Run-Time Monitoring with Adjustable Overhead Using Dataflow-Guided Filtering

Daniel Lo, Tao Chen, Mohamed Ismail, and G. Edward Suh
Cornell University
Ithaca, NY 14850, USA
{dl575, tc466, mii5, gs272}@cornell.edu

***Abstract*—Recent studies have proposed various parallel run-time monitoring techniques to improve the reliability, security, and debugging capabilities of computer systems. However, these run-time monitors can introduce large performance and energy overheads, especially for flexible systems that support a range of monitors. In this paper, we introduce a hardware dataflow tracking engine that enables adjustable overhead through partial monitoring. This allows a trade-off to be made between monitoring coverage and overhead. This dataflow engine can also be extended to filter out monitoring operations associated with null metadata in order to reduce overhead. Given this architecture, we investigate how the dropping decisions should be made for partial monitoring and show that there exist interesting policy decisions depending on the target application of partial monitoring. Our experimental results show that overhead can be reduced significantly by trading off coverage. For example, for monitoring techniques with average overheads of 2-6x, the proposed architecture is able to reduce overhead to 1.5x while still achieving 14-85% average coverage.**

I. Introduction

Run-time monitoring techniques have been shown to be useful for improving the reliability, security, and debugging capabilities of computer systems. For example, Hardbound is a hardware-assisted technique to detect out-of-bound memory accesses [1]. Intel has recently announced plans to support buffer overflow protection similar to Hardbound in future architectures [2]. Similarly, run-time monitoring can enable many other new security, reliability, and debugging capabilities such as fine-grained memory protection [3], information flow tracking [4], [5], hardware error detection [6], data-race detection [7], [8], etc.

Previous studies have explored various design points in terms of efficiency, flexibility, and hardware costs for implementing run-time monitors. Custom hardware designs that target a single or narrow-range of monitors have been shown to have low performance overhead. On the other hand, flexible systems that support a range of monitors have shown much higher overhead. In this paper, we explore a new trade-off dimension for run-time monitor designs. We propose to enable low overhead monitors without sacrificing flexibility by using partial monitoring. In essence, this enables a trade-off between overhead and monitoring coverage or accuracy.

Although partial monitoring comes at the cost of reduced monitoring coverage (i.e., effectiveness), this can still be useful in many scenarios. For example, partial monitoring can enable a level of protection even for systems where the full monitoring overhead is too high. This is especially true for energy- or power-constrained systems or soft real-time systems where the monitoring overhead should not exceed energy/power limits or real-time deadlines. Additionally, partial monitoring can be used for sampling-based or cooperative debugging techniques which expect low coverage per run but use a large number of runs and users to collect debugging information [9], [10], [11].

Partial monitoring with adjustable overhead is achieved by dynamically dropping monitoring operations when the overhead exceeds a specified overhead budget. Although this results in reduced monitoring coverage (i.e., false negatives), false positives can also occur due to out-of-date metadata information. In order to prevent these false positives, we need to track the dropped metadata flows. We present a hardware-based dataflow tracking engine that keeps track of these dropped flows. With this architecture, we investigate different policies for deciding which events to drop for partial monitoring. These different policies show a trade-off between closely matching the overhead budget and increasing the monitoring coverage. In addition to enabling partial monitoring, we show how a simple extension to our dataflow engine can enable null metadata filtering. For example, for an array bounds check, checking non-pointer accesses is unnecessary and can be filtered out using the dataflow engine.

The main contributions of this paper are summarized as follows:

- We present a hardware architecture using a dataflow tracking engine to efficiently enable partial monitoring while preventing false positives. This hardware architecture is designed to be generally applicable to a wide-range of run-time monitoring techniques and monitor implementations.
- We investigate multiple policies on when and which operations to drop for partial monitoring, and show the trade-offs in the design space.
- We extend the dataflow engine to enable filtering null metadata to reduce overhead.

In order to evaluate our approach, we applied it to five different monitoring techniques. These monitoring techniques vary in what events they monitor, the size and semantics of their metadata, and the operations performed on metadata. These monitors show average slowdowns of 1.1-5.8x with the null metadata filtering when implemented on a multi-core platform. Our results show that partial monitoring can still achieve significant coverage with reduced overhead. For example, for the monitors which showed over 2x average overhead, slowdown could be cut down to 1.5x while still achieving a coverage of 14-85% on average.

978-1-4799-8930-0/15/$31.00 ©2015 IEEE

This paper is organized as follows. Section II introduces the notion of adjustable overhead through partial monitoring. Section III discusses the hardware architecture that enables partial monitoring using our dataflow tracking engine. Section IV investigates the design space for dropping policies that determine when and which monitoring operations to drop. Section V presents our evaluation methodology and results. Finally, we discuss related work in Section VI and conclude in Section VII.

II. Partial Run-Time Monitoring

A. Overhead of Run-Time Monitoring

There have been a number of proposals for run-time monitoring systems exploring various design points. Table I summarizes some of the representative designs and their reported performance overheads. The previous studies clearly show that there exist trade-offs between efficiency, flexibility, and hardware costs. For example, a run-time monitoring scheme can often be realized with fairly low performance overhead (less than 20%) if implemented with custom hardware that is designed only for one monitor or a narrow set of monitors. However, the custom hardware monitors cannot be modified or updated, and require dedicated silicon area. On the other hand, flexible systems that support a wide range of monitors lead to noticeable performance overhead, often too high for wide deployment in practice. Software-only implementations [18], [19], [20], [21] or multi-core monitors with minimal hardware changes [17] are reported to have severalfold slowdowns. On-chip FPGA monitors [14] and cores with monitoring accelerators [16], [15] can reduce overhead significantly, but still show slowdowns of several tens of percents in some cases. In today's monitoring systems, the overhead is also unpredictable because it depends heavily on the characteristics of applications and monitoring operations. In summary, users currently need to either pay noticeable overhead or the cost of custom hardware in order to use fine-grained run-time monitoring in deployed systems.

B. Partial Monitoring for Adjustable Overhead

In this paper, we aim to develop a general framework that enables monitoring overhead to be dynamically adjusted by dropping a portion of monitoring operations if necessary. In essence, this framework adds a new dimension to the monitor design space by allowing coverage or accuracy to be traded off for lower overhead. For example, the capability to adjust overhead allows users to use monitoring with partial coverage in order to reduce performance or energy overhead. Alternatively, partial monitoring allows designers to use less expensive hardware for a given performance overhead budget.

In this framework, a user specifies how much monitoring should be done in the form of a target overhead budget, a target coverage, a percentage of monitoring operations to be performed, etc. Then, the framework dynamically drops a portion of monitoring operations to match the target. In particular, this paper focuses on matching a performance overhead target while maximizing the coverage. Given that the overhead of custom hardware monitors can already be quite low, the focus is on enabling the trade-off in *general-purpose* monitoring systems that support a wide range of monitors. We also consider the target overhead as a soft constraint and do not aim to provide a strict worst-case guarantee.

C. Applications and Metrics

While it is ideal if run-time monitoring can be performed in full, we believe that the capability to trade-off coverage/accuracy for lower performance/hardware overhead will be useful in many application scenarios where full monitoring is not a viable option. Here, we briefly discuss example applications of partial monitoring and the metrics that are important in each case.

Cooperative testing, debugging, and protection: Recent studies have shown that software testing, debugging, or attack detection may be done in a cooperative fashion across a large number of systems [9], [10], [11], [5]. In this case, each system is only willing to pay very low overhead (e.g., a few percent) and only performs a small subset of checks. High coverage is achieved by having different systems check different parts of a program. The partial monitoring framework allows high-overhead monitoring to be used in a cooperative fashion. The main metric that represents the effectiveness of monitoring in this case is the coverage (the percentage of checks that were performed) over multiple runs.

Monitoring of soft real-time systems: Soft real-time systems or interactive systems need to meet deadlines or response-time requirements. Unfortunately, the overhead of run-time monitoring is often unpredictable and varies significantly depending on the application and monitor characteristics. The partial monitoring framework allows monitoring to be performed while providing a level of guarantee on its impact on the execution time. In this case, it is important that the system can closely match the desired overhead target while maximizing the effectiveness of monitoring.

