Slack-Aware Opportunistic Monitoring for Real-Time Systems

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

Abstract—**Recent studies have shown that run-time monitoring is a promising approach for improving the security and reliability of computer systems. In this paper, we present a framework and architecture for applying run-time monitoring to hard real-time systems. In this framework, monitoring is only performed when enough dynamic slack exists in order to ensure that the monitoring does not impact the timing guarantees of tasks. If the slack is insufficient, a dropping operation is run which minimizes the timing impact on the task while ensuring that no false positives occur. We present a novel hardware architecture that can perform this dropping operation in a single cycle, matching the throughput of the task being monitored. Thus, run-time monitoring is able to be applied opportunistically, with no impact on the worst-case execution time of tasks. Our experimental results for three different monitoring techniques verify that timing is never violated and that false positives never occur. In addition, on average, 15-66% of monitoring coverage is achieved with no impact on the worst-case execution times of tasks depending on the monitoring technique. With an FPGA-based monitor, this average coverage of monitoring ranged from 62-86% depending on the monitoring technique.**

I. Introduction

Real-time systems are becoming increasingly prevalent as our world continues to become more computerized. Many of these real-time systems are found in safety-critical situations such as automobiles, airplanes, and medical devices where secure and reliable computation is critical. Errors in these systems can cause physical damage, injury, or even loss of life.

Recent studies have shown that monitoring of run-time program behavior can significantly improve the security and reliability of computing systems. For example, Dynamic Information Flow Tracking (DIFT) is a recently proposed security technique that tracks and restricts the use of untrusted I/O inputs, and has been shown to be able to effectively detect a large class of common software attacks [1]. Similarly, run-time monitoring can enable many new protection capabilities such as fine-grained memory protection [2], array bounds check [3], hardware error detection [4], etc. In fact, Intel has recently announced plans to support buffer overflow protection in future architectures [5].

These run-time monitors introduce performance overheads, both in the average case and in the worst case. Architectures that use programmable parallel hardware for monitoring, such as extra cores on a multi-core system or an FPGA coprocessor, have shown low average case overheads (low tens of percent) [6], [7], [8]. However, the worst-case execution time (WCET) can still be high. Previous work has shown an increase in WCET of up to 3.5x for a task with monitoring compared to without monitoring [9]. Thus, applying monitoring to real-time systems requires a large increase in the time allocated to tasks. Currently, if a real-time system cannot support this extra utilization, then monitoring cannot be applied to the system. In this work, we have developed a system that greatly decreases the amount of time that must be statically allocated to tasks in the worst case (i.e., WCET) in order to enable monitoring. Specifically, our system exploits dynamic slack in order to opportunistically perform as much monitoring as possible within the time constraints of the system.

The work in this paper is based on three key observations:

1) Tasks often run faster than their worst-case time.
2) The performance impact of monitoring is rarely equal to the worst-case impact.
3) Performing partial monitoring can still provide some degree of protection.

Since tasks typically run faster than their conservatively estimated WCET, there exists dynamic slack at run-time that can be used for monitoring. Similarly, the estimation of the WCET of monitoring is conservative. In actuality, the overheads of monitoring are much lower, as shown by the average-case performance impact. Finally, although performing all monitoring operations in full is preferred, performing only a portion of the monitoring still gives increased protection over a system with no monitoring applied. By shifting the decision of whether to enable or disable monitoring to run time, we are able to trade off monitoring coverage in order to reduce the WCET impact of monitoring. The system we present decides whether or not to perform monitoring based on whether there is enough dynamic slack to account for the worst-case performance impact of the monitoring.

A main challenge in skipping monitoring operations is ensuring that the monitor can still run in a useful manner. Although we trade off the coverage provided by monitoring in order to meet timing requirements, we must also guarantee that no false positives occur in order to prevent false alarms. We prevent false positives through the use of a dropping task that invalidates metadata when monitoring is skipped. With hardware optimizations, this invalidation can be handled with no impact on the task's WCET. In addition, the hardware architecture we present skips monitoring on invalidated metadata, saving dynamic slack to be used for useful monitoring on valid metadata.

We evaluate our architecture for three different monitoring techniques and several benchmark programs. In all our experiments, we saw that the monitoring program never caused the original program to exceed its WCET. We also saw that

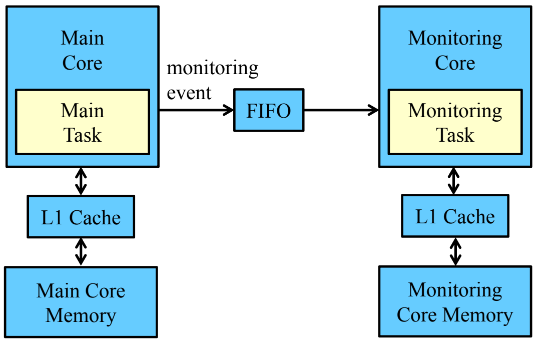

Fig. 1. Overview of run-time monitoring architecture.

false positives never occurred, as designed. For the three different monitoring techniques that we tested, we found that on average, depending on the monitoring technique, 15-66% of monitoring coverage could be achieved with no impact on the program's WCET. By applying our architecture to a high-performance FPGA-based monitor, the average coverage ranged from 62-86% depending on the monitoring technique. Additionally, by designing additional slack in the system, we were able to increase the number of monitoring operations that are performed.

This paper is organized as follows. Section II presents an overview of the run-time monitoring architecture assumed in this paper. In Section III, we discuss how to decide when to skip monitoring and how to handle this in a safe manner. Hardware optimizations for handling this dropping operation are presented in Section IV. We present our evaluation methodology and experimental results in Section V. Finally, we discuss some related work in Section VI and conclude in Section VII. In addition, the Appendix details how our architecture applies to three different monitoring techniques and includes some extra evaluation results.

II. Monitoring Architecture Model

There are several options on how to implement run-time monitoring. Implementing run-time monitoring in software using binary instrumentation or other similar methods introduces especially high overheads. For example, dynamic information flow tracking (DIFT) implemented in software suffers a 3.6x slowdown on average [10]. Implementing monitors in hardware greatly decreases the performance impact by performing monitoring in parallel to a program's execution. A dedicated hardware implementation of DIFT reduces average performance overheads to just 1.1% [1]. However, fixed hardware loses the programmability of software. A fixed hardware implementation cannot be updated and cannot change the type of run-time monitoring performed. Thus, recent studies have proposed using programmable parallel hardware, such as extra cores in a multi-core system or FPGA coprocessors, for monitoring [6], [7], [8]. Our work in this paper is targeted at these programmable parallel hardware monitors.

Figure 1 shows a block diagram of the architecture we focus on in this paper. This is a dual-core system where one core is used to run tasks while the second core is used to perform monitoring. The *main task* is the computation task that performs the original function of the real-time system and is run on the *main core*. The main task has a worst-case execution time (WCET) which can be upper-bounded using a variety

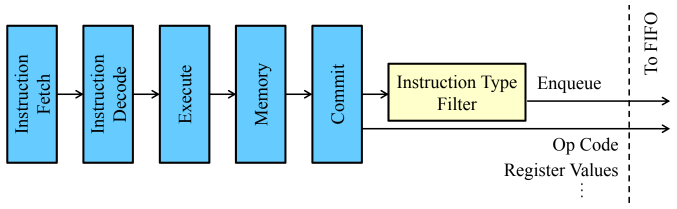

Fig. 2. The main core pipeline is modified to forward information on certain instruction types.

