Fast Development of Hardware-Based Run-Time Monitors Through Architecture Framework and High-Level Synthesis

Mohamed Ismail and G. Edward Suh
School of Electrical and Computer Engineering
Cornell University
Ithaca, NY, USA
{mii5,gs272}@cornell.edu

***Abstract*—Recent work has shown that hardware-based run-time monitoring techniques can significantly enhance security and reliability of computing systems with minimal performance and energy overheads. However, the cost and time for implementing such a hardware-based mechanism presents a major challenge in deploying the run-time monitoring techniques in real systems. This paper addresses this design complexity problem through a common architecture framework and high-level synthesis. Similar to customizable processors such as Tensilica Xtensa where designers only need to write a small piece of code that describes a custom instruction, our framework enables designers to only specify monitoring operations. The framework provides common functions such as collecting a trace of execution, maintaining meta-data, and interfacing with software. To further reduce the design complexity, we also explore using a high-level synthesis tool (Cadence C-to-Silicon) so that hardware monitors can be described in a high-level language (SystemC) instead of in RTL such as Verilog and VHDL. To evaluate our approach, we implemented a set of monitors including soft-error checking, uninitialized memory checking, dynamic information flow tracking, and array boundary checking in our framework. Our results suggest that our monitor framework can greatly reduce the amount of code that needs to be specified for each extension and the high-level synthesis can achieve comparable area, performance, and power consumption to handwritten RTL.**

I. INTRODUCTION

As computing devices permeate more industries and as their use becomes more widespread, security and reliability of those devices become increasingly important. Run-time monitoring of program execution serves as an effective way to ensure a wide range of security and reliability properties. For example, Dynamic Information Flow Tracking (DIFT) protects the integrity of a system by limiting where inputs from untrusted sources can be used. More specifically, DIFT records and tracks values from untrusted inputs and ensures that these values can never used in security-critical operations such as a control transfer. It has been shown to be quite effective in detecting a large class of common software attacks [1], [2], [3]. Other run-time monitors such as memory protection [4], array bounds checking [5], [6], and soft error detection [7] can also protect a system against a wide range of vulnerabilities.

Unfortunately, many run-time monitoring applications perform poorly on existing processors. Software monitoring approaches often incur significant slowdowns even with an additional processing core for parallel execution. In addition, the 32-bit (or 64-bit) data-path is wasteful for operations that require single bits for checks and bookkeeping. A software implementation for DIFT on a single core is reported to have an average slowdown of 3.6 times even with aggressive optimizations [8]. On the other hand, hardware modules [9], [10] are efficient and result in much lower overheads. In order to implement such extensions, however, the designer needs to make significant modifications to the processor pipeline to forward the proper values and handle any exception that results from the monitor. Such modifications are time-consuming and requires additional validation efforts to ensure correct and efficient functioning of the monitor itself.

This paper proposes an architectural framework that allows designers to quickly prototype and develop run-time monitors and other bookkeeping hardware. Instead of writing thousands of lines of code, the proposed framework provides common interfaces and infrastructures for typical run-time monitors so that a designer only needs to define monitor-specific operations in a much smaller piece of code, often only a few hundred lines of code. The code will be simpler and less error-prone, allowing companies to save in design and verification costs. Therefore, the framework enables companies to secure their computing devices through hardware monitors with minimal time-to-market delays. The overall approach of this framework is similar to that of customizable (ASIP) processors such as Tensilica Xtensa, where custom instructions can be quickly added to a processor through the use of a common architecture framework and tools.

The experimental results suggest that our monitoring framework can save much of the redundancy in incorporating run-time monitors in hardware. We find that the designer only needs to make modifications to 0.62% to 2.75% of the code in contrast to 31.44% to 36.39% if the extensions are

This work was partially supported by the National Science Foundation grants CNS-0746913 and CNS-0708788, 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.

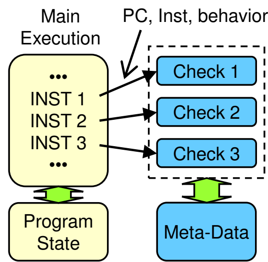


Fig. 1. Computation model for parallel monitoring and bookkeeping.

implemented without the common framework. We additionally test the possibility of using high-level synthesis to generate these monitoring modules. For our test cases, we find that SystemC requires more lines of code to express the functionality of a module when compared to a single-stage handwritten RTL version. However, we find that high-level synthesis can generate modules with less area and power overheads, and provide an additional opportunity of easy design exploration with little or no changes to the code.