Partial protection for low overhead: Even without real-time constraints, systems may have tight budgets for the performance, energy, or hardware overhead that they can tolerate for run-time monitoring. In such cases, full monitoring cannot be enabled unless its overhead is low enough. Adjustable monitoring allows partial protection on such systems. For example, array bounds may be checked for a subset of memory accesses. For dynamic information flow tracking (DIFT), a subset of information flows may be tracked for partial attack detection. In this scenario, the effectiveness of partial monitoring can be measured as the percentage of run-time checks that are performed on each program run, which reflects how likely it is for a bug or an attack to be detected for a system.

Profiling: The run-time monitoring system can be used to implement various profiling tools to collect statistics on program behavior for performance optimizations as well as security protection. For example, a recent study showed that an instruction mix can be used to identify malware from normal programs [22], [23]. In such profiling tools, the partial monitoring framework can be used to obtain statistical samples rather than complete counts of all program events, essentially trading off accuracy for lower overhead.

D. Design Challenges

While conceptually simple, designing a general framework to dynamically adjust monitoring overhead introduces new

Name	Type	Monitoring scheme and flexibility	Slowdown (avg./worst)
DIFT [4]	Custom HW	DIFT only	1.01x / 1.23x
FlexiTaint [12]	Custom HW	DIFT w/ programmable policies	1.01x-1.04x / 1.09x
Hardbound [1]	Custom HW	Array bounds checks only	1.05x-1.09x / 1.22x
Harmoni [13]	Custom HW	Tag-based monitors	1.01x-1.10x / 1.20x
FlexCore [14]	Dedicated FPGA	Instruction-trace monitoring	1.05x-1.44x / 1.84x
FADE [15]	Core+Custom HW	Instruction-trace monitoring (effective when HW filters work)	1.2x-1.8x / 3.3x
LBA-accelerated [16]	Multi-core+Custom HW	Instruction-trace monitoring (effective when accelerators work)	1.02x-3.27x / 5x
LBA [17]	Multi-core+Custom HW	Instruction trace monitoring	3.23x-7.80x / 11x
Multi-core DIFT [18]	SW (multithreaded)	DIFT (compiled for each application)	1.48x / 2.2x
LIFT [19]	SW (DBI)	DIFT (fully flexible)	3.6x / 7.9x
Purfiy [20]	SW (DBI)	Memory leak checks (fully flexible)	2.3x / 5.5x
TaintCheck [21]	SW (DBI)	DIFT (fully flexible)	10x / 27x

TABLE I: Trade-off between performance overhead and flexibility/complexity of run-time monitoring systems.

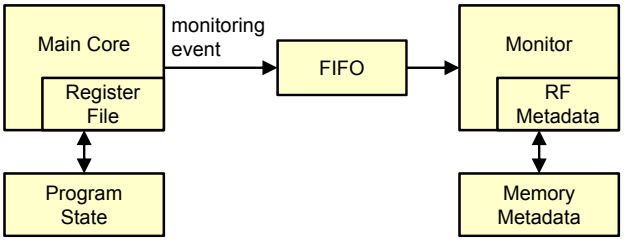


Fig. 1: Overview of run-time monitoring architecture.

challenges that need to be addressed. The following summarizes the main design goals and associated design challenges.

1) **General:** Since we mainly target flexible run-time monitoring systems, which often have high overhead, the framework also needs to be general enough to be applicable to a wide range of monitoring schemes.
2) **No false positive:** The framework needs to ensure that dropping a portion of monitoring operations does not lead to a false positive. We found that a dataflow engine that tracks invalid metadata can serve as a general solution to this problem (Section III).
3) **Intelligent dropping:** The framework needs to match the overhead budget while maximizing the effectiveness of monitoring. To this end, partial monitoring needs to carefully choose which operations to drop and when. We address this problem by studying different dropping policies (Section IV) and their trade-offs.

III. Architecture for Partial Monitoring

In this section, we present our hardware architecture that enables partial monitoring for adjustable overhead. Section III-A first describes our baseline model for run-time monitoring while the rest of the section describes our design.

A. Baseline Architecture

Figure 1 shows an overview of the run-time monitoring model that is assumed in this paper. The *main program* is a computation task that performs the original function of the system and is run on the *main core*. On certain events during the main program, such as the execution of certain types of instructions, the *monitor* performs a series of *monitoring operations*. The monitor operates in parallel to the main core. Events that trigger monitoring are referred to as *monitoring events*. Depending on the type of monitoring event, different monitoring operations are executed. Information about monitoring events is sent to the monitor and buffered in a FIFO structure to decouple the running of the main core and the monitor. If the FIFO is full, then the main core is forced to stall on a monitoring event until a FIFO entry becomes available. These stalls are a major source of overhead because the monitor may take several cycles to process a single event from the main core. Monitors typically maintain metadata corresponding to each memory location (`mem_metadata[addr]`) and register (`rf_metadata[reg_id]`) of the main core. If the monitor detects an inconsistent or undesired behavior in the monitoring events, then an error is detected.

There are many possible monitoring schemes that can be implemented on this type of fine-grained parallel monitoring architecture such as memory protection [3], information flow tracking [4], [5], soft error detection [6], data-race detection [8], [24], [25], [26], etc. For example, an array bounds check (BC) [1] can be implemented in order to detect when software attempts to read or write to a memory location outside of an array's bounds. This can be done by associating metadata with array pointers that indicates the array's base (start) and bound (end) addresses. On loads or stores with the array pointer, the monitor checks that the memory address accessed is within the base and bound addresses. In addition, this base and bound metadata is propagated on ALU and memory instructions to track the corresponding array pointers.

Figure 2 shows an example pseudo-code segment, its assembly level instructions, and the corresponding monitoring operations for an array bounds check monitor. First, an array `x` is allocated using `malloc` (line 1). As `malloc` returns the array's address in a register, the monitor associates base and bounds metadata with the corresponding register. Next, pointer `y` is set to point to the middle of array `x` (line 2). At the assembly level, a register `r2` is set to array `x`'s address plus an offset. The monitor propagates the metadata of the original pointer in `r1` to the metadata of `r2`. This is to ensure that pointer `y` is not used to exceed the array's bounds. Line 3a shows setting register `r3` to a constant value of 1. When this happens, `r3`'s metadata is reset to NULL in case it previously stored a pointer. Finally, the value of `r3` is written to memory using both pointers `x` and `y`. In both cases, the monitor checks whether the store address is within the bounds of the register metadata. In the first case (line 3b), `x+12` is within the original array's bounds. No error is raised and the metadata of `r3` is

Main Core (High-level)	Main Core (Assembly)	Monitor
1: int *x = malloc(4*sizeof(int));	1: call malloc ;return pointer 0x12340000 in r1	1: rf_metadata[1] = {r1, r1 + 0xf} // {base, bounds} of array
2: int *y = x + 2;	2: add r2, r1, #8	2: rf_metadata[2] = rf_metadata[1]
3: x[3] = 1;	3a: mov r3, #1	3a: rf_metadata[3] = NULL

	3b: str r3, [r1, #12] ; store to 0x1234000c	3b: if (r1 + 12 < base(rf_metadata[1]) ‖
		r1 + 12 > bound(rf_metadata[1])) {
		// raise error
		}
		mem_metadata[r1 + 12] = rf_metadata[3];

4: y[3] = 1;	4: str r3, [r2, #12] ; store to 0x12340014	4: if (r2 + 12 < base(rf_metadata[2]) ‖
		r2 + 12 > bound(rf_metadata[2])) {
		// raise error
		}
		mem_metadata[r2 + 12] = rf_metadata[3];

Fig. 2: Example of array bounds check monitor.

propagated to the metadata of the store address. If `r3` were a pointer, then this would allow a future instruction to load the pointer and use it to access its corresponding array or data structure. In the second case, `y+12` (line 4) corresponds to `x+20` which is not within the array's bounds and the monitor will raise an error.

