High-Performance Parallel Accelerator for Flexible and Efficient Run-Time Monitoring

Daniel Y. Deng and G. Edward Suh
Computer Systems Laboratory, Cornell University
Ithaca, New York 14850
{deng, suh}@csl.cornell.edu

***Abstract*—This paper proposes Harmoni, a high performance hardware accelerator architecture that can support a broad range of run-time monitoring and bookkeeping functions. Unlike custom hardware, which offers very little configurability after it has been fabricated, Harmoni is highly configurable and can allow a wide range of different hardware monitoring and bookkeeping functions to be dynamically added to a processing core even after the chip has already been fabricated. The Harmoni architecture achieves much higher efficiency than software implementations and previously proposed monitoring platforms by closely matching the common characteristics of run-time monitoring functions that are based on the notion of tagging. We implemented an RTL prototype of Harmoni and evaluated it with several example monitoring functions for security and programmability. The prototype demonstrates that the architecture can support a wide range of monitoring functions with different characteristics. Harmoni takes moderate silicon area, has very high throughput, and incurs low overheads on monitored programs.**

I. Introduction

As we expand the use of computing devices to handle more sensitive information and control critical infrastructure, secure and reliable operation becomes increasingly more important. In this context, run-time monitoring of program execution at an instruction granularity provides an effective approach to ensure a wide range of security and reliability properties, especially with dedicated hardware support. As an example, Dynamic Information Flow Tracking (DIFT) is a recently proposed security technique that tracks and restricts the use of untrusted I/O inputs by performing additional bookkeeping and checking operations on each instruction that could handle data derived from untrusted I/O inputs. DIFT has been shown to be quite effective in detecting a large class of common software attacks [1], [2], [3]. Similarly, run-time monitoring has been shown to enable many types of capabilities such as fine-grained memory protection [4], array bound checking [5], [6], software debugging support [7], managed language support such as garbage collection [8], hardware error detection [9], etc.

This paper presents a programmable accelerator that is designed to enable a large class of run-time monitoring techniques to execute efficiently in parallel to the main processing core. Unfortunately, existing proposals for run-time monitoring techniques suffer from either limited programmability or high performance/energy overheads.

For example, while a custom hardware implementation of a runtime monitoring scheme can have negligible run-time overheads, high development costs and inflexibility of hardware have made custom hardware difficult to deploy in practice. Modern microprocessor development may take several years and hundreds of engineers from an initial design to production, and custom hardware cannot be added or updated after the fabrication. Because of the high costs of development and silicon resources, processor vendors would rather not implement a mechanism in custom hardware unless the mechanism is already proven and is widely used.

At the other end of the spectrum, monitoring in software offers very high flexibility. Software writers can leverage the inherent programmability of a general purpose processor to implement mechanisms of arbitrary complexity using as many general purpose processor instructions as necessary. However, software implementations of instruction-grained runtime monitoring mechanisms typically suffer from high performance and energy overheads. For example, a software implementation for DIFT monitoring on a single core is reported to have an average slowdown of 3.6X even with aggressive optimizations [10]. The performance overheads of software monitoring can be mitigated using parallel processing [11], however, this entails using multiple processing cores to run each computation thread and is likely to increase power consumption by a factor of two or more.

In essence, a traditional processing core is a poor match in terms of efficiency for many runtime monitoring schemes; for instance, DIFT needs to propagate and check 1-bit tags on each instruction while a processing core is optimized for sequential 32-bit (or 64-bit) operations. The bit-level configurability of FPGAs makes them a promising solution as an efficient accelerator for runtime monitoring. A recent proposal discussed the flexibility and energy efficiency of the FPGA co-processing approach [12] but also reported that the traditional FPGA architecture can only keep up with low-performance processing cores running at hundreds of MHz ($<$ 500MHz). For higher performance cores, an FPGA co-processor can incur a significant performance penalty.

Fundamentally, there exists a trade-off between efficiency and programmability as illustrated in Figure 1. Because programmability requires additional hardware such as a multiplexer to allow a choice, greater programmability implies higher overheads. In this paper, we develop an accelerator architecture that can provide low-overhead run-time monitoring even for high-performance processors with a few GHz

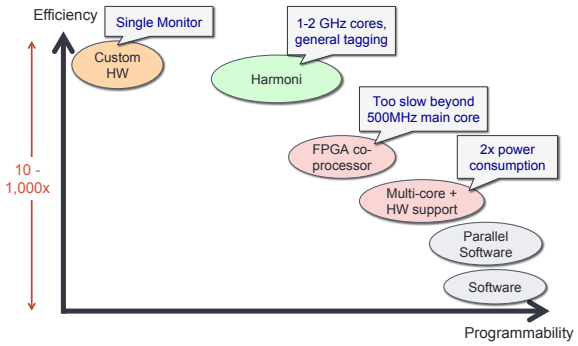


Fig. 1. Trade-off between efficiency and programmability.