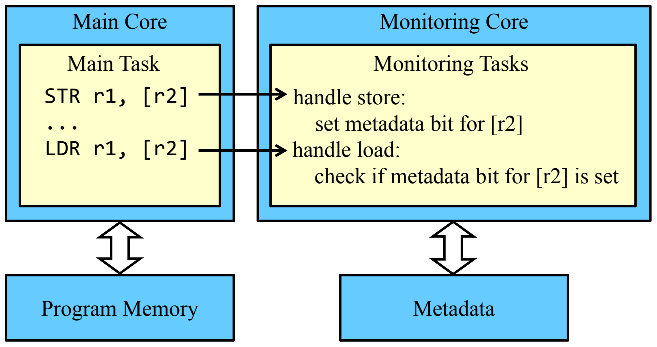

Fig. 3. Example of how an uninitialized memory check (UMC) monitoring scheme works.

of existing techniques [11]. This WCET is used for timing analysis in the design of the real-time system. As long as the execution time of the main task does not exceed this bound, then all timing guarantees from the analysis hold.

On certain events during the main task, a packet of information is sent to the *monitoring core*. This core will perform a *monitoring task* in order to check whether the system is operating properly and to perform bookkeeping for future monitoring tasks. The events that trigger monitoring are referred to as *monitoring events*. For example, a common monitoring event is the commit of certain instruction types (e.g., ALU, store, load, etc.). Figure 2 shows how an example main core pipeline can be modified to automatically forward information when instructions of certain types are committed. When an instruction commits, a comparator checks whether the opcode of the committed instruction is one that should be forwarded. If the instruction should be forwarded, then an enqueue signal is sent to a FIFO. Information about the instruction that is needed for monitoring, such as register values, memory locations accessed, etc., is stored in the FIFO on these events. In this way, the detection and handling of monitoring events is handled in hardware, with no need to modify the main task. A different monitoring task is executed depending on the type of monitoring event. We refer to the collection of monitoring tasks as a *monitor* or a *monitoring scheme*. Different monitoring schemes seek to ensure different run-time properties. If the monitoring scheme detects an inconsistent or undesired behavior in the sequence of monitoring events, then an error is detected. We do not focus here on how to handle such an error. However, there are several options on how to handle an error such as raising an exception, notifying the user, or switching to a simpler, more trusted main task [12].

There are many possible monitoring schemes that can be implemented on this type of monitoring architecture. One such monitoring scheme is an uninitialized memory check (UMC) where monitoring is used to detect when software attempts to read from a memory location that was not previously initialized. Figure 3 shows how UMC works. UMC forwards the memory addresses of store and load instructions to the monitoring core. For every memory location, the monitoring core keeps one bit of metadata information. On a store to a memory location, the monitoring core marks the corresponding metadata bit to indicate that the memory location has been initialized. On a load, the monitoring core checks that the corresponding metadata bit has been previously marked as initialized. If the metadata bit is not set, then an error is detected.

In order to decouple the execution of the main task and the monitoring task, a dedicated hardware FIFO is used to buffer monitoring events between the main core and the monitoring core. In the ideal case, the monitoring task operates in parallel with the main task. However, since the rate of monitoring events can be high (i.e., every instruction of the main task) and monitoring an event may take several instructions, the monitoring core is not always able to keep up with the throughput of monitoring events generated by the main core. Thus, when the FIFO is full, the main core is forced to stall on monitoring events. Figure 4 shows an example execution where a series of instructions are monitored. In this simple example, the FIFO only has 2 entries and it takes the monitoring task 2 cycles to process a monitoring event. For simplicity, only the commit stage of the main core pipeline is shown. After the 4th instruction (`ADD`) is committed by the main core, the FIFO is full and the monitoring core is busy. Thus, in order to not miss information about a monitoring event, the main core must stall before continuing execution and committing the 5th instruction (`STR`).

The baseline architecture we discuss uses an extra core on a multi-core system for monitoring. However, an alternate implementation would be to use an FPGA coprocessor to perform monitoring [7]. This system is similar to the system shown in Figure 1 except the monitoring core is an on-chip FPGA or FPGA-like fabric and the monitoring task is a hardware circuit implemented on this FPGA. With pipelining, such a design can achieve better performance than a processor-core based monitor. Although our discussion focuses on the processor-core based model, our method and architecture for opportunistic monitoring can also apply to an FPGA coprocessor based implementation and our evaluation in Section V includes results for such a design.

III. Opportunistic Monitoring

In this section we present a framework for opportunistically performing monitoring in such a way as to ensure that the main task's execution time does not exceed its WCET bound. The basic idea is that on a monitoring event, the system checks whether there is enough slack to account for the worst-case impact of monitoring on the main task's execution time. If there is enough slack, then the monitoring task proceeds. If there is not enough slack, then instead the event is dropped.

There are two main challenges in this scheme. The first is how to measure slack at run time and decide when it is necessary to drop monitoring tasks, which we discuss in Section III-A. Once it has been decided to drop a monitoring event, the second issue is how to drop the monitoring task in such a way as to maintain the correctness of the monitoring scheme which we discuss in Section III-B.

A. Measuring Slack

In order to decide when it is possible to perform monitoring, we must be able to measure the dynamic slack available. Dynamic slack is defined as the difference between a task's expected worst-case execution time (WCET) and its actual execution time [13]. This is only a portion of the total slack which is the difference between a task's finish time and its deadline (see Figure 5). Although the dynamic slack only accounts for a portion of the total slack, we only focus on dynamic slack because this is the portion of slack that is specific to a task. Additional slack in the schedule could be assigned to a specific task to be used for monitoring by the system designer or scheduler. For brevity, we will use the term slack to refer to dynamic slack.

In order to perform monitoring as the task runs, we need to be able to measure dynamic slack as the task runs. We can track dynamic slack by setting a number of checkpoints throughout the task. These checkpoints effectively divide the task into a number of sub-tasks. For each of these sub-tasks, the sub-task's WCET is determined. At run-time when a sub-task finishes, the difference between its actual run-time and its WCET is the slack generated by the sub-task. Specifically, we insert code to mark each sub-task boundary. At the beginning of a sub-task, the WCET of a sub-task is loaded into a timer. Each cycle, the timer decreases. At the end of a sub-task, the remaining value in the timer is the slack generated by the sub-task. This value is added to the current slack. An example of this process is shown in Figure 6. In our experiments, the division of a task into sub-tasks was done by hand but this process could be automated to divide a task using function boundaries, code length, or some other criteria. In addition to the slack accumulated while running, a portion of *headstart slack* can be initially assigned by the designer or scheduler to the task. For example, if the designer knows that static slack exists in the schedule, this slack can be added to the initial dynamic slack of a task to be used for monitoring.

By accumulating this slack as the task runs, we can determine whether monitoring can be performed while still meeting the real-time constraints. If the worst-case impact of a monitoring task on the main task is less than the accrued slack, then the monitoring task can execute. If running the monitoring task causes the main task to stall, as in the example shown in Figure 4, then slack is consumed. Slack was initially generated since the main task was running ahead of its WCET, so stalling up to the slack time will not cause the main task to exceed its WCET (see Figure 6). In the best case, the monitoring task executes entirely in parallel and does not affect the main task and thus consumes no slack. On the other hand, if the worst-case impact of the monitoring task on the main task's execution time is greater than the slack, then, conservatively, the monitoring task cannot be run. Instead, it must be dropped in order to guarantee that the main task finishes within its WCET.

