as high as ~3x for the 512B cache, because LockSet has a metadata factor of one. We find that a 4KB cache performs well across all monitors while only adding moderate storage overhead. The average slowdown with a 4KB metadata cache is 1.34x, and the maximum (MemLeak) is 1.78x

7.3 Energy Analysis

Because energy consumption is a primary concern in modern processors, we measure the effect of BugSifter's filtering and metadata cache optimizations on energy efficiency. We compare the energy consumed on accesses to both the instruction and data caches in the baseline LBA and BugSifter designs. In the case of BugSifter, we also account for the energy dissipated by accessing the 4KB metadata cache.

Because filtering elides the execution of software handlers, we expect BugSifter to reduce instruction cache energy, as well as the energy of non-metadata L1-D accesses. While BugSifter does not reduce the number of metadata accesses, the small metadata cache acts like a filter that reduces the energy of accesses that hit.

The results match the intuition. For MemCheck, BugSifter reduces the energy consumption in the first level of the cache hierarchy to 16% of the baseline. TaintCheck and AddrCheck both reduce the cache energy to about 30% of the baseline, and for LockSet, MemLeak and AtomCheck we reduce the energy to 22%, 80% and 80% respectively. The average energy reduction across the six monitors is 57% and arises through a combination of filtering, which elides the execution of software handlers and their associated L1-I and L1-D accesses, and the small metadata cache that reduces the energy spent on metadata accesses.

7.4 Comparing BugSifter with LBA+3ACC

We compare BugSifter's performance with the state-of-the-art prior work (LBA+3ACC) that enhances LBA with three hardware acceleration techniques but limits its bug-finding capability [8]. The three acceleration techniques are unary inheritance tracking, idempotent filters, and metadata TLBs. The first two techniques filter the event queue entries to reduce the number of software event handlers executed, and are monitor-specific. Unary IT applies only to MemCheck and TaintCheck. Furthermore, Unary IT assumes that instructions with two source operands always propagate a clean result to the destination's metadata; thus, TaintCheck may not detect all security violations. Idempotent filters store a small number of addresses of allocated memory in order to filter subsequent checks. This accelerator only applies to AddrCheck, LockSet, and only the part of MemCheck's functionality that deals with the allocated metadata value. The third technique, metadata TLB, applies to all monitors and reduces the size of software handlers for events having memory operands.

The LBA+3ACC system makes certain assumptions to reduce overhead, which limits bug coverage. In particular, LBA+3ACC assumes that MemCheck and LockSet do not monitor the stack, while on TaintCheck it assumes that instructions with two source operands are safe. To understand the implications of these assumptions and provide for a fair comparison, we also evaluate a variant of the LBA+3ACC without the coverage-limiting assumptions.

In Figure 7(c), we compare the performance of BugSifter with LBA+3ACC, both with and without the assumptions, averaged across all benchmarks. For AddrCheck, we applied a simple optimization that assumes stack memory to be allocated, or clean. This optimization benefits the two LBA+3ACC systems (i.e., with and without assumptions), as BugSifter provides hardware support for stack events.

As the figure shows, excluding AddrCheck, the slowdown for the LBA+3ACC systems without assumptions is up to 4.7x higher (MemCheck) than that of BugSifter due to the former's (a) lack of hardware acceleration for stack update events, and (b) much lower filtering rate (25-78% of instruction events for LBA+3ACC, versus 80-98% for BugSifter). The LBA+3ACC with assumptions offers better performance than the assumption-free version, but offers limited bug coverage while still slowing down the execution by up to 3.3x compared to BugSifter. On MemLeak and AtomCheck, the two LBA+3ACC systems offer the same performance, as both only benefit from the Metadata TLB optimization.

Compared to the baseline LBA system evaluated in Section 7.1, both LBA+3ACC variants offer some degree of performance improvement. The only exception to this is the AtomCheck monitor, which sees virtually no performance gain from 3ACC. AtomCheck benefits only from the Metadata TLB accelerator that eliminates just three instructions from the software handlers that often exceed 30 instructions in size. In contrast, MemLeak also benefits from only the Metadata TLB accelerator, but with an average initial handler size of eight instructions, the performance gain is higher. BugSifter, on the other hand, successfully accelerates all monitors, including AtomCheck, through a combination of flexible filtering, hardware-assisted stack updates, and metadata caching.