Note that this monitoring is performed automatically and transparently with almost no modification needed to the main program. The only modification to the main program that is needed is to initially set metadata, such as setting the base and bounds addresses on a `malloc` call for array bounds checking. The propagation and checks of the metadata occur automatically as instructions are forwarded to the monitor. Here, we have shown the dynamic sequence of operations executed by the monitor, but the actual static code consists of a set of instructions to be run for each possible instruction type (load, store, etc.). Only instruction types relevant for the particular monitoring scheme are forwarded.

B. Effects of Dropping Monitoring

Our goal is to drop some monitoring operations in order to reduce the overhead of run-time monitoring. This dropping can affect the functionality of the monitoring scheme. There are three possible outcomes for dropping a monitoring operation. The first is that there is no difference in operation from the original execution. For example, if we drop line 3b from our array bounds check example (Figure 2), then the check on accessing `x+12` is skipped. However, this is a valid access and so skipping the check does not change anything.

On the other hand, if the monitoring for accessing memory location `y+12` on line 4 is skipped, then a false negative occurs. Originally, the monitor would catch this access as an out-of-bounds access and raise an error. However, if the monitoring operation for this is dropped, then the error is not detected. This reflects the trade-off that we make in order to reduce overhead. Instead of either 100% coverage with all the associated overhead or no coverage and no overhead, we enable middle points of partial coverage with some fraction of the full overhead.

The final possible outcome of dropping a monitoring operation is a false positive. For example, suppose the monitoring for line 1 is dropped, causing the bound information for pointer `x` to never be set. The result is that when the access using `x` is checked on line 3b, an error will be raised. This creates a false positive where an error is incorrectly raised. Although false negatives are part of the trade-off we make to reduce overhead, we need to prevent false positives.

C. Invalidation for Preventing False Positives

The key cause of false positives is dropping monitoring operations that update metadata. Dropping monitoring operations that check for an error (such as the check for line 4 in Figure 2) can only cause false negatives and will never cause false positives. On the other hand, skipping monitoring operations that update metadata can lead to false positives and false negatives. Essentially, when an update operation is skipped, we can no longer trust the corresponding metadata. Thus, our general approach for preventing false positives is to associate a 1-bit invalidation flag with each metadata in order to mark these metadata as valid or invalid. Furthermore, this invalidation information is propagated in the same way that metadata is. Figure 3 shows an example of associating invalidation flags with metadata. Suppose that the monitoring for line 1 is dropped in order to meet the overhead target. When a monitoring event is dropped, metadata that would have been updated is marked as invalid. In this case, instead of the normal operation of marking `rf_invalid[1]` as `false`, it is instead marked as `true`. Thus, when line 3 is reached, the monitoring event is dropped since `rf_invalid[1]` is marked as true. Note that in this case, line 2 also propagates this invalidation flag to `rf_invalid[2]` and causes the check performed on line 4 to also be dropped. This is necessary because an error would have been raised even if the access on line 4 was within bounds.

D. Dataflow Engine for Preventing False Positives

Although the functionality of dropping and invalidation could be implemented on the monitor, this is unlikely to be much faster than performing the full monitoring operations. Instead, in order to efficiently support dropping monitoring events and to prevent false positives, we propose to insert a hardware module between the main core and monitor (see Figure 4). This module handles the invalidation operations shown in the middle column of Figure 3. There are two operations that are done for handling invalidation information:

Main Core (Assembly)	Invalidation	Monitor
1: call malloc ;return pointer 0x12340000 in r1	1: rf_invalid[1] = false // true if dropped	1: rf_metadata[1] = {r1, r1 + 0xf} // {base, bounds} of array
2: add r2, r1, #8	2: if (rf_invalid[1]) { // drop monitoring } rf_invalid[2] = rf_invalid[1]	2: rf_metadata[2] = rf_metadata[1]
3a: mov r3, #1	3a: rf_invalid[3] = false	3a: rf_metadata[3] = NULL
3b: str r3, [r1, #12] ; store to 0x1234000c	3b: if (rf_invalid[1] || rf_invalid[3]) { // drop monitoring } mem_invalid[r1 + 12] = rf_invalid[3]	3b: if (r1 + 12 < base(rf_metadata[1]) || r1 + 12 > bound(rf_metadata[1])) { // raise error } mem_metadata[r1 + 12] = rf_metadata[3];
4: str r3, [r2, #12] ; store to 0x12340014	4: if (rf_invalid[2] || rf_invalid[3]) { // drop monitoring } mem_invalid[r2 + 12] = rf_invalid[3]	4: if (r2 + 12 < base(rf_metadata[2]) || r2 + 12 > bound(rf_metadata[2])) { // raise error } mem_metadata[r2 + 12] = rf_metadata[3];

Fig. 3: Example of using invalidation information to prevent false positives.

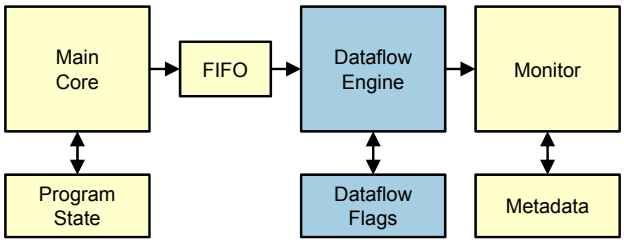


Fig. 4: Hardware support for dropping.

1) Propagate invalidation flags, following the dataflow of metadata.
2) Filter out monitoring operations based on invalidation flags.

Thus, the hardware acts effectively as a dataflow tracking engine in order to track a 1-bit invalidation flag per metadata. Figure 5 shows a detailed block diagram of this hardware module. The dataflow engine uses two address generation units to read in up to two invalidation flags. These source invalidation flags are used to decide whether a monitoring event should filtered. A third address generation unit is used to optionally specify a target to propagate the invalidation information. The module includes a register file to store invalidation flags corresponding to register file metadata. In addition, it uses a small memory-backed cache to handle invalidation flags corresponding to memory metadata.

Since different monitoring operations are performed based on instruction type, the dataflow engine is also configured based on instruction type. The source and operation of the address generation units are set based on instruction type. Note that the address generation units also take information from the monitored event as inputs. Thus, in the same way that the monitor selects metadata based on register indices or memory address of the specific monitoring event, the dataflow engine also reads the appropriate flags. In addition, the filter decision table is configured based on instruction type to decide what combination of input flags will lead to a filtered event and whether propagation is required.

Main Core (Assembly)	Null Filtering
1: call malloc ;return pointer 0x12340000 in r1	1: rf_null[1] = false
2: mov r2, #8 ; r2 is not a pointer	2: rf_null[2] = true
3a: mov r3, #1	3: rf_null[3] = true
3b: str r3, [r1, #12] ; store to 0x1234000c	3b: if (rf_null[1] && rf_null[3]) { // filter monitoring } mem_null[r3 + 12] = rf_null[r3]
4: add r2, r2, r3	4: if (rf_null[2] && rf_null[3]) { // filter monitoring } rf_null[r3] = rf_null[3] & rf_null[2]

Fig. 6: Example of using information about null metadata to filter monitoring events.

E. Filtering Null Metadata

One way to reduce the number of monitoring events that must be handled by the monitor is to filter out monitoring events that correspond to operating on null metadata. Null metadata correspond to events that are not relevant to the monitor. For example, Hardbound [1] filters out operations on non-pointer (i.e., no base and bounds metadata) instructions since it is not relevant to array bounds checking. More recently, FADE [15] has been proposed as a general hardware module to perform this null metadata filtering for a variety of monitoring schemes. Our architecture is able to support this null metadata filtering with a small modification.

Figure 6 shows an example of how this null filtering operates. Here, the main core's code has been slightly modified from Figure 2 and on line 2, `r2` is no longer set as an array pointer. Without null metadata filtering, the instruction for line 4 would be forwarded to the monitor since the system does not know whether `r2` contains a pointer address or not. However, if we use a 1-bit flag to mark `r2` as null when it is loaded with a constant, then we can propagate this null information and filter out monitoring for line 4.