The rest of the paper is organized as follows. Section II describes a co-processor model for parallel monitoring and bookkeeping and shows how example extensions map to the model. Section III presents an architecture framework for fast development of run-time monitors. Section IV studies the hardware complexity and the overheads of the framework and compares high-level synthesis with handwritten RTL. Section V discusses related work, and Section VI concludes the paper.

II. Instruction-Grained Run-Time Monitoring

This section presents the computation model for the fine-grained run-time monitoring and bookkeeping that our framework is designed to support and describes common characteristics of such co-processing operations. In the following discussion, we use the term "co-processor" to refer to a monitoring and bookkeeping extension.

A. Co-Processing Model

Figure 1 shows the high-level model of how fine-grained monitoring and bookkeeping techniques typically work. In the figure, dark (blue) blocks represent co-processing and light (yellow) blocks represent the main computation. A co-processor maintains its own meta-data, which are often disjoint from the program state, to keep track of the history of computation by the main core. At run-time, the co-processor monitors the execution of the main core at an instruction granularity, updates its meta-data for bookkeeping, and checks certain properties of the main computation. If a check fails, the co-processor may raise an exception. Conceptually, the co-processor observes a trace of all or a subset of instructions and performs its operations on each forwarded instruction. The main core can also communicate with the co-processor with explicit instructions to either configure the co-processor or read information from it.

In general, the run-time monitoring and bookkeeping extensions can be characterized by its meta-data, transparent operations, and software visible operations. Here, we briefly summarize the common characteristics and their implications for the architecture design.

- *Meta-data*: The extensions often need meta-data for both registers and memory. Therefore, the co-processor need to support a memory subsystem for meta-data.
- *Operations*: The monitoring or bookkeeping is often *fine-grained*. For many applications, run-time checks and meta-data updates happen quite frequently, possibly for every instruction on the main core. Therefore, pure software implementations often incur a significant slowdown. At the same time, the operations are largely *decoupled* from the main computation and can be performed in parallel to the main core. The main computation is typically independent from the monitoring extension unless there is an exception. Finally, the extension usually performs *bit operations* rather than 32-bit word operations. Therefore, the operations are a poor match for a regular computing core.
- *Software interfaces*: The co-processor needs to be able to raise an exception and communicate with the main core with explicit instructions from the core. The instructions may read/write configuration registers on the co-processor and/or perform custom operations for each extension. The exception and the explicit instructions are usually infrequent and do not have to be fast.

B. Example Monitoring Extensions

This subsection presents a set of monitoring and bookkeeping extensions as examples and shows how those extensions map to our co-processing model. Table I summarizes the operations of four example extensions: UMC, DIFT, BC, and SEC.

a) Uninitialized Memory Check (UMC): Uninitialized Memory Check (UMC) is a simple mechanism that is widely used for software debugging to ensure that a memory location is initialized (written) before a read. For each memory location, the UMC mechanism maintains a one-bit tag that represents whether the location is initialized or not. The tag is set after a write to the corresponding memory location and cleared with an instruction from the main core when the corresponding memory is de-allocated. On a load operation, the mechanism checks the tag and raises an exception if the memory location is not initialized.

b) Dynamic Information Flow Tracking (DIFT): Dynamic Information Flow Tracking (DIFT) detects software attacks by tracking potentially malicious values from I/O and checks their uses. DIFT can detect low-level exploits such as buffer overflows and format string attacks [1], [13], [2], without any modifications to executables, and detect high-level attacks such as SQL injections and cross-site scripting with simple application-level checks [3]. For DIFT, a co-processor needs meta-data (called taint tags) to indicate the source of each value in registers and memory, and explicit instructions so that the OS on the main core can set or clear

Extension	Meta-Data	Transparent Operations	SW Visible Operations
UMC [11]	1. 1-bit tag per word in memory.	1. Set the tag on a store. 2. Check the tag on a load.	1. Clear tags on a de-allocation. 2. Exception when a tag check fails.
DIFT [1]	1. 1-bit tag per register. 2. 1-bit tag per word in memory.	1. Propagate tags on ALU/load/store. 2. Check tags on a control transfer.	1. Set tags for values from I/O. 2. Clear tags on a declassification. 3. Set a security policy register. 4. Exception when a tag check fails.
BC [6]	1. 4-bit tag per register. 2. 8-bit tag per word in memory.	1. Propagate tags on ALU/load/store. 2. Check a pointer tag (register) with a memory tag on a load/store.	1. Set reg/mem tags on array allocation. 2. Clear tags on a de-allocation. 3. Exception when a tag check fails.
SEC [12], [7]		1. Check an ALU operation.	1. Exception when a check fails.