8 Related Work

Software-only solutions [12, 36, 37, 38, 39, 41, 42] provide flexibility but are impractical for deployed code due to up to two orders of magnitude overhead [36, 38, 48]. Hardware-assisted monitoring provides a trade-off between software flexibility and hardware performance, but the proposed solutions differ in the degree of generality, added hardware complexity, and overall monitoring acceleration. In this section we discuss the most closely related work on flexible monitoring and metadata caches.

8.1 Hardware-assisted monitoring

Early hardware-only proposals implement the monitor directly in hardware and hardwire the monitoring policy. Examples include data race detection [57], and propagation tracking [14, 46].

Other proposals add a monitoring pipeline to the processor. Unfortunately, the supported monitors are limited by the implementation complexity, because their pipelines can only accommodate fixed-sized metadata per application word, and cannot implement monitoring algorithms with more complex metadata, such as AtomCheck, bounded pointers [17] and vector clocks [43]. For example, MemTracker [50] and FlexiTaint [49] append an in-order pipeline to an out-of-order one. Among other fast path optimizations, FlexiTaint includes a form of filtering for tainting analysis. However, both pipelines supports only up to four bits of metadata per application word. Another system, Raksha [15], proposes a separate pipeline that propagates and checks the metadata transparently to the main processor's pipeline. Registers, caches, and memory are extended with fixed 4-bit tags.

iWatcher [58] and AccMon [56] check memory accesses that belong to pre-specified (i.e., "watched") memory ranges but cannot support propagation tracking monitors, such as MemCheck. AccMon [56] extends iWatcher to reduce the required monitoring activity using Bloom filters, but those cannot accommodate frequent metadata state updates (e.g., stack updates) efficiently.

INDRA [45] and Heapmon [44] use a helper thread to monitor application's execution. Heapmon employs filtering to reduce the overhead of the heap monitoring for MemCheck. However, it does not target other monitors, and does not provide mechanisms needed to accelerate stack monitoring (e.g., stack updates).

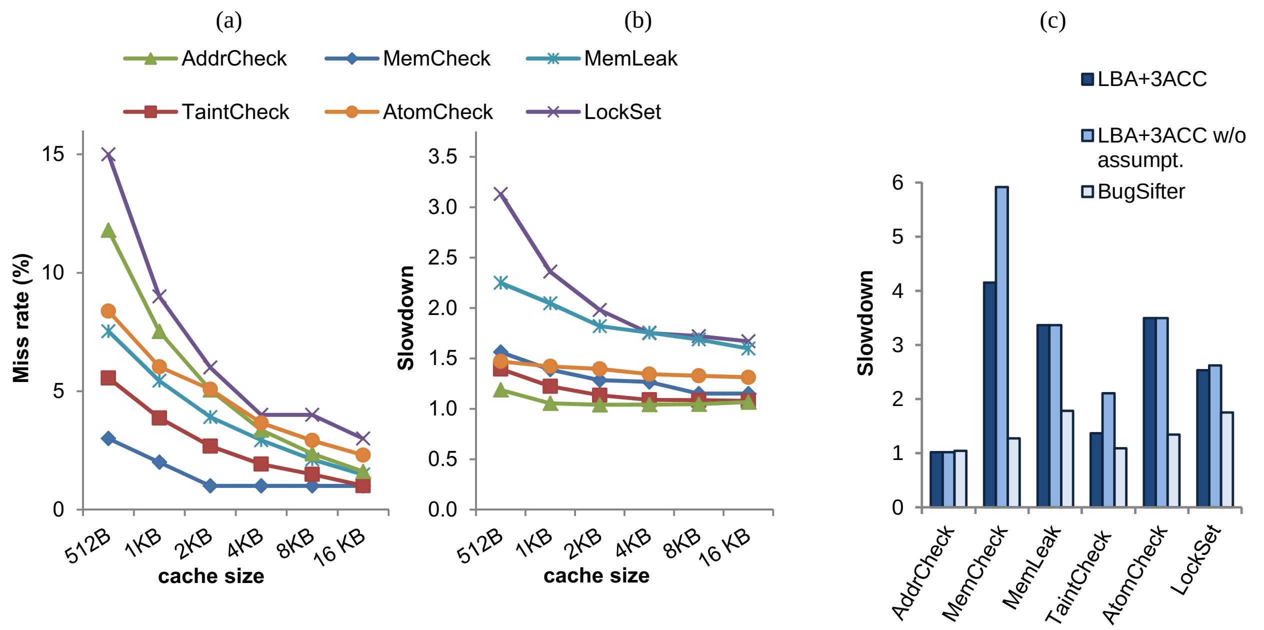


Figure 7. BugSifter (a) miss rate and (b) performance sensitivity analysis to cache size for the six monitors, and (c) the performance benefit of BugSifter compared to LBA+3ACC

DISE [13] targets the instrumentation cost of software monitors by augmenting instruction fetch and decode with a "macro-expansion" capability to dispatch handlers on the fly. BugSifter is orthogonal to DISE and could filter the event stream before the macro-expansion step.