The operations performed by null filtering are almost identical to the operations needed for invalidation shown in

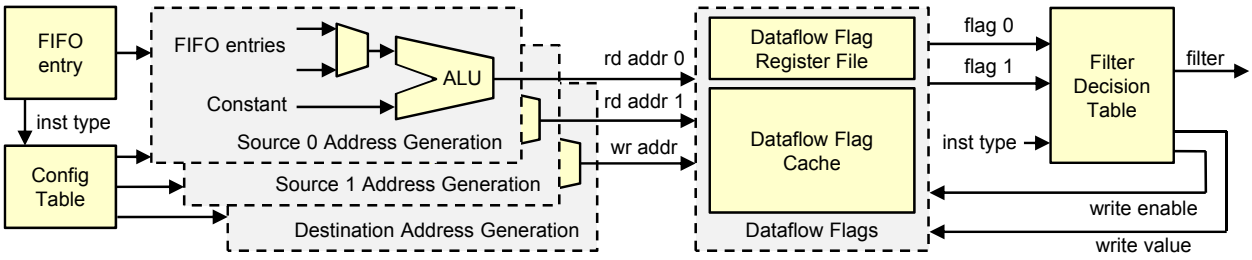


Fig. 5: Hardware architecture of the dataflow engine.

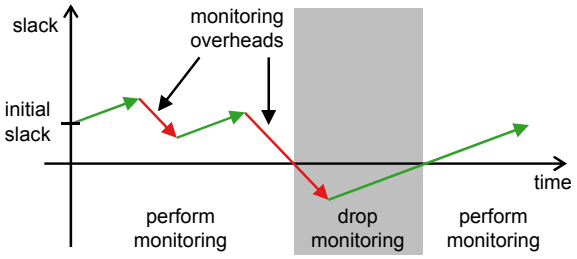


Fig. 7: Slack and its effect on monitoring over time.

Figure 3 except for a small change in the propagation policy. Instead of taking a logical OR of the source invalidation flags to determine whether monitoring can be skipped, null metadata filtering takes a logical AND of the source null flags. Thus, we can easily enable this null metadata filtering on our architecture by extending the dataflow flags to be two bits wide. One bit is used to keep track of invalidation information while the second bit is used to keep track of null information. All flags are initialized to null and the filter decision table is extended with the propagation and filtering decision rules for null metadata filtering. The result is a single hardware design that enables both partial monitoring and null metadata filtering.

IV. Dropping Policies

In order to use partial monitoring to enable adjustable overhead, we must also specify a policy for when and which monitoring events are dropped. In this section, we discuss some of the options and trade-offs for dropping policies. We split this decision into two components:

1) When do we need to drop events in order to enable reduced overhead? (Section IV-A)
2) Which events should be dropped? (Section IV-B)

A. Deciding When to Drop

In this section, we discuss two possible ways to determine when events should be dropped. The first possibility is to probabilistically drop events. By setting the probability of dropping events appropriately, overhead can be reduced. This works well for enabling partial monitoring for cooperative testing and debugging since the randomness allows different users and runs to monitor different portions of the program. However, using a probabilistic dropping policy can make it difficult to meet a target overhead without prior profiling.

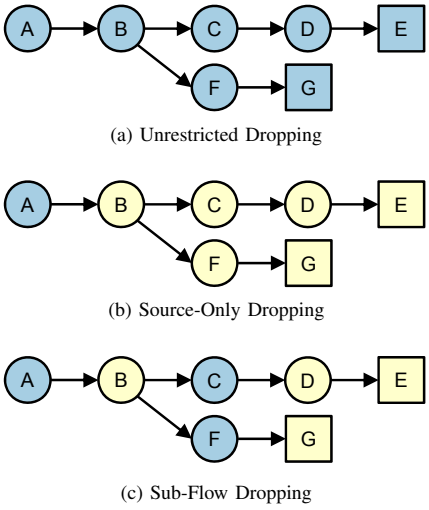
Alternatively, we can specify a target overhead and estimate, at run-time, the overhead of monitoring in order to decide whether dropping is needed. The overhead budget is specified as a percentage of the main program's execution cycles without monitoring. In order to estimate the overhead at run-time, we define *slack* as the number of cycles of monitoring overhead that can be incurred while staying within the budget target. Slack is essentially the difference between the actual overhead seen and the budget specified. Slack is generated as the main program runs and consumed as monitoring overheads occur. For example, if no monitoring overheads occur during 1000 cycles of the main program's execution and the designer sets a 20% overhead target, then the slack that is built up during this period is 200 cycles. If the main core is then stalled for 50 cycles due to monitoring, then the remaining slack is 150 cycles. In addition to this accumulated slack, a small amount of initial slack can be given in order for monitoring to be performed at the start of a program. Figure 7 shows an example of how slack can change over time. In this slack-based policy, if the slack falls below zero (i.e., the overhead budget is exceeded), then monitoring events are dropped.

Slack can be easily measured in hardware by using a counter that increments on every k-th cycle of the main core (e.g., every 5th cycle for a 20% target budget). The value of this counter is the accumulated slack. Whenever the main core is stalled due to the monitor, the measured slack is decremented. It is difficult to precisely determine the entire impact of monitoring on the main core due to the difficulty in measuring certain overheads such as contention for shared memory. However, we have found that using only the stalls due to FIFO back pressure works well in practice.

B. Deciding Which Events to Drop

In addition to deciding when dropping is required, trade-offs also exist in deciding which events should be dropped. The simplest policy is to drop monitoring events when slack is less than or equal to zero. However, this can result in wasted work. For example, consider the metadata dependence graph shown in Figure 8a. Here, an edge from node `A` to node `B` represents that if event `A` is dropped, then due to its invalidated metadata, it will cause event `B` to be dropped. Square nodes indicate events where monitoring checks are performed. In the example, suppose that event `E` is meant to perform a check operation but is dropped. In this situation, the monitoring operations that were done for events `C` and `D` were wasted since their results were not used for any monitoring checks. That is, by the time we decide to drop event `E`, we have already updated metadata for events `C` and `D` even though they are no longer needed.

An alternative dropping policy which eliminates this



(a) Unrestricted Dropping

(b) Source-Only Dropping

(c) Sub-Flow Dropping

Fig. 8: Comparison of dropping policies using metadata dependence graphs. Square nodes represent events where check are performed. Blue (dark) nodes indicate which nodes can be dropped.

wasted work is to only make dropping decisions at the root or source of these metadata flows (see Figure 8b). That is, we will decide to either monitor or not monitor an entire metadata flow. An example of these source nodes is the monitoring done to initialize base and bounds information on `malloc` for an array bounds check. These source nodes are easily identifiable by the dropping hardware because they typically correspond to the special instructions that are used to set up metadata information. Thus, it is not necessary to generate and analyze the monitoring dependence graph to identify source nodes. We refer to this dropping decision policy as *source-only dropping* and we refer to the previous policy of dropping any event as *unrestricted dropping*.

More complex dropping policies can be implemented given more detailed information about the monitoring dependence graph. This information can be found through static analysis or profiling, though we do not explore how to find it in detail here. Given this information, we describe one possible policy to use it, which we call *sub-flow dropping*. Sub-flow dropping makes dropping decisions at a granularity in between unrestricted dropping and source-only dropping (see Figure 8c). The basic idea of sub-flow dropping is to drop portions of the monitoring dependence graph at the smallest granularity such that no work is wasted. Sub-flow dropping allows source nodes to be dropped and nodes after branch points in the monitoring dependence graph to be dropped. From Figure 8c, this corresponds to nodes `A`, `C`, and `F`. For example, dropping node `C` causes nodes `D` and `E` to be skipped. However, performing monitoring on the remaining nodes allows the check at node `G` to be performed with no wasted work. Similarly, dropping `F` allows node `E` to be checked with no wasted work. Note that sub-flow dropping can still result in wasted work if there are merge points in the monitoring dependence graph and only part of the incoming flows are dropped, but it should result in less wasted work than compared to an unrestricted dropping policy.

Source-only dropping will result in no wasted work and thus better coverage at a given overhead than unrestricted dropping. However, because of the coarser-grained decision, it may be more difficult to closely match overhead targets. Sub-flow dropping enables a design point in between unrestricted dropping and source-only dropping in terms of matching overhead and coverage achieved. If the monitoring dependence graph is highly connected, then source-only dropping will perform poorly. On the other hand, we expect source-only dropping to work well when there are a large number of independent metadata flows. Monitoring dependence graphs with a large number of branches will favor a sub-flow dropping policy.