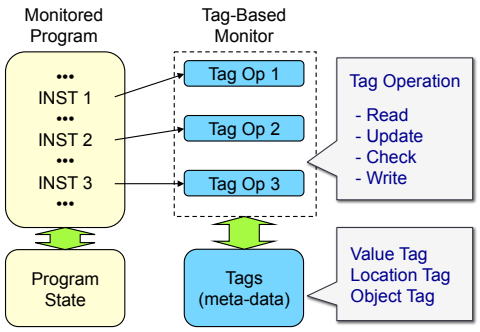


Fig. 2. Parallel run-time monitoring with tags.

clock frequency. We achieve this goal by carefully restricting the programmability and optimizing the architecture for common monitoring operations. In particular, we found that many instruction-grained monitoring techniques are built upon tagging, where a piece of meta-data is attached to each value, memory location, or program object, and the meta-data is updated/checked based on instruction events in the monitored program. While the notion of tagging has been studied before, this work presents a unified architectural framework that can support a broad range of run-time monitoring techniques that use tagging.

Our on-chip accelerator architecture, named Harmoni (Hardware Accelerator for Runtime MONItoring), provides an efficient realization of the general tagging model. The Harmoni architecture is designed to match common tagging operations, which consist of reading, updating, checking, and writing the tag for each instruction on the main processing core. Harmoni maintains programmability by broadly supporting monitoring schemes that use tagging in various types and granularities. Harmoni also supports operations on tags that range from regular ALU computations to irregular tag update and check by combining ALUs and memory-based look-up tables.

By focusing specifically on supporting monitoring schemes that make use of tagging, Harmoni can achieve a high operating clock frequency of 1.25GHz on a 65nm standard cell technology. This higher clock frequency allows Harmoni to keep pace with high-performance processors that are running at clock frequencies of a few GHz and have minimal performance overheads. An evaluation of the Harmoni prototype also shows that the architecture is far more energy efficient compared to using multiple identical processing cores for both main computations and monitoring operations in parallel. Harmoni has moderate area overheads for a small 5-stage embedded processor, but is quite small compared to processing cores that run at higher frequencies.

This paper makes the following main contributions:

- *General run-time monitoring model*: The paper proposes a general model for parallel run-time monitoring based on the notion of tagging. This model enables efficient hardware implementations while capturing a broad range of monitoring techniques.
- *Accelerator architecture*: The paper presents an accelerator architecture that realizes the proposed monitoring model. Unlike other programmable monitoring platforms available today, the accelerator can match the throughput of a high-frequency processing core with low energy overheads.
- *Prototype implementations and evaluations*: The paper implements and studies a prototype in RTL (VHDL) and presents results from its evaluation.

The rest of the paper is organized as follows. Section II describes the model for the parallel monitoring with tagging, and shows how example monitors can map to the model. Section III presents the tagging architecture. Section IV studies the performance, area, and power consumption of our architecture. Section V discusses the related work, and Section VI concludes the paper.

II. Tag-Based Monitoring Model

The main challenge in designing a high-performance, programmable accelerator for run-time monitoring lies in identifying functions that are common across a broad class of monitors so that certain aspects of programmability can be limited without sacrificing functionality. In this context, we studied a number of run-time monitoring techniques and found that many of them can be seen under the common framework of tagging; monitoring schemes often maintain and check meta-data information for data that are used in the main core. This section presents our run-time monitoring model used on Harmoni that is based on the notion of tagging.

A. Overview

Figure 2 shows a high-level model of typical instruction-grained monitoring techniques. In the figure, the light blocks on the left represent the main computation and the darker blocks on the right represent the monitoring function. A monitor often maintains its own meta-data to keep track of the history of the monitored computation; we will refer to this meta-data as "tags" in this discussion. Conceptually, the monitor observes a trace of instructions from the monitored program and checks program properties by maintaining and checking tags. We will refer to maintaining and checking tags as "tagging operations" in this discussion. A failed tag check indicates that a monitored-for event, which was intended to be caught or avoided, had occurred in the program running on

the main processing core. Therefore, if a tag check fails, the monitor raises an exception.