A number of proposals suggest hardware support for a specific monitor. Greathouse et al. [22] propose a demand-driven race detection system that performs monitoring activity only when there is sharing among threads. Inter-thread sharing is identified by cache events using performance counters. However, this system can result in inaccuracies (miss some races), and performance degradation due to false sharing. Atom-Aid [30] leverages Transactional Memory to detect and survive potential atomicity violation bugs, by reducing the degree of memory interleavings. Radish [18] is a software/hardware approach for vector-clock based race detection, and maintains metadata in L1-D caches in order to reduce the number of data race checks in software. Radish requires specialized logic for SIMD-style vector clock computations. Hardbound [17] is a system that supports bounded-pointer analysis to provide spatial memory safety for C programs. WatchDog [34] is a hardware-based approach to provide safe and secure manual memory management. The insights used to provide acceleration are specific to pointer use and hence not applicable to other monitors.

Range cache [47] is a metadata cache designed to handle large, multi-bit metadata. Unlimited Watchpoints [23] proposes a mechanism for setting watchpoints on an arbitrary number of memory locations and uses the range cache, along with bitmaps and a lookaside buffer, to handle various spatial metadata patterns. There are performance benefits when the monitor's metadata show good spatial behavior. However, performance benefits are lower when byte-level watchpoints are necessary, or when the monitor performs frequent range updates that may spawn multiple ranges, such as stack updates.

Other proposals, such as FlexCore [16, 25], use FPGA tightly coupled to the processor to implement monitoring support. Hardware-only proposals that incorporate reconfigurable logic are able to accommodate multiple monitors but face two issues: (i) they require low-latency access to metadata, which is challenging for large metadata (e.g., a vector clock per word for FastTrack [20]) and (ii) they require the reconfigurable fabric to be clocked at frequency comparable to the monitored core. BugSifter's filtering methodology can assist these monitors (i) by identifying whether an event is filterable by accessing much less metadata kept in an auxiliary map (common case is encoded with one-two bits) and (ii) by filtering a large portion of application events allowing the reconfigurable fabric to run at lower frequency. Similarly, Introspective cores [33] implement monitoring support on a separate logic die using 3D die stacking. This work is orthogonal to BugSifter and can benefit from the acceleration and the energy savings demonstrated in our work.

8.2 Summary

Bug finding and general program monitoring are important challenges that have received considerable interest from the research community, which has proposed a number of acceleration techniques to overcome the performance drawbacks of software-only schemes. Thus, event filtering has been studied as a way to accelerate monitoring [17, 44, 49]; however, prior proposals only considered filtering for a narrow range of behaviors. In contrast, BugSifter demonstrates a filtering framework capable of accommodating a wide range of bug-finding tools. Additionally, BugSifter is unique in supporting partial filtering that provides a performance benefit even when software intervention is required. Finally, BugSifter accelerates stack events that have been ignored in previous work. The combination of configurable filtering and stack event acceleration enable a monitoring framework that is both general and fast.

9 Conclusions

We present BugSifter, a new light-weight hardware design to speed up instruction-grain monitoring that affords online bug detection. BugSifter exploits common monitoring behavior to provide simple, programmable hardware logic to accelerate bug finding. For frequently occurring instruction events, BugSifter takes advantage of the fact that most of the time applications behave correctly (i.e., invariant checks succeed) and that the monitor's metadata do not need to be updated (i.e., most updates are redundant). These observations allow the vast majority of the costly software handlers invoked in response to instruction events to be filtered out. Unlike prior work that does not monitor the stack, BugSifter performs bulk metadata updates in simple hardware to accelerate stack update events, which constitute up to 40% of the execution time in the software-only monitoring framework. To accelerate accesses to metadata both for filtered and unfiltered events, BugSifter uses a dedicated metadata cache. While BugSifter filters 80-98% of software handlers, it maintains full flexibility and generality by supporting software handler execution whenever needed. Our evaluation on single- and multi-threaded benchmarks using six diverse monitors reveals that BugSifter reduces the overhead over unmonitored applications from up to 6x incurred by previous techniques to just 1-1.8x, enabling continuous monitoring in development and in the field.