The choice of dropping policy can also depend on whether probabilistic dropping is performed instead of slack-based dropping. Probabilistic dropping works poorly with an unrestricted dropping policy. Since every event in a dependent chain (e.g., events `A` through `E`) needs to be monitored in order for the monitoring check to be useful, randomly dropping any event is likely to cause the final check to be invalid by dropping at least one event in the chain. Instead, source-only dropping works well when performing probabilistic dropping.

Depending on the target application of partial monitoring, different policies are more applicable. For applications where closely matching an overhead target is important, a slack-based, unrestricted dropping policy is appropriate. However, if matching the overhead target is not as important, then a slack-based, sub-flow or source-only dropping policy could provide better coverage. Finally, if the goal is to use partial monitoring to enable cooperative debugging and testing with very low overhead, then a probabilistic, source-only or sub-flow dropping policy can be used to provide good total coverage over multiple runs.

V. Evaluation

A. Experimental Setup

We implemented our dataflow-guided monitoring architecture by modifying the ARM version of the gem5 simulator [27] to support parallel run-time monitoring. We implement the monitor as a software-based monitor running on a parallel processor core, similar to LBA [17]. We model the main and monitoring cores as running at 2.5 GHz with 4-way set-associative 32 kB private L1 I/D caches and a shared 8-way 2 MB L2 cache. This setup is similar to the Snapdragon 801 processor commonly found in mobile systems. The dataflow engine uses a 1 kB cache for invalidation and null flags.

In order to explore the generality of the architecture for different monitors, we implemented five different monitors: uninitialized memory check (UMC), array bounds check (BC), dynamic information flow tracking (DIFT), instruction-mix profiling (IMP), and LockSet-based race detection (LS). Since we implement the monitor using a processor core and our dataflow engine was designed to be generally applicable, the same hardware platform supports all of these monitoring techniques. Uninitialized memory check seeks to detect loading from memory locations that are not initialized first. Array bounds check, as described in Section III, is a monitoring scheme that aims to detect buffer overflows where memory

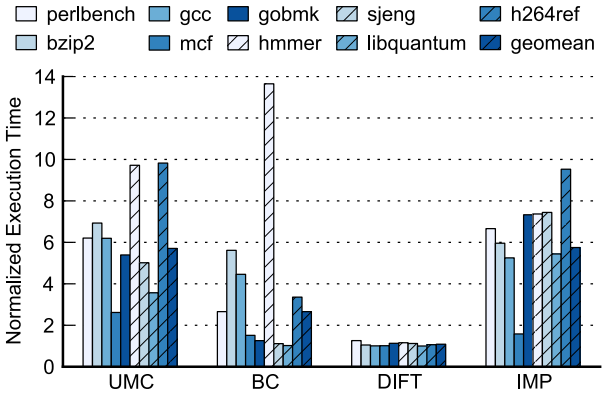


Fig. 9: Monitoring overhead with null metadata filtering.

accesses go beyond the boundaries of an array. We modify the implementation of `malloc` to set base and bound metadata information. Dynamic information flow tracking is a security monitoring scheme which detects when information from untrusted sources is used to affect the program control flow (i.e., indirect control instructions). For the benchmarks we consider, we mark data read from files as untrusted. We implemented a multi-bit DIFT scheme which marks untrusted data with a 32-bit metadata identifier so that if an error is detected, it is possible to have information about where the data originated from. Instruction-mix profiling counts the number of ALU, load, store, and control instructions that are executed. This profiling information can be useful for performing optimizations or to detect malicious software [23]. LockSet [24] attempts to detect possible race conditions in multi-threaded programs by tracking metadata about which locks are protecting shared memory locations. If a shared memory location is accessed while unprotected then a race condition may exist.

We tested our system using benchmarks from SPECint CPU2006 [28]. Since our implementation of BC depends on the modification of `malloc` to set array bounds information, we focus on the C SPECint benchmarks. Although we do not show results for the C++ benchmarks, we note that the results for UMC, DIFT, and IMP for these benchmarks are similar to the other results shown. We fast-forwarded each benchmark for 1 billion instructions and then simulated for 2 billion instructions.

Since LockSet race detection requires multi-threaded programs, we tested it using the applications from the SPLASH-2 benchmark set [29]. Each benchmark was run using 2 main cores to run the benchmark application. `fmm` and `raytrace` were run without fast-forwarding because they ran to completion in under 2 billion instructions. Each main core was connected to a dataflow engine and a monitoring core.

B. Baseline Monitoring Overhead

Figure 9 shows the execution times of performing monitoring with null filtering enabled normalized to the execution time of the benchmarks without monitoring. UMC sees normalized execution times of 2.6-9.8x with an average of 5.7x. For BC, normalized execution times are 2.7x on average

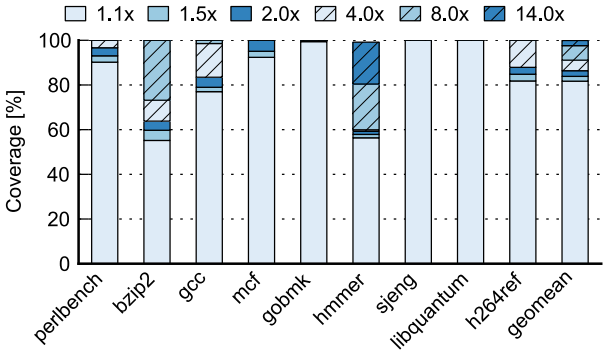


Fig. 10: Coverage vs. varying overhead budget for BC.

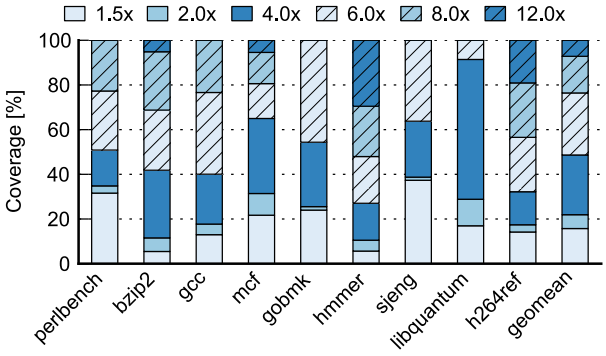


Fig. 11: Coverage vs. varying overhead budget for UMC.

and range from 1.02x to 13.6x. DIFT sees an average of 1.1x slowdown with a maximum slowdown of 1.3x with null metadata filtering. This low overhead is due to the fact that for our implementation of DIFT on SPEC benchmarks, we only mark data read from files as tainted. Instead, if we targeted network or streaming applications, which have larger amounts of untrusted input data, we would observe higher overhead. IMP shows normalized execution times of 1.6-9.5x with an average of 5.8x. Overheads for LS (not pictured) applied to the SPLASH-2 benchmarks are 1.04-1.29x after null metadata filtering and the average overhead observed is 1.13x. Our baseline system and overheads are similar to previous multi-core monitoring systems such as LBA-accelerated [16] and FADE [15] (see Table I). In Section V-D, we also evaluate a higher performance, FPGA-based monitor that shows low tens of percent of overhead.