B. Dropping Tasks

Dropping a monitoring task implies that some functionality of the monitor has been lost. This may cause either false

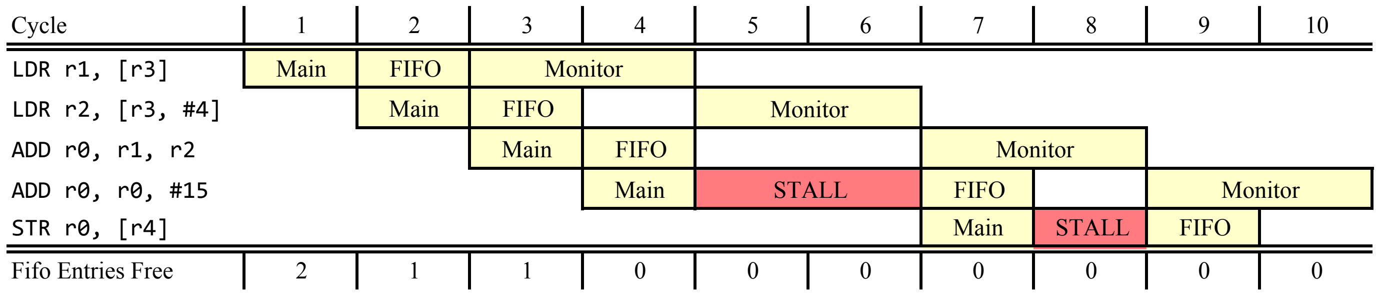

Fig. 4. Pipeline diagram of monitoring. The main core stalls due to the slower throughput of the monitor.

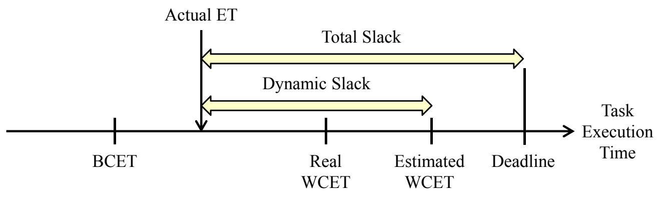

Fig. 5. Dynamic slack and total slack.

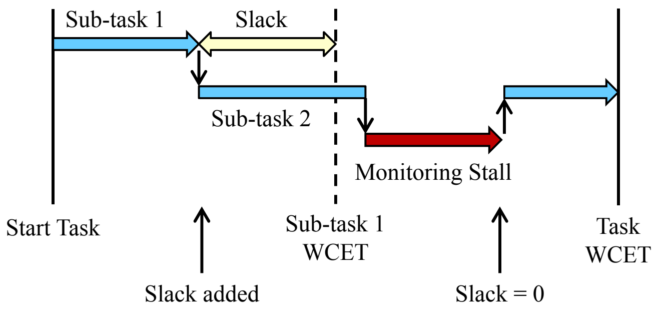

Fig. 6. Dynamic slack increases when a sub-task finishes early. Slack is consumed as monitoring causes the task to stall.

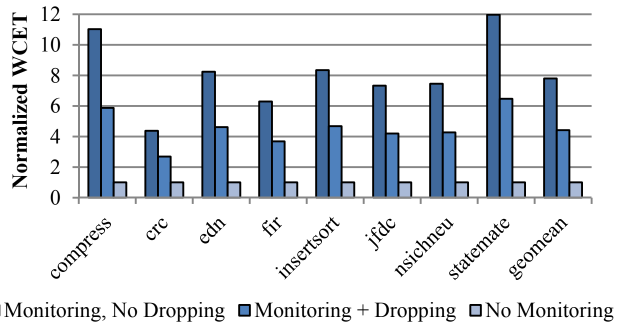

Fig. 7. WCET with original monitoring task, with software dropping task, and with no monitoring for UMC. Results are normalized to the WCET with no monitoring.

negatives, where an error that occurs in the main task's execution is not detected, or false positives, where the monitor incorrectly believes an error has occurred. For example, a false positive can occur for UMC if a store monitoring event is dropped. This causes the memory location of the store to not be marked as initialized. A subsequent load for the memory location will incorrectly cause an error to be raised. We accept false negatives as the loss in coverage that we trade off in order to ensure the WCET of the main task is met. However, we must safely drop monitoring events in such a way as to avoid false positives so that the system does not incorrectly raise an error.

In order to ensure that no false positives occur, we need to run a *dropping task* when a monitoring task is dropped. The specifics of how this dropping task operates may vary for different monitoring schemes. However, in analyzing various monitoring schemes, we found that most monitoring tasks perform operations of primarily two types: *checks* and *metadata updates*. Monitoring tasks *check* certain properties to ensure correct main task execution and they *update* metadata for bookkeeping. Skipping a check operation can only cause false negatives and will never cause a false positive. Therefore, the dropping task may simply skip a check operation. Skipping an update operation can cause false negatives but may also cause false positives. Essentially, when an update operation is skipped, we can no longer trust the corresponding metadata. In some cases, the dropping task can handle this by setting the metadata to a neutral value that will not cause false positives (e.g., cleared or null). A more general solution is for the dropping task to mark the corresponding metadata as invalid, to prevent future false positives.

In the worst case, no monitoring can be done and the system must ensure that there is enough time to run a dropping task for every monitoring event in order to avoid false positives. Thus, the main task's WCET estimation must be modified to take into account the worst-case impact due to the dropping task. By minimizing the dropping task's execution time, the impact on the main task's WCET can be much lower than the impact due to the monitoring task. Figure 7 compares the original WCETs with and without monitoring to the WCET with dropping tasks implemented in software for UMC. The WCETs are normalized to the WCET without monitoring. The WCET with dropping is reduced by 43% on average from the WCET without dropping.

IV. Hardware-Based Dropping Architecture

In Section III, we presented a general framework for how to design a system that allows dropping of monitoring tasks to ensure the main task's execution time. However, implementing this framework in software still has significant impacts on the WCET as shown in Figure 7. In this section, we present a novel hardware architecture that eliminates these impacts to the main task's WCET.

There are two main sources that affect the main task's WCET: the additional code for keeping track of slack and the impact of the dropping task. In Section IV-A, we describe hardware to keep track of slack and to make the decision on whether to run the monitoring or dropping task. In Section IV-B, we present a hardware-based metadata invalidation scheme that allows dropping to be performed in a single cycle. By handling the dropping in a single cycle, the throughput is able to match the throughput of the main core. Section IV-C

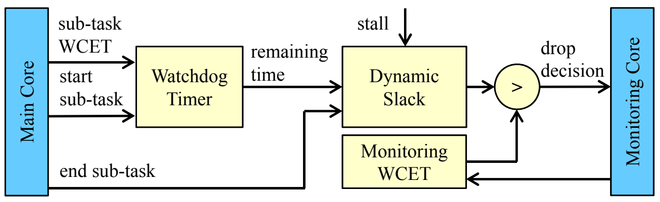

Fig. 8. Hardware modules for keeping track of slack and making a drop decision.

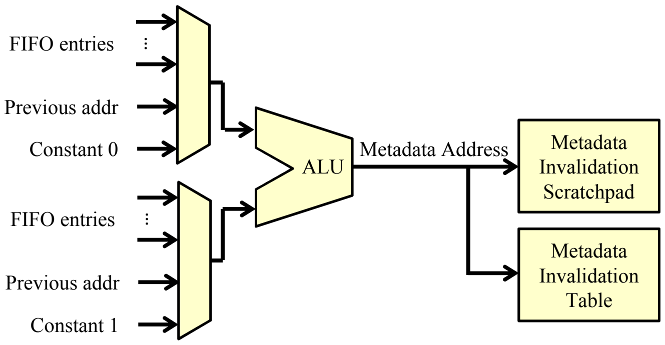

Fig. 9. Metadata invalidation module (MIM).

builds upon the hardware-based invalidation scheme to filter out monitoring tasks for invalid metadata in order to reserve slack for more useful cases of running the monitoring task. Finally, in Section IV-D we show the full architecture.

A. Slack Tracking

Figure 8 shows a hardware slack tracking module (STM). In order to keep track of slack, a watchdog timer is loaded with a sub-task's WCET at the start of a sub-task and counts down each cycle. At the end of a sub-task, the remaining time in this watchdog timer is added to the current dynamic slack which is stored in a register. At the start of a task, the dynamic slack register is cleared and set to the headstart slack if one is specified. In addition, whenever the main task is stalled due to the monitoring core, this dynamic slack is decremented.