TABLE I

EXAMPLE CO-PROCESSING EXTENSIONS. UMC: UNINITIALIZED MEMORY CHECK, DIFT: DYNAMIC INFORMATION FLOW TRACKING, BC: ARRAY BOUND CHECKING, SEC: SOFT ERROR CHECKING.

those taint tags. For ALU, load, and store operations, the co-processor propagates taint tags from the source operand(s) to the destination; an OR operation of the source tag bits determine the destination tag. On security critical operations such as indirect jumps, the co-processor checks the tag and raises an exception if tainted values are used.

c) Array Bound Check (BC): An array bound check (BC) is a popular way to detect spatial memory errors such as buffer overflows, which account for a large portion of software errors and vulnerabilities in unsafe languages such as C. For BC, a co-processor keeps meta-data that encode the array bounds for a pointer and checks if each memory access is within the bounds. The array bounds can be expressed in various ways including an object table [14], [15], base and bound addresses [16], [5], or color tags [6], [17]. In our prototype, we encode bounds by assigning a color tag to each pointer and memory location. On a memory allocation, a program marks the resulting pointer and the corresponding memory locations with an identical tag using special instructions. For each memory access, the co-processor checks whether the pointer tag matches the memory location tag, and raises an exception if they do not match.

d) Soft Error Check (SEC): Recently there have been significant efforts to develop architectural techniques to detect hardware errors such as a transient bit-flip. In a high-level, these techniques either re-execute each instruction in parallel [12] or check simpler checksums on each operation [7]. These soft-error checks can be easily mapped to a monitor extension. As an example, consider a simple checker that verifies the result of each ALU operation by computing checksums as proposed by Argus [7]. A co-processor performs the checksum operation on each ALU instruction using the source and result values from the main core, and raises an exception if the check fails.

e) Other Extensions: We believe that the co-processing model will be applicable to a large class of hardware extensions that perform monitoring and/or bookkeeping operations in parallel to the main computation. As an example, the co-processing model can support simple profiling applications such as custom performance monitors and detailed analysis of software characteristics. The co-processing model also supports various techniques to enhance software security and reliability, including fine-grained memory protection [4], debugging support [18], checkpointing [19], and others. Parallel bookkeeping can also provide an efficient support for high-level language features such as garbage collection [20].

III. COMMON INTERFACE ARCHITECTURE DESIGN

This section describes an architecture framework for run-time monitors and bookkeeping extensions. The description focuses on how the architecture is designed to enable fast development of the extensions in an efficient manner by exploiting the common characteristics discussed in the previous section.

A. Scope and Design Goals

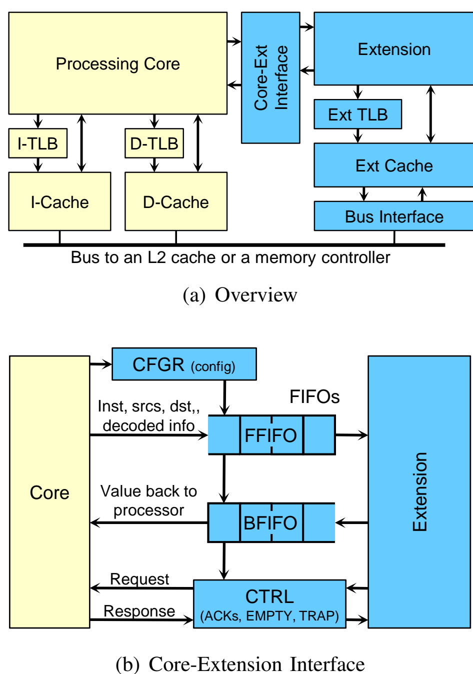
This paper illustrates the overall architecture in the context of a single-issue processor without multi-threading. While the general architecture is also applicable to super-scalar processors, further performance optimizations may be necessary for high-performance processors. Multi-threading introduces a coherence issue between program data and meta-data. This paper does not discuss this issue in detail because it is common for many run-time monitoring techniques with meta-data and has been studied already [10], [21].