C. Coverage with Adjustable Partial Monitoring

In this section, we evaluate the effectiveness of using partial monitoring to trade-off coverage for reduced overhead. For these results, we use a slack-based, unrestricted dropping policy. Figure 10 shows the monitoring coverage achieved by array bounds checking as we vary the overhead budget. We define *monitoring coverage* as the percentage of dynamic checks that are performed (indirect jumps in DIFT, loads in UMC, and memory accesses in BC and LS). The metric is chosen to understand how likely an error/attack instance is to be detected on an individual system, assuming that

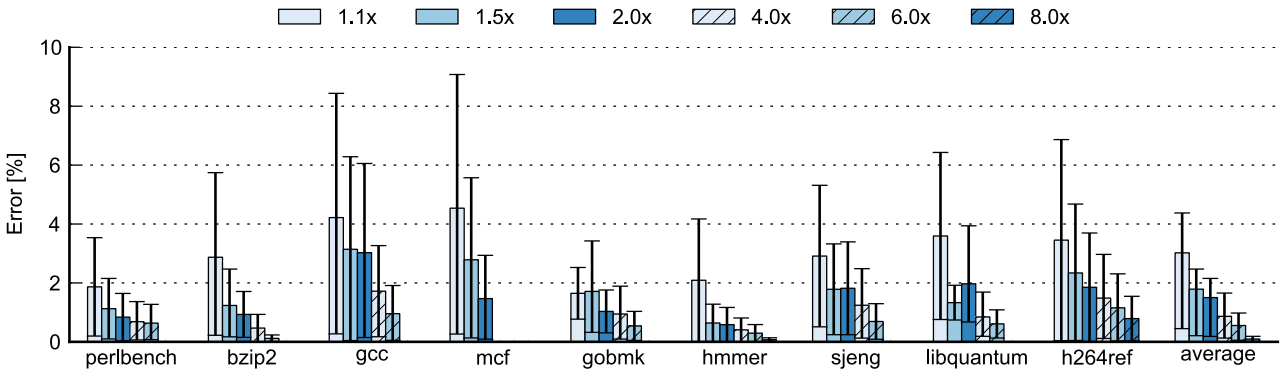


Fig. 13: Average error vs. varying overhead budget for instruction-mix profiling. Whisker bars show min and max errors.

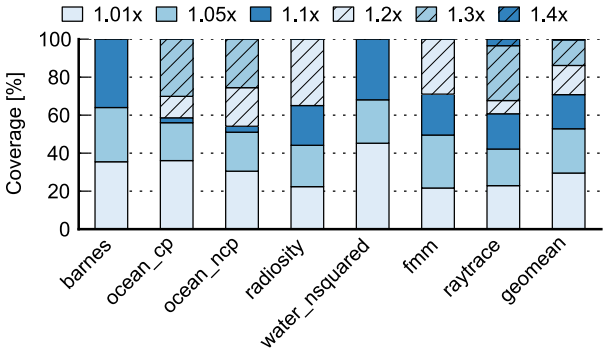


Fig. 12: Coverage vs. varying overhead budget for LS.

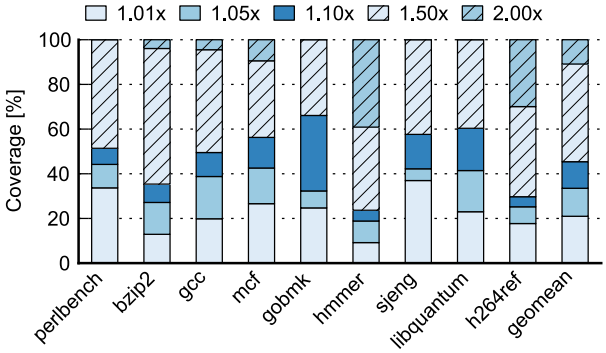


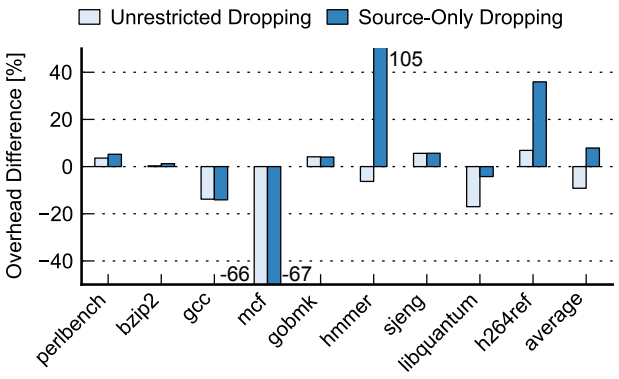
Fig. 14: Coverage versus varying overhead budget for UMC running on an FPGA-based monitor.

errors/attacks are uniformly distributed across checks. This may not necessarily be true for actual errors/attacks and so the reported coverage may not be the same as the probability of detecting actual errors/attacks. However, we believe this metric provides a good initial estimate of the detection capabilities of the system. In the figure, the bottom portion of each bar shows the coverage for a 1.1x overhead target and the additional coverage for increased overhead targets are stacked above this.

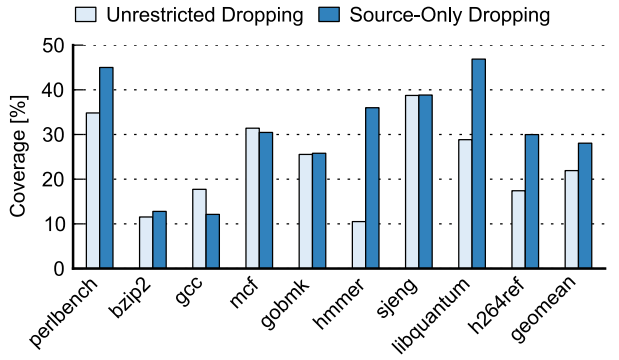
We see that by varying the overhead budget, the coverage achieved also varies. With only a 1.1x overhead target, array bounds check still achieves over 80% monitoring coverage on average. The high coverage achieved with such low overhead is due to two main effects. The first is that monitoring can be done in parallel, providing a certain level of monitoring coverage without introducing overhead. The second effect is that there may still exist a large number of monitoring events that do not lead to bounds checks. As a result, dropping these events can reduce overhead without a large impact on monitoring coverage.

Figure 11 shows the analogous graph for UMC. Again we see that varying overhead budgets enables partial monitoring. With a 2x overhead target, UMC achieves 22% monitoring coverage on average and with a 4x overhead target, this increases to 49%. Even higher coverage can be achieved by allowing higher overhead budgets. Similarly, Figure 12 shows the coverage achieved by LS. With only a 1.01x overhead, LS achieves 30% coverage on average. This increases to 71% coverage with a 1.1x overhead budget. For DIFT (not pictured), an overhead target of 1.05x is enough for all benchmarks tested to reach 100% coverage. Note that the overheads shown are the target overhead. In some cases, a high overhead target is needed in order to achieve 100% coverage. For example, for `mcf` with UMC monitoring, a 12x overhead target is needed to achieve 100% coverage while the overheads of performing monitoring in full were 2.6x (see Figure 9). This is due to the fact that we accumulate slack gradually and so bursty monitoring events may require a higher overhead target in order for all monitoring to be performed. However, the actual overheads seen are in-line with the overhead of performing monitoring without dropping (e.g., `mcf` with UMC monitoring at a 12x overhead target shows a 2.6x overhead).

The instruction-mix profiling monitor does not perform check operations. Thus, the concept of coverage is not applicable here. Instead, for each instruction type profiled, we calculate its count as a percentage of the total instructions monitored. We take the difference of these percentages compared to the case when full monitoring is performed to calculate the error. Figure 13 shows the min, max, and average error across instruction types for each benchmark and for varying overhead. We see that with a 1.1x overhead, a max error of 9% is observed across the benchmarks and the average error across instruction types and benchmarks is 3%.

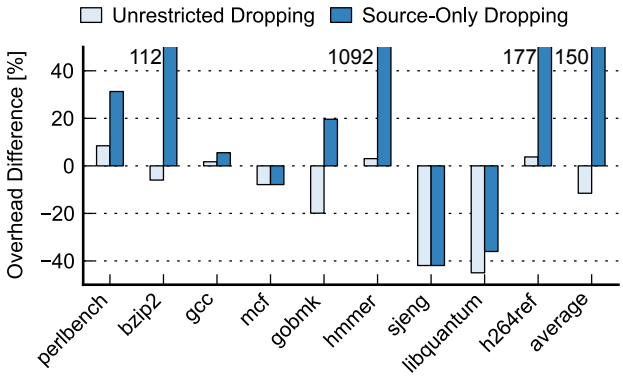


(a) Overhead budget error.