The currently measured dynamic slack is used to determine whether the monitoring task can run in full. When the monitor is initialized, the monitor will load its worst-case needed slack in order to perform the monitoring task into a register. When there is a monitoring event in the FIFO to be processed, a hardware comparator checks whether the dynamic slack is greater than or equal to the necessary slack for full monitoring. If enough slack exists, the monitoring core is signaled to perform the monitoring task. Otherwise, the monitoring event is dropped.

B. Metadata Invalidation Module

In the worst-case, all monitoring events must be dropped. Thus, it is important that whatever dropping task needs to be run has a minimal impact on the main task. If this dropping task can match the maximum throughput of the main core, then the original WCET of the main task is not affected, removing the need to redo the WCET analysis. In this section, we present a hardware architecture that can handle dropping monitoring events in a single cycle, matching the throughput of the main core.

As mentioned in Section III-B, the dropping task must invalidate metadata on a dropped monitoring event. Thus, we have designed a hardware module to perform this invalidation. Figure 9 shows a block diagram of this module, which we call the metadata invalidation module (MIM). When the slack tracking module indicates that a monitoring task must be dropped, the metadata invalidation module sets a bit in the metadata invalidation table (MIT) corresponding to the metadata to be invalidated. The metadata to be invalidated depends on the monitoring scheme and the monitoring event. For example, for the uninitialized memory check monitoring scheme, on a store event, metadata corresponding to the memory access address is set to indicate initialized memory. Thus, on a dropped event, the MIM must be able to calculate the address of this metadata in order to set a corresponding invalidation flag. The MIM includes a small ALU to perform these simple address calculations. Since this metadata address mapping varies for different monitoring schemes, the inputs to the ALU can either be data from the monitoring event, the previous metadata address, or a constant. The input selection and the ALU operation to perform are looked up from a configuration table depending on the type of monitoring event. The monitor sets up this configuration table during initialization.

In order for this dropping operation to match the throughput of the main core (i.e., up to one monitoring event per cycle), the metadata invalidation information is stored on-chip. The MIT is implemented as a small on-chip memory, similar to a cache. This memory stores invalidation flags and is indexed using part of the metadata address. It stores the remaining portion of the address as a tag. Unlike a cache, if an access misses in the MIT, there is no lower-level memory structure to go to. This is done in order to ensure that the MIM can handle a monitoring event every cycle. Instead of backing the MIT with lower-level memory, if writing to the MIT would force an eviction, we instead mark the corresponding cache set as "aliased". From this point on, we are forced to conservatively consider any metadata that would be mapped to this cache set as invalid, regardless of the tag corresponding to its metadata invalidation address. This reduces the amount of useful monitoring that can be done, but guarantees that the dropping hardware can match the main core's throughput. These aliased sets can be reset by re-initializing (e.g., resetting to a null value) all metadata that could map to the aliased set. By using dynamic slack or a dedicated periodic task, the system can occasionally reset aliased sets. In either case, a sufficiently sized MIT should ensure that aliasing is rare.

In some cases we would like to use the MIM hardware to operate in such a way as to ensure that aliasing does not occur. For example, we found that some monitoring schemes save metadata information about registers. Since this metadata is used often, it is important to manage it in such a way that aliasing does not occur. Thus, the MIM also includes a metadata invalidation scratchpad (MISP). This MISP is explicitly addressed and it is up to the system designer to utilize it in such a way that aliasing will not occur.

Both the MIT and MISP are accessible by the monitor. This allows the monitor to be aware of what has been invalidated. The monitor can also re-validate metadata when possible, such as when the monitoring task writes metadata values.

We note that the MIM was designed with the idea of

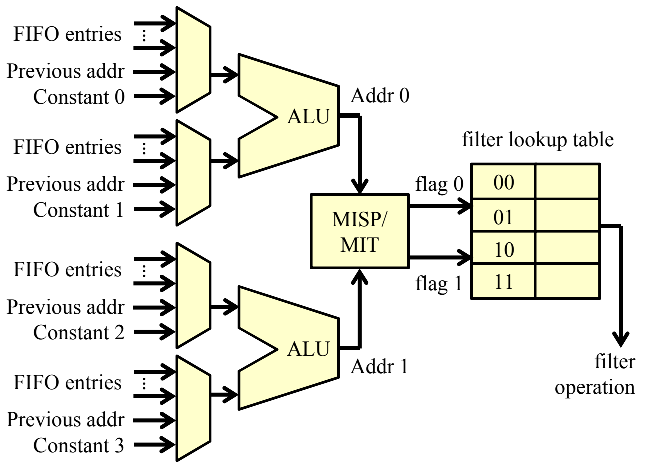

Fig. 10. Metadata filtering module (MFM).

marking certain metadata as valid or invalid with a 1-bit flag. However, in general the MIM simply sets or clears a bit based on a calculated address. We expect that for certain monitoring schemes, designers may be able to use the MIM in new ways such as using the MIT and MISP to directly express certain metadata. Section A3 of the Appendix shows how the MISP can be used to directly express the register file taint metadata used for a Dynamic Information Flow Tracking (DIFT) scheme.

C. Filtering Invalid Metadata

Given that certain metadata becomes invalidated by the metadata invalidation module, performing check and update monitoring operations based on the invalid metadata is not useful. Thus, we can drop these monitoring tasks. By skipping these invalid monitoring tasks, slack can be reserved for monitoring tasks which operate on valid metadata. Determining when a monitoring event can be filtered out in this manner is done by the metadata filtering module (MFM) which is shown in Figure 10.

We examined multiple monitoring schemes and found that most monitoring schemes read in up to two metadata, corresponding to the two input operands of an instruction, in order to perform an update. Thus, the MFM was designed with a pair of configurable metadata address generation units, similar to the ones used in the MIM. These address generation units calculate a pair of addresses which correspond to a pair of metadata invalidation flags which are read from the MIT and/or MISP. The two flag bits are then used to look up an entry in a lookup table that specifies whether to filter the event or not. Typically, if either of the source metadata operands are marked as invalid, then the monitoring task can be filtered. We must ensure no false positives occur due to filtering out these monitoring events. Thus, the entry in the lookup table can also inform the MIM of metadata that must be invalidated. Similar to the MIM, the monitor also configures the MFM address calculations and the lookup table during initialization.

D. Full Architecture

Figure 11 shows a block diagram of the complete architecture. Monitoring events from the main core are first enqueued in a FIFO. The events in the FIFO are dequeued and processed by the metadata filtering module. The MFM checks the MIT

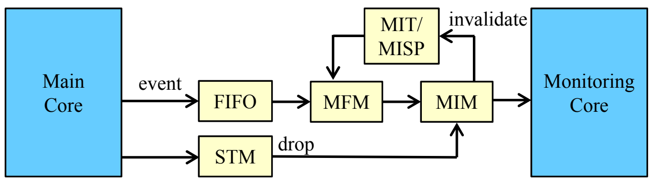

Fig. 11. Block diagram of full hardware architecture for opportunistic monitoring on hard real-time systems.

and/or MISP to decide whether the event should be filtered because of invalid metadata. If the event is not filtered, then the metadata invalidation module decides whether to drop the event based on the slack tracking module. If the event is dropped or filtered, then the MIM marks invalidation flags using the MIT/MISP. Instead, if the event is not dropped or filtered, then it is forwarded to the monitoring core to perform the monitoring task.