To be useful in practice, the proposed architecture framework needs to be flexible, simple, and have minimal overheads. The following list summarizes our main design goals:

- *Flexible*: The framework should support a broad range of run-time monitoring and bookkeeping extensions so that the designer does not need to make changes to the interface in order to support a new extension.
- *Simple*: The framework should enable the designer to write extensions with a small amount of code. The designer should save time by focusing only on the functionality and correctness of the extension and not the interface.
- *Efficient*: The added extension and interface should not degrade the performance of the processor when not in use. The architecture should also enable run-time monitoring with minimal overheads.

B. Architecture Overview

Figure 2(a) shows the high-level block diagram of the co-processor architecture with an extension. The yellow (light) rectangles represent components in traditional microprocessors and the blue (dark) rectangles represent the new components necessary to support the interface. In a high-level, the architecture closely represents the co-processing model described in the previous section. The main processing core forwards its execution trace for run-time monitoring or bookkeeping



(a) Overview

(b) Core-Extension Interface

Fig. 2. Co-processor architecture block diagrams.

through a core-extension interface. The extension retrieves the data through the interface and performs some monitoring or bookkeeping. The extension also has access to its own TLB (if virtual memory is supported) and cache for meta-data storage. The L2 cache and main memory is shared with the processing core. The core-extension interface can also be used to send data back to the processor.

The architecture is carefully designed to exploit the common characteristics of run-time monitoring operations without restricting the specifics of the monitoring operations. A large subset of signals which may be used by monitoring extensions are forwarded through the interface allowing the designer to pick and choose which signals to use. To eliminate repeating logic, the interface forwards decoded signals along with the original instructions. The synthesis tool can synthesize out any unused signals, thereby reducing the overhead of forwarding unnecessary signals. In many cases, the extension does not need data for every instruction executed by the processor. The core-extension interface can be easily modified to filter out unnecessary instructions, enabling the extension to work on a single instruction for multiple cycles.

In addition to the core-extension interface, there are other commonly used modules which the designer does not need to redesign. Most extensions use and store meta-data, so memory structures including the TLB and cache are already provided as input and output ports to the extension. The designer can just use the I/O pins in order to store and retrieve data from memory. In order to match possible data-size mismatches between the extension and the memory, the extension can be efficiently optimized for any bit-width with parametrized data widths for reading and writing to the cache.

C. Co-Processor Interface

The extension communicates with the main processor through a set of FIFO interfaces as shown in Figure 2(b). The FIFOs are connected to/from the commit stage of the main core. The core-to-extension interface works to enable fine-grained instruction communication between the core and the extension. The processing core sends its execution trace to the co-processor using the FIFO interface, so that the extension can perform monitoring or bookkeeping operations on each forwarded instruction.

A forward FIFO sends a trace of instructions, which are completed and ready to commit, in the program order. A FIFO packet contains fairly comprehensive information, including a program counter, source and destination register values, ALU results, condition codes, and branch outcome. The FIFO packet also includes decoded instruction fields such as an op-code, source register numbers, and a destination register number, in order to reduce duplicate logic in the extension.

A forwarding configuration register (CFGR) specifies how a forward FIFO handles each instruction type[1]. For example, the CFGR can be configured to forward load/store instructions but ignore ALU or control instructions when implementing uninitialized memory checking. The architecture provides three choices regarding whether an instruction should be forwarded or not. A FIFO can be configured to (i) ignore an instruction, (ii) forward an instruction only if a FIFO entry is available (ignore if the FIFO is full), or (iii) always forward an instruction. The third choice implies that an instruction commit is stalled if the FIFO is full. The FIFO may also be configured to either allow an instruction to commit as soon as it is enqueued or stall the commit until there is an acknowledgment from the co-processor.

In many co-processor extensions, the main core does not need to wait for the extension result because an exception from the co-processor does not need to be precise. For example, all four extensions in our prototype (UMC, DIFT, BC, and SEC) terminate a program if a check fails. There is no need to support a restart on these extensions. If an instruction requires a value from the monitor as in the "read from co-processor" instruction or if an exception from the monitor must be precise, the main core needs to delay a commit operation. For modern out-of-order processors, such delays simply mean that instructions stay in an ROB (Re-Order Buffer) longer without necessarily stalling the following instructions. In-order cores can either stall an instruction at the commit stage or add a simple roll-back mechanism to provide a precise exception and allow instructions to speculatively commit.