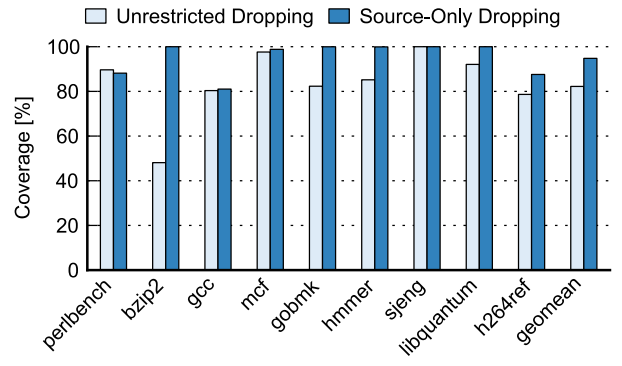


(b) Coverage.

Fig. 15: Comparing dropping policies for UMC.



(a) Overhead budget error.



(b) Coverage.

Fig. 16: Comparing dropping policies for BC.

D. FPGA-based Monitor

In addition to evaluating our system for a core-based monitor, we also evaluated partial monitoring on a higher performance FPGA-based monitor. This setup is based on the FlexCore [14] setup and uses a fully-pipelined monitor running on an FPGA fabric at half the frequency of the main core. We only show results for UMC due to space constraints, though other monitors show similar trends. In this case, the full overheads of monitoring range from 1.1-1.9x with an average overhead of 1.4x. Figure 14 shows the coverage achieved as we sweep the overhead target from 1.01-2.0x. We see that partial monitoring can allow the overhead of such a system to be pushed to 1.1x while still providing 45% coverage.

E. Comparing Dropping Policies

In this section, we evaluate the trade-offs between different dropping policies. Figure 15a shows the difference between the overhead budget and the run-time monitoring overhead for UMC when the overhead target is set to 2.0x. A positive value means that the overhead target was overshot while a negative value indicates that the overhead budget was met. For most benchmarks, we see similar results for unrestricted dropping and source-only dropping due to the fact that UMC consists of a large number of short, independent monitoring dependence chains. Source-only dropping causes overshoot of the overhead target for `hmmer` and `h264ref`. Figure 15b shows the coverage of UMC for unrestricted dropping and source-only dropping. We see that, in several cases, source-only dropping achieves higher coverage than unrestricted dropping while still closely matching the overhead target. For example, `perlbench` shows a 10% increase in coverage and `libquantum` shows an 18% increase in coverage by using source-only dropping. This is due to the fact that some of the overhead of monitoring for unrestricted dropping is being spent on wasted work as discussed in Section IV-B.

Next, we evaluate these trade-offs between source-only dropping and unrestricted dropping for BC. Figure 16a shows the overhead differences for BC and Figure 16b shows the coverage for BC. Here, the overhead target is 1.5x. From Figure 16a, we see that for several benchmarks, source-only dropping fails to meet the specified overhead target. The overshoot of the overhead target is quite high with overhead differences over 100% for `bzip2`, `hmmer`, and `h264ref`. Since only infrequently occurring array allocations are considered as source events for BC, it can be difficult for source-only dropping to match overhead targets. Although Figure 16b shows higher coverage for source-only dropping, this is largely due to the fact that it is running with higher overhead.

From these results, we see that depending on the monitor and the program behavior, source-only dropping can provide better coverage than unrestricted dropping with the same overhead. However, unrestricted dropping is better at meeting an overhead target. Thus, for applications where staying within

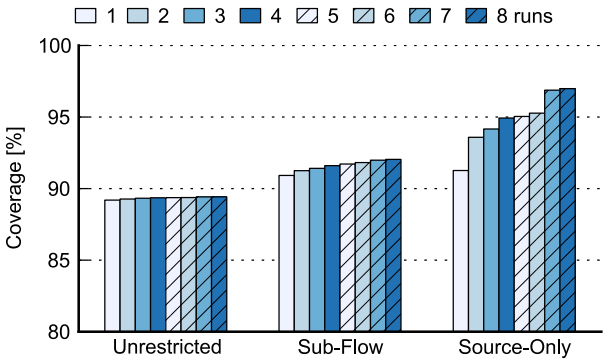


Fig. 17: Coverage over multiple runs for BC with a 10% probability to not drop events.

Monitor	Peak Power [mW]	Runtime Power [mW]
UMC	48 (4.9%)	29 (7.7%)
BC	69 (7.1%)	41 (10.7%)
DIFT	72 (7.4%)	41 (10.6%)
IMP	41 (4.3%)	20 (5.1%)
LS	31 (3.2%)	13 (3.9%)

TABLE II: Average power overhead for dropping hardware. Percentages are normalized to the main core power.

Monitor	1.5x Overhead	Full Monitoring
UMC	338 mW	363 mW
BC	320 mW	343 mW
DIFT	304 mW	304 mW
IMP	345 mW	367 mW
LS	327 mW	327 mW

TABLE III: Average runtime power of the monitoring core.

an overhead target is especially important, such as soft real-time systems, a slack-based, unrestricted dropping policy is more appropriate. On the other hand, if maximum coverage is desired and occasional slowdowns are acceptable, then source-only dropping can provide better coverage on average.

The results for the sub-flow dropping policy are not included in these graphs because our profiling infrastructure to identify sub-flow nodes currently does not support fast-forwarding. Instead, we compared the sub-flow dropping with other policies by running simulations for 1 billion instructions without fast-forwarding. The results (not shown) showed that sub-flow dropping produced similar coverage and overhead target matching to unrestricted dropping for the benchmarks and monitors that we tested.

F. Multiple-Run Coverage

One application of partial monitoring with low overhead is to enable cooperative debugging. The idea with cooperative debugging is to use partial monitoring with very low overhead across a large number of users or runs. By varying the pattern of partial monitoring done on each run, the goal is to achieve high coverage across multiple runs. Varying the monitoring that is done for different runs can be achieved by using a probabilistic dropping policy. Figure 17 shows the total coverage achieved over multiple runs for array bounds check using unrestricted, sub-flow, and source-only dropping policies with probabilistic dropping. These numbers are averaged across all benchmarks. Here, we use a 10% probability that events are not dropped. Each run was simulated for 500 million instructions. Since the effectiveness of cooperative debugging is often measured by code coverage, coverage here is measured as the percentage of static instructions which are monitored in at least one of the runs. We see that for the unrestricted dropping policy there is almost no increase in coverage. This is due to the fact that it is likely that at least one monitoring event in a metadata dependence chain will be dropped with the unrestricted dropping policy. Instead, sub-flow dropping and source-only dropping are much better suited for achieving high coverage over multiple runs. Source-only dropping shows a 6% increase in coverage with only eight runs while sub-flow dropping shows a 1% increase in coverage over eight runs. While sub-flow dropping shows a slower increase in coverage than source-only dropping, sub-flow dropping is better able to meet overhead targets. With enough runs, both sub-flow dropping and source-only dropping should be able to reach 100% code coverage.

G. Area and Power Overheads

Adding the dataflow engine in order to enable filtering and partial monitoring adds overheads in terms of area and power consumption. We use McPat [30] to get a first-order estimate of these area and power overheads in a 40 nm technology node. McPat estimates the main core area as 2.71 mm^2 and the peak power usage as 965 mW averaged across all benchmarks. The average runtime power usage was 385 mW. These area and power numbers consist of the core and L1 cache, but do not include L2 cache, memory controller, and other peripherals. The power numbers include dynamic as well as static (leakage) power. For the dataflow engine, we modeled the ALUs used for address calculation, the dataflow flag register file and cache, the configuration tables, and the filter decision table. These were modeled using the corresponding memory and ALU objects in McPat. We note that this is only a rough area and power estimate since components such as the wires connecting these modules have not been modeled. However, this gives a sense of the order-of-magnitude overheads involved with implementing our approach.

Our results show that an additional 0.197 mm^2 of silicon area is needed, an increase of 7% of the main core area. Table II shows the peak and runtime power overheads averaged across all benchmarks running with a 1.5x monitoring overhead target. The peak power is 31-72 mW, which is less than 8% of the main core's peak power usage. Similarly, the average runtime power is 13-43 mW, corresponding to 4-11% of the main core's runtime power.

Table III shows the runtime power usage of the monitoring core averaged across all benchmarks. These results are shown for an overhead target of 1.5x as well as when full monitoring is performed.