V. Evaluation

A. Methodology

We implemented our monitoring architecture for real-time systems by modifying the ARM version of the gem5 simulator [14] to support parallel hardware monitoring and our hardware optimizations for opportunistic monitoring. In order to explore the generality of the architecture for different monitors, we implemented the three different monitors: uninitialized memory check (UMC), return address check (RAC), and dynamic information flow tracking (DIFT). Uninitialized memory check was mentioned in Section II and seeks to detect loading from memory locations that are not initialized first. Return address check protects against certain security attacks, such as return-oriented programming [15], by checking that the address returned to after a function completes matches the address that originally called the function. Dynamic information flow tracking is another security monitoring scheme. DIFT attempts to detect when information from untrusted sources is used to affect the program control flow. The details of how each of these monitoring schemes works with our architecture is discussed in the Appendix. We tested our system using several benchmarks from the Mälardalen WCET benchmark suite [16].

We model the main and monitoring cores as 500 MHz in-order cores, each with 16 KB of L1 I/D-caches. The latency to main memory is 15 ns. This setup is similar to Freescale's i.MX353 processor which targets embedded, automotive, and industrial applications. For our experiments, we used a FIFO of 16 entries connecting the main and monitoring cores. The MISP was 16 entries matching the 16 registers found in the ARM architecture. The MIT was configured with 2 ways and 256 entries.

No static WCET analysis tools exist for the gem5 simulator. In order to estimate the WCET of tasks, we ran tasks several times on the gem5 simulator and took the worst-case observed execution time. Our WCET estimate is expected to be lower than those that a WCET analysis tool would generate since WCET analysis tools guarantee a conservative estimate. As a result, in our experiments the gap between actual execution times and our estimated WCET is lower than what we would expect with a WCET analysis tool. Therefore, the results we present for the amount of monitoring that can be done are less than what is expected when a strictly conservative WCET is used.

B. Amount of Monitoring Performed

Table I shows the number of monitoring events that are monitored, dropped, and filtered as a percentage of the total number of monitoring events. We find that a portion of monitoring can still be done without exceeding the main task's WCET. This is due to the dynamic slack that is gained during run time. On average, UMC can perform 17% of its monitoring tasks. RAC can perform 66% of its monitoring and DIFT can perform 31% of its monitoring. No results are shown for `insertsort` and `nsichneu` for RAC because these benchmarks do not make any function calls. For DIFT, we store the actual DIFT register file metadata in the MISP instead of invalidation flags. This optimization allows us to use the MIM and MFM to perform certain monitoring tasks (see Section A3 of the Appendix for details). These correct metadata updates are counted as "Monitored" in the statistics while events are counted as dropped or filtered only if they were due to insufficient slack.

As expected, a false positive never occurred in our experiments. However, false negatives can occur due to dropping monitoring tasks. Specifically, dropped or filtered check-type monitoring operations can result in false negatives. Table II shows the number of checks that are monitored as a percentage of the total checks. This acts as a measure of the coverage achieved by the monitors. The coverage for UMC is 15% on average and the coverage for RAC is 66% on average. The average coverage for DIFT is 59% which is much higher than the percentage of monitoring tasks that are not dropped or filtered. In fact, for some of the benchmarks, DIFT is able to achieve 100% coverage. This implies that only a portion of the monitoring operations performed by DIFT actually affect the checks. For example, DIFT propagates metadata on every ALU, load, and store instruction. However, only instructions that eventually propagate metadata to an indirect jump instruction affect the coverage.

For an underutilized system, if some headstart slack is given to a task initially, then the amount of monitoring that can be performed can be increased. As an example, Figure 12 shows how the coverage increases as we increase the headstart slack given to the main task for UMC. Figure 13 shows how the coverage varies with headstart slack for DIFT. Results for RAC are similar and can be found in Section B of the Appendix. The headstart slack is displayed as a percentage of the main task's WCET for each benchmark and is varied from 0% to 600%. With enough headstart slack, 100% of the monitoring is able to be performed. For DIFT, only benchmarks which were not able to reach 100% coverage with zero headstart slack are shown. `compress` and `statemate` do not reach 100% coverage across the varied range. This is not surprising as both had especially high WCET for performing monitoring without dropping (see Figure 7). With higher headstart slack, these benchmarks should also reach 100% coverage.

C. FPGA-based Monitor

Performing monitoring in software, although parallelized, can still incur high overheads since multiple instructions are needed to handle each monitoring event. One possible solution to improve the performance of monitoring while maintaining programmability is to use an FPGA-based monitor [7]. We model the FPGA-based monitor as being able to run at 250 MHz and handle up to one monitoring event each cycle. Note that this means that the FPGA-based monitor can process a monitoring event every two cycles of the main core which runs at 500 MHz. Table III shows the number of monitored, dropped, and filtered events with no headstart slack for this FPGA-based monitor. UMC and RAC are able to run 67% of their monitoring tasks on average, while DIFT is able to run 72% of its monitoring tasks on average. Table IV shows the coverage achieved by these monitoring schemes on an FPGA-based monitor. RAC shows similar numbers to processor-based monitoring because the number of calls and returns in these benchmarks is relatively small. However, for UMC and DIFT, we see that an FPGA-based monitor allows much more monitoring to be done without increasing the headstart slack. The coverage for UMC increases from 15% to 62% while the coverage for DIFT increases from 59% to 86%.

Figure 14 shows how the coverage for UMC varies as we increase the headstart slack from 0% to 10% for an FPGA-based monitor. Figure 15 shows this data for DIFT and results for RAC can be found in Section B of the Appendix. We can see that for a high-performance FPGA-based monitor, with a small amount of slack, we are able to achieve 100% monitoring for almost all benchmarks while guaranteeing the main task's execution time does not exceed its WCET.

D. Area and Power Overheads

Adding the dropping hardware in order to enable adjustable overheads adds overheads in terms of area and power. We use McPAT [17] 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 1.96 mm^2 and the peak power usage as 152.9 mW averaged across all benchmarks. The average runtime power usage was 71.6 mW. These area and power numbers consist of the core and L1 cache, but do not include memory controllers and other peripherals. The power numbers include dynamic as well as static (leakage) power. For the dropping hardware, the ALUs, MISP, MIT, and configuration tables are modeled using the corresponding objects in McPAT. We note that this is only a rough area and power result 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.

An additional 0.132 mm^2 of silicon area is needed, an increase of 7% of the main core area. Table V shows the peak and runtime power overheads with zero headstart slack for both a processor-based monitor as well as an FPGA-based monitor. The peak power is 5-17 mW, which is 3-11% of the main core's peak power usage. The average runtime power is 5-9 mW, corresponding to 7-12% of the main core's runtime power. For UMC and DIFT, the core-based monitor has lower power usage due to more monitoring events being filtered out that do not require invalidation. This reduces the activity of the Metadata Invalidation Module which reduces the power usage.

VI. Related Work

The architecture we present in this paper is applicable to a wide range of parallel hardware run-time monitoring

TABLE I. NUMBER OF MONITORED, DROPPED, AND FILTERED MONITORING EVENTS AS A PERCENTAGE OF THE TOTAL NUMBER OF MONITORING EVENTS. THESE PERCENTAGES ARE SHOWN FOR ZERO HEADSTART SLACK.

Monitor	Type	Benchmark								Average
		compress	crc	edn	fir	insertsort	jfdc	nsichneu	statemate	
UMC	Monitored	11.9%	59.0%	4.4%	28.4%	15.9%	16.2%	0.0%	3.3%	17.4%
	Dropped	29.4%	9.4%	5.9%	3.9%	45.6%	43.4%	0.13%	33.4%	21.4%
	Filtered	58.7%	31.6%	89.7%	67.7%	38.6%	40.4%	99.9%	63.3%	61.2%
RAC	Monitored	93.8%	50.0%	83.3%	90.9%	-	0.0%	-	80.0%	66.3%
	Dropped	3.1%	25.0%	8.3%	4.5%	-	50.0%	-	10.0%	16.8%
	Filtered	3.1%	25.0%	8.3%	4.5%	-	50.0%	-	10.0%	16.8%
DIFT	Monitored	11.0%	92.2%	26.7%	22.0%	61.8%	22.1%	4.7%	10.6%	31.4%
	Dropped	26.3%	0.89%	19.6%	18.4%	16.6%	12.6%	57.5%	65.7%	27.2%
	Filtered	62.8%	6.9%	53.7%	59.6%	21.6%	65.2%	37.8%	23.6%	41.4%

TABLE II. PERCENTAGE OF CHECKS THAT ARE NOT DROPPED OR FILTERED. THESE PERCENTAGES ARE SHOWN FOR ZERO HEADSTART SLACK.