The extension uses additional FIFOs to communicate back to the main core. A back FIFO (BFIFO) sends a return value for the "read from co-processor" instruction. In addition to data, the control module (CTRL) allows a set of synchronization operations between main core and the co-processor. The co-processor sends an acknowledgment back (CACK) for an instruction when the commit stage in the main core

[1]There are 32 types in our prototype based on the SPARC architecture.

waits for a completion of the extension processing. On an exception or a trap on the main core, the core needs to wait for the co-processor to finish all pending instructions before starting the handler. For this purpose, the interface provides a signal (EMPTY) to indicate whether there are any pending instructions in the co-processor. The extension can also raise an exception using the trap signal (TRAP). If the interrupt level of the processor is sufficiently low, the main core acknowledges such an exception (PACK) and invokes a proper handler.

Note that the proposed FIFO interface with the extension can easily support custom instructions on the main core. For example, in order to implement an instruction to set a configuration register, an extension can be created to update the register on a particular instruction encoding.

D. Meta-Data Memory Hierarchy

For meta-data used by the extension for bookkeeping, the extension uses its own cache subsystem that is separate from the main core's L1 caches. This design minimizes changes to the main core's cache structures. Both the processing core and the extension share the lower-level memory hierarchy such as an L2 cache and main memory. Currently, the architecture does not maintain coherency between the main core's L1 caches and the meta-data L1 cache. For the extensions that we studied, the co-processor only needs to access meta-data in memory regions disjoint from program instructions and data. The architecture can be extended with a cache coherence mechanism if necessary.

The meta-data cache is almost identical to regular data caches except for the capability to write at a bit granularity. The meta-data cache holds 32-bit words as in regular caches. For reads and writes, the meta-data cache is given a 32-bit mask in addition to an address and a data word, and only updates or returns bits within the cache word where the bit mask is set. We found that the bit-level write capability is essential for efficient co-processing since many co-processing techniques work on meta-data much smaller than a word. Without this feature, a co-processor needs to perform an explicit cache read and then an explicit cache write in order to update meta-data. The data-width ports can be parametrized so the designer does not need to make any changes to the cache.

E. Extension

The designer only needs to create an extension that fits into this interface. The extension retrieves data from the core-to-extension FIFO using input pins and writes to the extension-to-core FIFO using output pins. Similarly, the module has input pins from the cache and output pins to the cache. Since the decoded signals as well as the execution trace are forwarded to the core-to-extension FIFO, the designer has many choices in how to process the given information. The designer can additionally use the CFGR to filter out instructions that are unnecessary for the execution of the extension. By reducing the number of forwarded instructions, the designer can perform complex calculations for a single instruction over multiple cycles. The FIFO will act as a buffer between the processor and monitoring extension. Alternatively, the designer can choose to pipeline the extension to increase the extension's throughput.

Type	Extension	Lines of Code			% Written
		Provided	Written	Total	
Baseline	-	4590	-	4590	-
Without Interface	UMC	4590	2105	6695	31.44%
	DIFT	4590	2530	7120	35.53%
	BC	4590	2626	7216	36.39%
	SEC	4590	2348	6938	33.84%
With Interface	UMC	7397	46	7443	0.62%
	DIFT	7397	97	7494	1.29%
	BC	7397	104	7501	1.39%
	SEC	7397	209	7606	2.75%

TABLE II
COMPLEXITY OVERHEAD OF WRITING AN SINGLE-STAGE RTL EXTENSION WITH AND WITHOUT THE INTERFACE.

The interface does not explicitly support multiple extensions. However, with an additional wrapper logic, designers can use the interface to support multiple extensions in parallel. The FIFO signals can be forwarded to each extension so that each extension can perform monitoring operations with relevant signals. The extension-to-processor signals such as an exception can be either combined with an OR operation or communicated back in a time-shared fashion. Similarly, the meta-data cache can be either duplicated for each extension or shared. If shared, the meta-data accesses from multiple accesses will be serialized, incurring additional performance overheads.

The design can be implemented in standard RTL, or high level synthesis can be used to generate the RTL. By using high level synthesis, the designer can explore the architectural design space with little modification of the code. In our designs, we use Cadence C-to-Silicon and find that it can generate RTL that is comparable to the handwritten RTL in terms of area, power, and performance. In addition, with C-to-Silicon we can explore the possibilities of pipelining with little or no change to the original code.

IV. EVALUATION

In this section, we evaluate the relative complexity of our architecture against the complexity of a custom extension using lines of code as a metric. To study overall overheads of the extensions, we also compare the silicon area, power, frequency, and performance against a baseline processor. Finally, we evaluate the relative complexity and area, power, and frequency of an extension generated using high level synthesis against one that is handwritten. We do not evaluate the functional effectiveness of each extension because we use extensions that have been previously proposed and studied.