VI. Related Work

There exists a number of previous projects that have looked into performing partial monitoring in order to reduce the

Name	General	Adjustable Overhead	Prevent False Pos.
Scalable Bug Isolation [9]	-	-	-
Mem Leak Detection [10]	-	-	-
LiteRace [25]	-	-	✓
PACER [26]	-	-	✓
Testudo [5]	-	-	✓
Scalable Dataflow [11]	-	✓	✓
Arnold & Ryder [31]	✓	-	-
Huang et al. [32]	✓	✓	-
Lo et al. [33]	✓	-	✓
QVM [34]	✓*	✓	✓

*Limited generalizability

TABLE IV: Previous work on partial run-time monitoring.

performance overhead. These platforms differ from ours in a number of ways including how monitoring is implemented and the monitoring techniques targeted. However, we note three main properties that differentiates our work:

1) **Generality:** Our architecture applies to a variety of monitoring techniques.
2) **Adjustable Overhead:** Our architecture allows an overhead to be targeted. Other work performs sampling to reduce overhead but does not try to bound overhead, which we have shown can vary greatly.
3) **Prevent False Positives:** We present a mechanism to prevent false positives. Previous work either has false positives or targets monitoring techniques which degrade gracefully with sampling rather than exhibit false positives.

Our work is the first that we know of to present a hardware platform for partial monitoring that is general, allows an overhead target to be specified, and explicitly prevents false positives. In contrast, previous work on partial monitoring achieves only some, but not all, of these properties. Table IV summarizes these differences.

For example, there exists previous work for using statistical sampling to reduce the performance overhead of various debugging techniques. These include sampling for bug isolation [9], memory leak detection [10], race detection [25], [26], and information flow tracking [5], [11]. These techniques modify the monitoring to support statistical sampling and so are not generalizable. For those that prevent false positives, this is also done with monitor-specific modifications. Finally, with the exception of the work by Greathouse et al. [11], they do not allow an overhead target to be specified.

There also exists several projects that have looked into more general partial monitoring platforms. Arnold and Ryder [31] presented a general platform for sampling of instrumented code, but do not allow an overhead target to be specified and do not prevent false positives. Huang et al. [32] proposed a general framework for controlling the overhead of software-based monitoring. However, they also do not explicitly address false positives and only target monitors which degrade gracefully when performed partially. Lo et al. [33] designed a hardware architecture for performing monitoring on hard real-time systems that also prevents false positives. Their architecture is designed to meet real-time deadlines rather than enable adjustable overheads. In addition, in order to give strong guarantees, the coverage achieved is lower than our system. Finally, QVM [34] proposes a modification to the Java Virtual Machine (JVM) to support monitoring with adjustable overhead. QVM is limited to monitoring for code run on a JVM which limits its generalizability while our framework works for any binary. QVM prevents false positives by enabling or disabling monitoring on a per-object basis. This limits the monitoring schemes that can be implemented. Also, this method of preventing false positives is similar to the idea of performing source-only dropping which our results show can lead to overshooting the overhead target depending on the monitoring technique.

VII. Conclusion

Parallel run-time monitoring techniques are attractive solutions for improving the reliability, security, and debugging capabilities of systems. In this paper, we have presented an architecture that enables adjustable overhead through partial monitoring. The architecture uses a dataflow tracking engine to prevent false positives through tracking invalidation information. In addition, we show how the architecture can be extended to enable null metadata filtering. Given this architecture, we explore the design choices related to the dropping policy. With partial monitoring, we see that with a 1.5x overhead target, BC can still achieve 85% average coverage and UMC can still achieve 14% average coverage. At this overhead target, IMP shows less than 2% error on average. LS can be pushed down to 1.1x overhead and still achieve 61% average coverage.

Acknowledgments

This work was partially supported by the National Science Foundation grant CNS-0746913, the Air Force grant FA8750-11-2-0025, the Office of Naval Research grant N00014-11-1-0110, the Army Research Office grant W911NF-11-1-0082, and an equipment donation from Intel.

References

[1] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural Support for Spatial Safety of the C Programming Language," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[2] Introduction to Intel Memory Protection Extensions. [Online]. Available: http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions

[3] E. Witchel, J. Cates, and K. Asanovic, "Mondrian Memory Protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[4] G. E. Suh, J. Lee, D. X. Zhang, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[5] J. Greathouse, I. Wagner, D. Ramos, G. Bhatnagar, T. Austin, V. Bertacco, and S. Pettie, "Testudo: Heavyweight Security Analysis via Statistical Sampling," in *Proceedings of the 41st International Symposium on Microarchitecture*, 2008.

[6] A. Meixner, M. E. Bauer, and D. Sorin, "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," in *Proceedings of the 40th International Symposium on Microarchitecture*, 2007.

[7] 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 *Proceedings of the 39th International Symposium on Computer Architecture*, 2012.

[8] M. Prvulovic, "CORD: Cost-Effective (and Nearly Overhead-Free) Order-Recording and Data Race Detection," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, 2006.

[9] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable Statistical Bug Isolation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[10] T. M. Chilimbi and M. Hauswirth, "Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[11] J. L. Greathouse, C. LeBlanc, T. Austin, and V. Bertacco, "Highly Scalable Distributed Dataflow Analysis," in *Proceedings of the 9th International Symposium on Code Generation and Optimization*, 2011.

[12] G. Venkataramani, I. Doudalis, D. Solihin, and M. Prvulovic, "FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation," in *Proceedings of the 14th International Symposium on High Performance Computer Architecture*, 2008.

[13] D. Y. Deng and G. E. Suh, "High-performance Parallel Accelerator for Flexible and Efficient Run-time Monitoring," in *Proceedings of the 42nd International Conference on Dependable Systems and Networks*, 2012.

[14] 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 *Proceedings of the 43rd International Symposium on Microarchitecture*, 2010.

[15] S. Fytraki, E. Vlachos, O. Kocberber, B. Falsafi, and B. Grot, "FADE: A Programmable Filtering Accelerator for Instruction-Grain Monitoring," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, 2014.

[16] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. Gibbons, T. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible Hardware Acceleration for Instruction-Grain Program Monitoring," in *Proceedings of the 35th International Symposium on Computer Architecture*, 2008.

[17] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser, "Log-based Architectures for General-purpose Monitoring of Deployed Code," in *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[18] V. Nagarajan, H.-S. Kim, Y. Wu, and R. Gupta, "Dynamic Information Flow Tracking on Multicores," in *Proceedings of the Workshop on Interaction Between Compilers and Computer Architectures*, 2008.

[19] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu, "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks," in *Proceedings of the 39th International Symposium on Microarchitecture*, 2006.

[20] R. Hastings and B. Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," in *Proceedings of the Winter 1992 USENIX Conference*, 1991.

[21] 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 Systems Security Symposium*, 2005.

[22] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the Feasibility of Online Malware Detection with Performance Counters," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[23] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised Anomaly-based Malware Detection using Hardware Features," in *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions, and Defenses*, 2014.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs," *ACM Transaction of Computer Systems*, 1997.

[25] D. Marino, M. Musuvathi, and S. Narayanasamy, "LiteRace: Effective Sampling for Lightweight Data-race Detection," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[26] M. D. Bond, K. E. Coons, and K. S. McKinley, "PACER: Proportional Detection of Data Races," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[27] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, 2011.

[28] SPEC CPU™2006. [Online]. Available: http://www.spec.org/cpu2006/

[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[30] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, and D. M. Tullsen, "McPat: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the 42nd International Symposium on Microarchitecture*, 2009.

[31] M. Arnold and B. G. Ryder, "A Framework for Reducing the Cost of Instrumented Code," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[32] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok, "Software Monitoring with Controllable Overhead," *International Journal on Software Tools for Technology Transfer*, 2012.

[33] D. Lo, M. Ismail, T. Chen, and G. E. Suh, "Slack-Aware Opportunistic Monitoring for Real-Time Systems," in *Proceedings of the 20th Real-Time and Embedded Technology and Applications Symposium*, 2014.

[34] M. Arnold, M. T. Vechev, and E. Yahav, "QVM: An Efficient Runtime for Detecting Defects in Deployed Systems," in *Proceedings of the 23rd Conference on Object-Oriented Programming, Systems, Langauges, and Applications*, 2008.