Monitor	Benchmark								Average
	compress	crc	edn	fir	insertsort	jfdc	nsichneu	statemate	
UMC	0.0%	59.9%	0.41%	27.8%	10.9%	16.3%	0.0%	0.41%	14.5%
RAC	93.8%	50.0%	83.3%	90.9%	-	0.0%	-	80.0%	66.3%
DIFT	3.8%	66.7%	44.4%	16.7%	100.0%	100.0%	100.0%	43.3%	59.4%

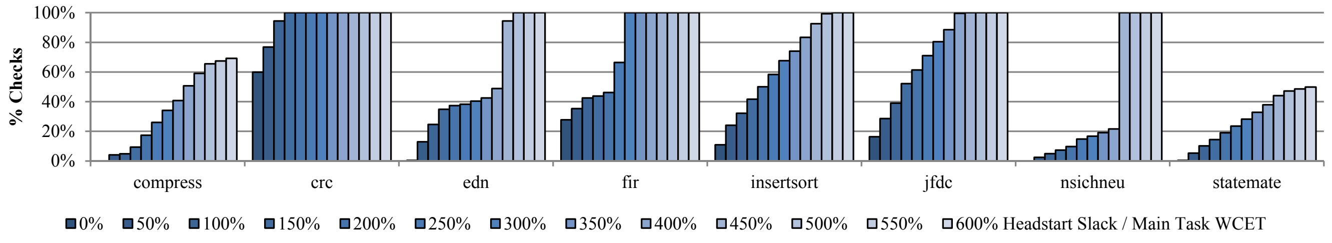

Fig. 12. Percentage of checks performed as headstart slack is varied for UMC. Headstart slack is shown normalized to the main task's WCET.

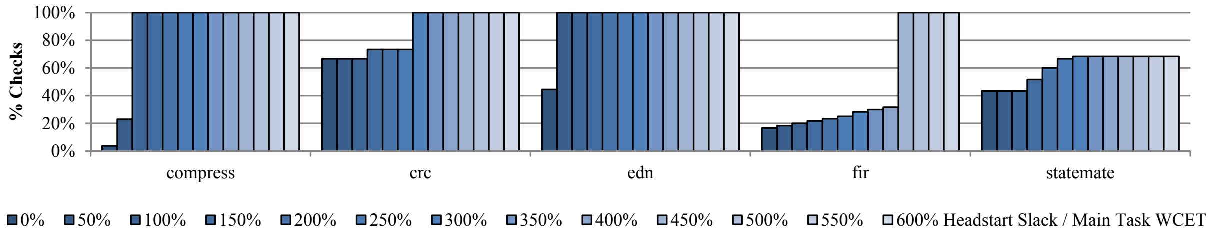

Fig. 13. Percentage of checks performed as headstart slack is varied for DIFT. Headstart slack is shown normalized to the main task's WCET.

techniques. We briefly mention some recent platforms as examples. For example, INDRA [18] uses a checker core to monitor coarse-grained events on a computation core such as function call/return, code origin inspection, and control flow inspection. Nagarajan et al. studied implementing DIFT on a multi-core system [19]. Chen et al. proposed hardware acceleration techniques for multi-core systems and showed that a set of parallel monitoring techniques for security and software debugging can be realized with low performance overheads (tens of percents) [6]. The FlexCore [7] and Harmoni [8] architectures showed that parallel monitoring can be made even more efficient by implementing monitors on reconfigurable hardware. SecureCore [20] is a monitoring scheme targeted specifically at real-time systems which attempts to detect code injection attacks by detecting anomalous timing behavior. However, the architecture assumes enough buffering so that the timing behavior of the main task is not affected. Overall, these previous studies demonstrate that parallel monitoring can be used to significantly improve system security and reliability with minimal overheads.

There have been several projects that have looked into performing limited monitoring. Testudo [21] performs limited monitoring across many users in order to limit the performance impact of monitoring. The Quality Virtual Machine (QVM) is a modification of the Java Virtual Machine that supports run-time monitoring with controllable overheads [22]. QVM enables and disables monitoring in an attempt to match the specified target overhead. Similarly, Huang et al. created a framework for controlling the overheads of software-based monitoring [23]. These projects provide no strict guarantees on the performance impact and thus are not applicable to hard real-time systems. In contrast, our architecture is able to provide a strict guarantee on the worst-case overheads. In addition, these projects enable and disable monitoring at a coarse granularity in order to guarantee that false positives do not occur. Instead, we have presented an invalidation-based mechanism to prevent false positives. This allows the decision of whether to perform monitoring to be extremely fine-grained (i.e., per monitoring event).

As mentioned earlier, there is existing work for statically estimating the WCET of monitoring using an MILP-based method [9]. In general there has been a multitude of work in

TABLE III. Number of monitored, dropped, and filtered monitoring events as a percentage of the total number of monitoring events for an FPGA-based monitor. These percentages are shown for zero headstart slack.

Monitor	Type	Benchmark								Average
		`compress`	`crc`	`edn`	`fir`	`insertsort`	`jfdc`	`nsichneu`	`statemate`	
UMC	Monitored	29.6%	85.9%	97.3%	83.3%	91.4%	88.7%	14.5%	45.7%	67.1%
	Dropped	17.4%	3.3%	0.05%	2.7%	3.0%	3.1%	0.10%	8.5%	4.8%
	Filtered	53.0%	10.8%	2.7%	14.0%	5.6%	8.3%	85.4%	45.8%	28.2%
RAC	Monitored	94.8%	50.0%	83.3%	95.5%	-	0.0%	-	81.0%	67.4%
	Dropped	3.1%	25.0%	8.3%	4.5%	-	50.0%	-	10.0%	16.8%
	Filtered	2.1%	25.0%	8.3%	0.0%	-	50.0%	-	9.0%	15.7%
DIFT	Monitored	40.6%	97.6%	64.7%	50.6%	97.9%	75.8%	96.6%	51.5%	71.9%
	Dropped	8.8%	0.06%	13.6%	0.91%	0.97%	0.95%	2.1%	33.6%	7.6%
	Filtered	50.6%	2.3%	21.7%	48.5%	1.1%	23.3%	1.3%	14.9%	20.5%

TABLE IV. Percentage of checks that are not dropped or filtered for an FPGA-based monitor. These percentages are shown for zero headstart slack.

Monitor	Benchmark								Average
	`compress`	`crc`	`edn`	`fir`	`insertsort`	`jfdc`	`nsichneu`	`statemate`	
UMC	9.9%	87.8%	97.1%	85.1%	86.1%	84.8%	14.5%	28.3%	61.7%
RAC	95.8%	50.0%	83.3%	100.0%	-	0.0%	-	82.0%	68.5%
DIFT	3.8%	93.3%	100.0%	100.0%	100.0%	100.0%	100.0%	88.3%	85.7%

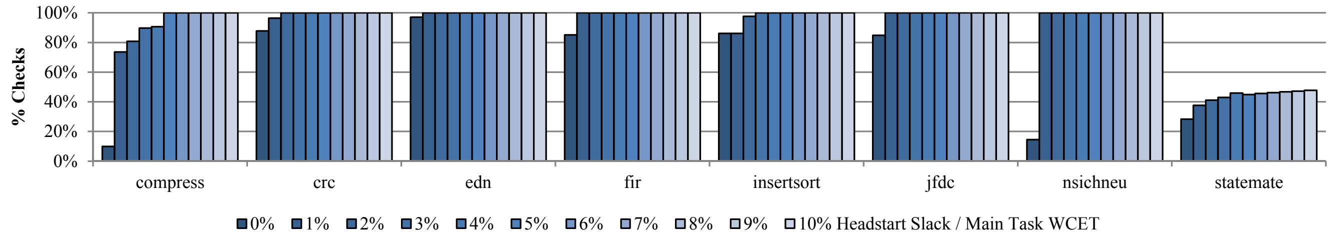

Fig. 14. Percentage of checks performed as headstart slack is varied for UMC on an FPGA-based monitor. Headstart slack is shown normalized to the main task's WCET.