References

[1] HP labs. Cacti 5.1. http://www.hpl.hp.com/research/cacti.

[2] The MITRE Corporation. Common vulnerabilities and exposures (CVE) . http://cve.mitre.org.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[4] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: fast garbage collection without a long wait. In *OOPSLA*, 2003.

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

[6] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7), 2000.

[7] M. Cekleov and M. Dubois. Virtual-address caches part 2: Miltiprocessor issues. *IEEE Micro*, 17(5), 1997.

[8] S. Chen, M. Kozuch, P. B. Gibbons, M. Ryan, T. Strigkos, T. C. Mowry, O. Ruwase, E. Vlachos, B. Falsafi, and V. Ramachandran. Flexible hardware acceleration for Instruction-Grain lifeguards. *IEEE Micro*, 29(1), 2009.

[9] 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 *ISCA*, 2008.

[10] S. Chen, B. Lin, S. W. Schlosser, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, and G. R. Ganger. Log-based architectures for general-purpose monitoring of deployed code. In *the 1st workshop on Architectural and System Support for Improving Software Dependability*, 2006.

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

[12] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2007.

[13] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: a programmable macro engine for customizing applications. In *ISCA*, San Diego, California, 2003.

[14] J. Crandall and F. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO*, 2004.

[15] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *ISCA*, 2007.

[16] 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 *MICRO*, 2010.

[17] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: architectural support for spatial safety of the C programming language. In *ASPLOS*, 2008.

[18] 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 *ISCA*, 2012.

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

[20] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[21] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, Berlin, Germany, 2002.

[22] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. M. Austin. Demand-driven software race detection using hardware performance counters. In *ISCA*, 2011.

[23] J. L. Greathouse, H. Xin, Y. Luo, and T. Austin. A case for unlimited watchpoints. In *ASPLOS*, 2012.

[24] H. Kannan. Ordering decoupled metadata accesses in multiprocessors. In *MICRO*, 2009.

[25] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *International Conference on Dependable Systems and Networks*, 2009.

[26] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *the 1st workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[27] A. Loginov, S. H. Yong, S. Horwitz, and T. W. Reps. Debugging via run-time type checking. In *the 4th International Conference on Fundamental Approaches to Software Engineering*, 2001.

[28] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[29] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[30] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-aid: Detecting and surviving atomicity violations. In *ISCA*, 2008.

[31] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[32] J. Maebe, M. Ronsse, and K. D. Bosschere. Precise detection of memory leaks. In *Proceedings of the 2nd International Workshop on Dynamic Analysis*, 2004.

[33] S. Mysore, B. Agrawal, N. Srivastava, S.-C. Lin, K. Banerjee, and T. Sherwood. Introspective 3D chips. In *ASPLOS*, 2006.

[34] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *ISCA*, 2012.

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

[36] N. Nethercote. *Dynamic binary analysis and instrumentation*. PhD thesis, University of Cambridge, 2004.

[37] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *International Conference on Virtual Execution Environments*, 2007.

[38] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[39] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Network and Distributed System Security Symposium*, 2005.

[40] Parallel bzip2. http://compression.ca/pbzip2/.

[41] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: a Low-Overhead practical information flow tracking system for detecting security attacks. In *MICRO*, 2006.

[42] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4), 1997.

[43] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Workshop on Binary Instrumentation and Applications*, 2009.

[44] 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.

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

[46] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.

[47] M. Tiwari, B. Agrawal, S. Mysore, J. Valamehr, and T. Sherwood. A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags. In *MICRO*, 2008.

[48] G. Uh, B. Yadavalli, R. Cohn, and R. Ayyagari. Analyzing dynamic binary instrumentation overhead. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, 2006.

[49] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: a programmable accelerator for dynamic taint propagation. In *HPCA*, 2008.

[50] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. MemTracker: efficient and programmable support for memory access monitoring and debugging. In *HPCA*, 2007.

[51] Virtutech simics. http://www.virtutech.com/.

[52] 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 *ASPLOS*, 2010.

[53] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs. In *ISCA*, 1995.

[54] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[55] M. Xu, M. D. Hill, and R. Bodik. A regulated transitive reduction (RTR) for longer memory race recording. In *ASPLOS*, 2006.

[56] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: automatically detecting Memory-Related bugs via program Counter-Based invariants. In *MICRO*, 2004.

[57] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted lockset-based race detection. In *HPCA*, 2007.

[58] P. Zhou, F. Uin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: simple, general architectural support for software debugging. *IEEE Micro*, 24(6), 2004.

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