A. Methodology

We can approximate the relative complexity of our design using lines of code as a metric. The more lines of code required to implement an extension, the more complex it may be. This metric also reflects the design time for the extension. An extension with more lines of code takes longer to design.

Extension	Description	Max Freq (MHz)	Area		Power		Performance Overhead
			μm^2	overhead	mW	overhead	
-	Unmodified Leon3 w/ 32KB L1	465	835,525	-	365	-	-
UMC	Leon3 w/ UMC	463	932,118	11.6%	388	6.3%	1.02
DIFT	Leon3 w/ DIFT	456	960,558	15%	388	6.3%	1.06
BC	Leon3 w/ BC	456	996,894	19.3%	393	7.7%	1.07
SEC	Leon3 w/ SEC	463	836,786	0.15%	364	-	1.00

TABLE III
THE AREA, POWER, FREQUENCY, AND PERFORMANCE OVERHEAD OF MONITORING EXTENSIONS. THE OVERHEADS IN SILICON AREA, POWER CONSUMPTION, AND PERFORMANCE ARE SHOWN RELATIVE TO THE BASELINE LEON3.

CLOC, a program that counts lines of code was used to count the lines in a standardized way. While the estimate is approximate, it serves as a good relative comparison.

For the evaluation, we built a prototype system based on the Leon3 microprocessor [22]. Leon3 is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. The Leon3 architecture provides a single-issue in-order pipeline with seven stages. The complexity was evaluated for a baseline Leon3 processor with 32-KB L1 caches. It was then evaluated for a Leon3 with a handwritten extension and a Leon3 with an extension written using our given framework.

In order to estimate the area, power, and frequency of the design, we used the Synopsys Design Compiler (DC) with a 65nm IBM technology library. The synthesis tool allowed us to evaluate the relative overhead of the extension in comparison to a baseline processor and allowed us to evaluate the effectiveness of a high level synthesis tool to generate an RTL with comparable overhead.

We implemented the extensions in RTL and in SystemC. For the SystemC implementations, the Cadence C-to-Silicon compiler was used to generate RTL that can be compared with the handwritten RTL. While we implemented extensions in both single-cycle and pipelined fashions for handwritten RTL, the SystemC implementations were not explicitly pipelined and the high-level synthesis tool generated a single-cycle RTL implementation.

B. Complexity

Table II shows the relative complexity, in terms of lines of code, of implementing the various extensions with the common architecture framework (interface) and without it. The baseline Leon3 core is written in approximately 4590 lines of code. Adding a single extension requires between 2105 and 2626 lines of code or between 31.44% and 36.39% of the total. In constrast, our interface and Leon3 core are written in approximately 7397 lines of code. Adding a single extension requires between 46 and 209 lines of code or between 0.62% and 2.75% of the total. The data clearly demonstrates that most of the components are the same between the extensions and can be reused without modification. Only a few hundred lines of code actually implement the functionality of the monitor. The remaining lines of code are used to create an interface between the extension and the processor. While it seems like the total number lines of code is actually more for the extensions with the proposed interface, the synthesis tool will optimize away most of the unused portions of the interface. For example, the interface includes a cache module, but SEC does not use the cache module, so the synthesis tool will remove the module from the netlist resulting in less actual overhead than the lines of code suggest.

C. Extension Overhead

Table III summarizes the estimated area, power consumption, operating frequency, and performance overhead for the Leon3 pipelined processor with and without various extensions. We found that the unmodified Leon3 with 32-KB L1 caches can run up to 465MHz and consume about 0.836mm^2 and 364.2mW. The full ASIC results, where the Leon3 processor with each extension is synthesized using the ASIC flow, show that UMC, DIFT, and BC consume 12 to 20% additional silicon area and 6 to 8% additional power. These overheads are dominated by the meta-data cache and FIFOs for the core interface. For SEC, the overheads are negligible because SEC does not require a meta-data cache or a complex interface. The Leon3 processor with an extension results in a slightly lower operating frequency because the extensions tap into internal pipeline signals. The performance overhead is calculated by taking the geometric mean of the performance in a set of benchmarks from MiBench such as stringsearch, fft, basicmath, and bitcount, and cryptographic kernels including SHA-1 and GMAC. The suite of benchmarks executes 7% worse on average for the BC extension. These overheads come from two sources, stalls from a full forward FIFO and contention for the shared memory.