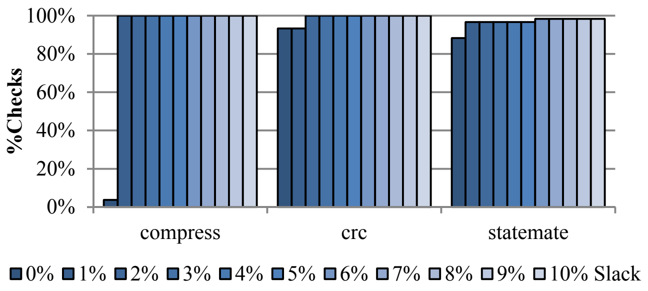

Fig. 15. Percentage of checks performed as headstart slack is varied for DIFT on an FPGA-based monitor. Headstart slack is shown normalized to the main task's WCET.

TABLE V. Average power overheads for dropping hardware at zero headstart slack. Percentages in parentheses are normalized to the main core's power usage.

Monitor		Peak Power [mW]	Runtime Power [mW]
Processor	UMC	4.9 (3.2%)	4.7 (6.6%)
	RAC	4.7 (3.1%)	4.7 (6.6%)
	DIFT	5.0 (3.3%)	4.8 (6.7%)
FPGA	UMC	16.7 (10.9%)	8.5 (11.9%)
	RAC	4.7 (3.1%)	4.7 (6.6%)
	DIFT	13.0 (8.5%)	8.8 (12.3%)

analyzing WCET [11], including for multi-core architectures [24], [25]. However, since these methods are static, they can be overly conservative and require redoing the timing analysis when any changes are made to the main or monitoring tasks. Instead, the architecture we presented in this paper takes advantage of run-time behavior to perform monitoring while maintaining timing guarantees.

There has also been a host of work on hardware architectures for real-time systems. The VISA architecture [26] uses dynamic slack to run in a faster but difficult to analyze mode. It switches to a simpler mode under which timing analysis and WCET is calculated when there is not enough slack. We use a similar method to keep track of slack. However, we use slack for monitoring rather than for running under a different mode. The PRET architecture [27] seeks to have a processor design that is both high performance and easy to analyze in terms of timing. Paolieri et al. proposed an analyzable multi-core hardware architecture for hard real-time systems [28]. These architectures all design hardware with real-time system requirements in mind, but ours is the first work that we know of that designs a hardware architecture for parallel hardware run-time monitoring on real-time systems.

VII. Conclusion

Run-time monitoring techniques are attractive for improving the security and reliability of systems. However, applying these techniques to real-time systems has a large impact on the WCET of programs. In this paper, we have presented an opportunistic monitoring framework that uses dynamic slack to perform monitoring while minimizing the WCET impact on programs. This is done by dropping monitoring events when there is insufficient slack and instead running a dropping task that invalidates metadata in order to prevent false positives. With hardware optimizations, we are able to eliminate this WCET impact entirely. We show that, for three different tested monitoring schemes, an average of 15-66% of monitoring checks, depending on the monitoring scheme, can be performed with no impact on the main task's WCET. For an FPGA-based monitor, a range of 62-86% of monitoring checks, on average, depending on the monitoring scheme, can be performed with no impact on WCET.

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] 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.

[2] 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.

[3] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic, "Effective Memory Protection Using Dynamic Tainting," in *Proceedings of the 22nd International Conference on Automated Software Engineering*, 2007.

[4] 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.

[5] *Intel Architecture Instruction Set Extensions Programming Reference*, Intel, 2013. [Online]. Available: http://download-software.intel.com/sites/default/files/319433-015.pdf

[6] 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.

[7] 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.

[8] 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.

[9] D. Lo and G. E. Suh, "Worst-case Execution Time Analysis for Parallel Run-time Monitoring," in *Proceedings of the 49th Design Automation Conference (DAC)*, 2012.

[10] 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.

[11] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems*, 2008.

[12] L. Sha, "Using Simplicity to Control Complexity," *IEEE Software*, 2001.

[13] A. Anantaraman, K. Seth, E. Rotenberg, and F. Mueller, "Enforcing Safety of Real-Time Schedules on Contemporary Processors using a Virtual Simple Architecture (VISA)," in *Proceedings of the 25th International Real-Time Systems Symposium*, 2004.

[14] 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," *SIGARCH Computer Architecture News*, 2011.

[15] H. Schacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *Proceedings of the 14th Conference on Computer and Communications Security*, 2007.

[16] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET Benchmarks – Past, Present and Future," in *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

[17] 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.

[18] W. Shi, H.-H. S. Lee, L. Falk, and M. Ghosh, "INDRA: An Integrated Framework for Dependable and Revivable Architectures Using Multi-core Processors," in *Proceedings of the 33rd International Symposium on Computer Architecture*, 2006.

[19] 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.

[20] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha, "SecureCore: A Multicore-based Intrusion Detection Architecture for Real-Time Embedded Systems," in *Proceedings of the 19th Real-Time and Embedded Technology and Applications Symposium*, 2013.

[21] 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.

[22] 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 Languages and Applications*, 2008.

[23] 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.

[24] S. Chattopadhyay, C. L. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, "A Unified WCET Analysis Framework for Multi-core Platforms," in *Proceedings of the 18th Real Time and Embedded Technology and Applications Symposium*, 2012.

[25] M. Lv, W. Yi, N. Guan, and G. Yu, "Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software," in *Proceedings of the 31st Real-Time Systems Symposium*, 2010.

[26] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller, "Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-Time Systems," in *Proceedings of the 30th International Symposium on Computer Architecture*, 2003.

[27] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee, "A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance," in *Proceedings of the 30th International Conference on Computer Design*, 2012.

[28] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems," in *Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

Appendix

A. Monitoring Extensions

In this section, we detail how our architecture works for the three different monitoring schemes we evaluated: uninitialized memory check (UMC), return address check (RAC), and dynamic information flow tracking (DIFT). Table VI shows a summary of the monitoring tasks for these monitoring schemes and Table VII shows how the MIM and MFM operate.

1) Uninitialized Memory Check: As described in Section II, uninitialized memory check (UMC) is a monitoring scheme that checks that memory is written to before it is read from. Table VI summarizes the monitoring tasks for UMC.

Table VII shows the operations of the MIM and MFM. The monitoring task performed on a load instruction is a check operation. Thus, it can simply be skipped without causing any false positives. However, the monitoring task on a store updates metadata. On a dropped store event, the MIM calculates the metadata address and sets an invalidation flag corresponding to this address in the MIT to indicate that the metadata for this word is invalid.

In terms of using the metadata filtering module, on a load, if the metadata is invalid, then performing the check is useless. Thus, if the filtering module detects that the metadata invalidation flag corresponding to the memory access address is invalid, then the monitoring event is dropped and no operation is performed. For a store event, the source operand of the operation is a register value which is always valid. Thus, the MFM never filters a store event. Instead, if the store event is not dropped, the monitor will set the metadata bit and can clear the corresponding invalidation flag.

2) Return Address Check: Several security attacks, such as return-oriented programming (ROP) [15], attempt to manipulate the return address of a function so that the control flow of a program is diverted. One method to prevent these types of attacks is to use monitoring to save the return address on a call to a function and to check on the return that the correct return address is used. We refer to this type of monitoring scheme as a return address check (RAC). On a call instruction, RAC pushes the correct return address for the function onto a stack data structure. On a return instruction, RAC pops an address of the stack and checks that this saved address matches the address the main task is returning to (Table VI).

In the case of UMC, the metadata that we cared about was a function of information from the monitoring event. For RAC, the metadata entry that we care about is based on the current stack pointer for the return address stack that the monitor maintains. The MIM maintains the previous metadata address used in a register and we can use this register to maintain the stack pointer between the monitoring task and the MIM. The monitoring task updates this register as necessary to ensure that the MIM has the correct metadata stack pointer. On a call instruction that needs to be dropped, the MIM marks the metadata at the current stack pointer (i.e., the previous metadata address) as invalid and increments the stack pointer. On a return, the MIM decrements the stack pointer and simply skips the check operation (Table VII).

On a call instruction, the source of our monitoring operation is the return address from the monitoring event. This is always valid, so call instructions are never processed by the MFM. On a return instruction, if the stored return address is marked as invalid, then the check can be skipped. Thus, if the MFM detects that the stored return address is invalid, then the stack pointer will be decremented but no check will occur, similar to the invalidation case.

3) Dynamic Information Flow Tracking: Dynamic information flow tracking (DIFT) [1] is a security monitoring scheme that seeks to detect when information from untrusted sources is used to affect the control flow. In its simplest form, DIFT keeps a 1-bit metadata tag for each memory location and each register, indicating tainted or untainted. Multi-bit versions of DIFT have been proposed to track more detailed semantic information about data. When data is read from an untrusted source, it is marked as tainted. As this data is used for other operations, the results of these operations are also marked as tainted. On an indirect jump, the taint bit of the register used is checked. If the register is found to be tainted, then an error is detected. These monitoring task operations are summarized in Table VI.

In the case of DIFT, since the registers have associated metadata and these are used very often, we use the MISP to avoid aliasing. Also, in this discussion, we are considering a 1-bit taint, so the MISP can be used to store the exact taint value, rather than an invalidation flag. This shows some of the flexibility in how these hardware structures can be used for specific monitoring schemes. For more complicated multi-bit DIFT schemes, the MISP would be used to store invalidation flags as before. For ALU and load instructions which have a register as a destination, the MIM will clear the register's taint in the MISP on a dropped monitoring event. This is safe because an error is only raised for a tainted data value. On a store instruction, which writes to memory as a destination, the MIM invalidates the associated memory address in the MIT, similar to how UMC operates. Finally, on an indirect jump, only a check operation is performed. Thus, this can be dropped without any extra invalidation.

The MFM operates similar to the UMC case for some of these monitoring events (Table VII). For an indirect jump, if the metadata is found to be invalid, then the event is filtered and no check is performed. For a load instruction, if the loaded data has its metadata marked as invalid, then the MFM will clear the taint bit for the target register, as in the invalidation case. For a store instruction, the source is from a register, which always has a valid taint bit in the MISP, so these are never filtered out. The most interesting case is for ALU instructions. The MFM is able to read the taint bits from the MISP and then signal to the MIM to write to the MISP, so we can actually perform the entire monitoring task in hardware. The MFM reads the taint bits from the MISP for both source registers. These are fed into the filter lookup table (see Figure 10) which is configured to signal the MIM to set or clear the taint bit in the MISP for the destination register depending on the read taint bits.

B. Coverage of Return Address Check

This section shows how the coverage varies as headstart slack is changed for RAC. Figure 16 shows the coverage for a processor core implementation of RAC while Figure 17 shows the coverage for an FPGA-based monitor. `insertsort` and

TABLE VI. MONITORING TASKS FOR UMC, RAC, AND DIFT.

Monitor	Event	Task	Operation
UMC	Store	Set metadata tag (data initialized)	Update
	Load	Check that metadata tag is set	Check
RAC	Call	Push return address	Update
	Return	Pop return address and check	Update & Check
DIFT	ALU	Set the output tag as the OR of the input tags	Update
	Load	Set the register tag as the tag of the loaded data	Update
	Store	Set the memory tag as the register tag	Update
	Indirect jump	Check that tag is not set (data is untainted)	Check

TABLE VII. OPERATION OF THE MIM AND MFM FOR UMC, RAC, AND DIFT.

Monitor	Event	Invalidation	Filter Condition	Filter Operation
UMC	Store	Set invalidation flag	Never	-
	Load	Do nothing	Invalid metadata	Do nothing
RAC	Call	Invalidate metadata, increment metadata address	Never	-
	Return	Decrement metadata address	Invalid metadata	Decrement stack pointer
DIFT	ALU	Clear taint in MISP	Always	Perform propagation
	Load	Clear taint in MISP	Invalid metadata	Clear taint in MISP
	Store	Invalidate metadata	Never	-
	Indirect jump	Do nothing	Invalid metadata	Do nothing

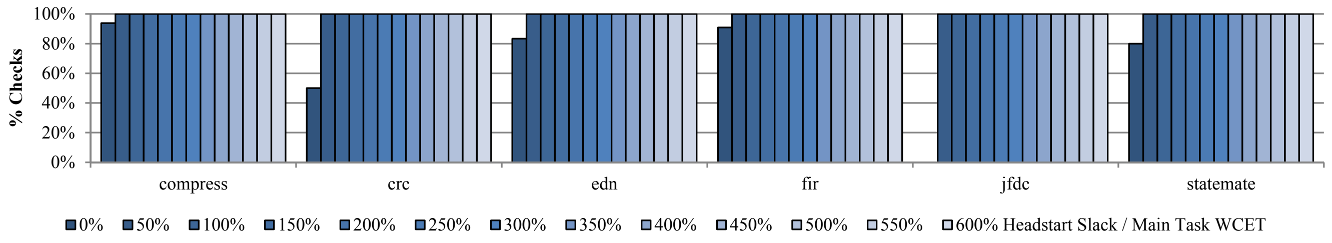

Fig. 16. Percentage of checks performed as headstart slack is varied for RAC implemented on a processor core. Headstart slack is shown normalized to the main task's WCET.

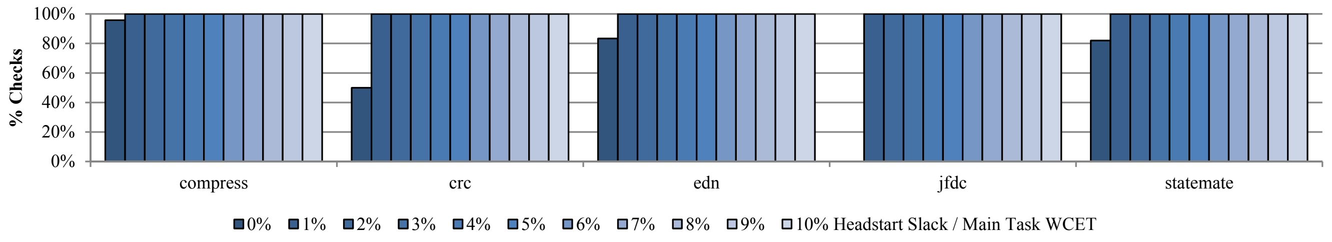

Fig. 17. Percentage of checks performed as headstart slack is varied for RAC on an FPGA-based monitor. Headstart slack is shown normalized to the main task's WCET.

`nsichneu` were omitted as they do not perform function calls. `fir` is not shown for the FPGA-based monitor because it already achieves 100% coverage with zero headstart slack. In both cases, we see that with a small amount of headstart slack, RAC is able to reach 100% coverage for all benchmarks.