While the relative area overheads are noticeable for the Leon3 processor, which is a tiny embedded processor, The results demonstrate that the proposed architecture framework has little affect on the frequency. The overheads in performance and power consumption are also minimal, less than 10% of the baseline processor. These results suggest that the proposed design framework, even while relying on common interfaces and infrastructures (meta-data caches), can still enable efficient run-time monitoring. The overheads are comparable to the ones that are reported in previous studies for custom implementations.

The area results again demonstrates that the standard synthesis tool is effective in removing unnecessary parts of the common interface. For example, SEC only has negligible area overheads because the meta-data cache and unused processor-to-extension signals are optimized out.

Extension	Max. Frequency (GHz)			Area (μm^2)			Power (mW)		
	RTL	SystemC	% Diff.	RTL	SystemC	% Diff.	RTL	SystemC	% Diff.
UMC	>2	>2	-	419	482	15.04%	0.436	0.428	-1.83%
DIFT	1.75	1.81	3.43%	3743	3328	-11.09%	2.33	2.30	-1.29%
BC	1.28	1.19	-7.03%	14417	13374	-7.23%	8.14	8.28	1.72%
SEC	1.37	1.39	1.46%	7051	6920	-1.86%	3.54	3.50	-1.13%

TABLE V
POWER, AREA, AND FREQUENCY COMPARISON BETWEEN SINGLE-STAGE RTL AND SYSTEMC CODE.

Extension	Lines		
	Single-Stage RTL	Pipelined RTL	SystemC
UMC	46	73	61
DIFT	97	180	135
BC	104	248	183
SEC	209	453	267

TABLE IV
COMPLEXITY COMPARISON BETWEEN RTL AND SYSTEMC CODE.

D. High Level Synthesis

Table IV summarizes the complexity difference in terms of lines of code between an extension that is handwritten in RTL and an extension that is written in SystemC and synthesized to RTL. For the four extensions that we tried, the SystemC code actually takes more lines to express the same functionality as in the single-stage RTL case. On the other hand, the SystemC code only takes between 59% to 84% as many lines to express the functionality compared to pipelined RTL code. One possible reason that the single-stage RTL code uses less lines is that the extensions that we implemented are relatively simple and do not require a high level language to implement. In more complex extensions, it is possible that the SystemC code could more efficiently describe the extension. In terms of absolute lines of code, any of the extensions can be implemented in less than three hundred lines of code. Because the designer does not need to implement the interface, the designer can implement a full extension with a relatively small amount of code. Although the SystemC code takes more lines to implement in the single-stage case, it allows the designer the benefit of more easily exploring the design space (such as the number of pipeline stages). Design space exploration becomes more critical for more complex extensions or if a reconfigurable fabric such as an FPGA is used instead of ASIC.

The synthesis results between the handwritten RTL and the SystemC generated RTL is shown in Table V. In the table, the area and power are compared at a common frequency of 1GHz, a frequency at which any extension can meet timing. In order to make a fair comparison, the extensions were coded in SystemC to match as much as possible with the original RTL code. In most cases, the generated SystemC code synthesizes into more efficient RTL than its handwritten counterpart. In BC, SEC, and DIFT, the SystemC code synthesizes to RTL which consumes less area. In the case of UMC, the SystemC code consumes more area, but the absolute area that is consumed is negligible (482μm^2). In most cases, the SystemC code consumes less power. In the BC case, however, the SystemC code does consume an additional 1.72% more power. Finally, the SystemC code can run at a faster maximum frequency, at 1.46% faster for the SEC case and 3.43% faster for the DIFT case. In the BC case, the SystemC code does perform worse by 7%. One possible reason that the BC extension performs worse in the SystemC code is due to the nature of the BC extension. The BC extension needs to read and write to the cache, so it is implemented like a state machine. In a normal state, it reads from the cache. For a store operation, it stalls for a cycle to perform the store to avoid the structural hazard on the cache. Implementing a timing sequence is more difficult in SystemC and may result in less efficient code. In simpler extensions, the SystemC code results in less overhead and better clock frequencies.

We attempted to use the CtoS compiler to generate pipelined RTL from the same SystemC code in order to compare it to handwritten pipelined RTL. However, the high-level synthesis tool could not automatically and successfully resolve and handle dependencies among pipeline stages through a register file. At the end, we were unsuccessful in automatically generating pipelined extensions for BC, DIFT, and UMC. For SEC, the CtoS compiler was able to generated pipelined RTL with a maximum frequency of 3.57GHz, compared to 3.23GHz for the handwritten RTL. The area and power of the generated RTL at a frequency of 1GHz were 17.28μm^2 and 23.68mW for the generated pipelined RTL compared to 7.97μm^2 and 8.46mW. The compiler was able to generate RTL using the same SystemC code with 10.5% performance improvement, but at the cost of 116.8% more area and 179.9% more power. The compiler allows the designer to set the target frequency, so the area and power numbers might be more comparable if the frequency target were not set so high.

V. RELATED WORK

This section summarizes the existing work on hardware-based run-time monitors. To the best of our knowledge, this work is the first that proposes the use of a framework with high level synthesis for fast development of run-time monitors. Previous work has focused on efficiently implementing run-time monitors and increasing the programmability of the run-time monitors.

Recently, researchers have proposed to utilize idle cores on a many-core processor for run-time monitoring of security and reliability properties. For example, INDRA [23] 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 multi-cores [24]. Unfortunately, because the checker core needs to run multiple instructions to process each event from the computation core, these early designs are either limited to

coarse-grained monitoring or incur significant slowdowns (3 to 10x).

Other researchers have focused on hardware run-time monitors that are programmable. MemTracker [9] uses a programmable finite state machine and tags in memory to support extensions that monitor memory accesses. FlexiTaint [10] supports DIFT operations with a fully programmable tag propagation and check policies. FlexCore [25] provides a framework very similar to our framework to program run-time monitors in a reconfigurable fabric. While these monitors allow for more flexibility after chip fabrication, they have more significant overheads associated with them. For example, SEC in FlexCore is reported to have a 46.7% area overhead, while the ASIC implementation in our approach has 0.15% area overhead.

VI. Conclusion

In this paper, we proposed an architectural framework that allows for fast development of run-time monitoring and bookkeeping extensions. By using a common interface, the designer can focus on developing the extensions with less lines of code and less modifications. Our case studies and prototypes of four extensions show that our architectural framework can support a wide array of monitors. We additionally explore the use of high level synthesis in the framework to enable the designer to develop the extension using C code. For the four extensions that we tried, we find that while the extension takes more lines of code to write in SystemC, the synthesized RTL performs slightly better than the handwritten RTL.

For future work, we can extend our study to multi-thread programs on multi-core processors and make modifications to solve coherence issues between the caches and the interface. We could also examine more complex extensions or multiple parallel extensions that require more logic and can be implemented using multiple cycles or through pipelining. Finally, we can consider pipelining in high level synthesis and supporting an arbitrary number of pipeline stages.

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] J. Newsome and D. Song, "Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software," in *Proceedings of the 2005 Network and Distributed Systems Symposium*, 2005.

[3] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A Flexible Information Flow Architecture for Software Security," in *Proceedings of the 34th International Symposium on Computer Architecture*, 2007.

[4] 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.

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

[6] 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.

[7] 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.

[8] F. Qin, C. Wang, Z. Li, H. seop 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.

[9] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, "MemTracker: Efficient and programmable support for memory access monitoring and debugging," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 273–284.

[10] G. Venkataramani, I. Doudalis, Y. 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.

[11] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors in C and C++ programs," in *Proceedings of the Winter 1992 USENIX Conference*, 1992, p. 125138.

[12] T. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Proceedings of the* 32th *International Symposium on Microarchitecture*, November 1999.

[13] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proceedings of the* 37th *International Conference on Microarchitecture*, December 2004.

[14] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *Proceedings of the* 11th *Annual Network and Distributed System Security Symposium*, 2004.

[15] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," in *Proceeding of the* 28th *International Conference on Software Engineering*, May 2006.

[16] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[17] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008, pp. 263–277.

[18] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas, "iWatcher: Efficient Architectural Support for Software Debugging," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[19] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003, pp. 122–135.

[20] J. Joao, O. Mutlu, and Y. Patt, "Flexible reference-counting-based hardware acceleration for garbage collection," in *Proceedings of the* 36th *International Symposium on Computer Architecture*, 2009.

[21] H. Kannan, "Ordering decoupled metadata accesses in multiprocessors," in *ACM/IEEE* 42nd *International Symposium on Microarchitecture (MICRO-42)*, December 2009.

[22] J. Gaisler, E. Catovic, M. Isomaki, K. Glembo, and S. Habinc, "GRLIB IP Core User's Manual," 2008.

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

[24] 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.

[25] D. Deng, D. Lo, G. Malysa, S. Schneider, and G. Suh, "Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric," in *Proceedings of the 43rd International Symposium on Microarchitecture*, 2010.