In general, a monitoring function can be characterized based on its tag type and tag operations. The tag type defines the meta-data that are maintained by the monitor. The set of tag operations define which events in the monitored program triggers a tag operation and how the tags are updated and/or check on such events.

B. Tag (Meta-data) Types

Run-time monitoring techniques typically associate a tag (meta-data) with each piece of state in the monitored program. In particular, monitoring techniques typically rely on tags for three types of program state: data value, memory location, and high-level program objects.

Value tag: Many monitoring techniques keep a meta-data tag for each data or pointer value in a program. For example, DIFT maintains a 1-bit tag to indicate whether a word/byte is from a potentially malicious I/O channel or not, and array bound checks may keep base and bound information for each pointer. Because most programming languages use 8-bit or 32-bit (or 64-bit) variables to express a value, the *value tag* is often maintained for each word or byte in registers and memory. Each tag also follows the corresponding value as it propagates during an execution. As an example, loading a value from memory into a register moves the corresponding value tag from memory to a tag register, and an output of an ALU often inherits its tag from tags of source operands.

Location tag: A tag may be associated with a location such as a memory address instead of a value. Such a *location tag* is often used to keep information on the properties of storage itself rather than its content. For example, a memory protection technique can keep permission bits for each memory location and check if an access is allowed. A software debugging support may use a location tag to check if each memory location is initialized before a read. Similar to the value tags, the location tags are generally kept at a word or byte granularity, matching typical sizes of variables (int, char, etc.) in program languages. Yet, the location tag does not follow memory content.

Object tag: A monitoring scheme may keep coarse-grained tags for relatively large program objects such as classes, structures, arrays, etc. instead of keeping fine-grained tags per byte or word. For example, a reference counter for a garbage collection is maintained for each program object. While it is possible to implement such coarse-grained tags using per-byte or per-word tags - essentially, make all tags that correspond to a large object to be the same value - it is far more efficient to manage and update the *object tags* separately.

C. Tag Operations

In addition to the type of tags, a tag-based run-time monitoring scheme can be characterized in terms of which events in the monitored program triggers tag operations and what actions are taken within the tag operation. In general, actions to update or check tags are triggered when the corresponding values or locations are used by the monitored program. Information about the values or location used in a program can be deduced from each instruction that executes. For example, load/store instructions indicate accesses to memory locations or values. ALU instructions show processing of values. As a result, low-level tag operations can often be determined transparently based on the instruction opcode.

On the other hand, certain tag operations may be triggered by high-level program events that need to be explicitly communicated from the monitored program to the monitor. For example, a monitor to detect out-of-bound memory accesses needs to set a tag, which encodes bounds for each pointer, on memory allocation and deallocation events in order to check bounds on each memory access. A compiler can often automatically annotate a program to add explicit tag operations for common program events and information such as function calls, memory management operations, type information, etc. High-level program events may need to be annotated by a programmer.

For each monitored program event, the tagging operation typically consists of the following common sequence of operations.

- **Read**: The monitor reads tags that correspond to the values or locations used by the monitored program: registers or memory for value tags, memory for location tags, and a special table for object tags.
- **Update**: The monitor updates tags based on the monitored program event.
- **Check**: The monitor may checks tags for an invariant and signals an exception if the invariant is violated.
- **Write-back**: The monitor writes back the updated tag. The value tag is typically written to the tag that corresponds to the result of the monitored program's instruction. The location tag is often written to the location that is accessed by the monitored program.

D. Monitoring Examples

Here, we survey several previously proposed monitoring schemes for security, debugging, and reliability, and discuss how they map to the proposed tagging model. This is not a comprehensive list of all possible run-time monitoring functions. However, these schemes represent a spectrum of monitoring functions that are diverse in terms of the operations that they perform, information from the main processing core that they act on, and the hardware requirements on the meta-data operations.

Dynamic information flow tracking (DIFT) [1]: DIFT is a security protection technique that prevents common software exploits from taking over a vulnerable program by tracking and limiting uses of untrusted I/O inputs. For example, typical attacks that changes a program control flow can be detected by preventing I/O inputs from being used as a code pointer. DIFT uses a 1-bit value tag per memory word and register to indicates whether the value has been tainted by data from untrusted I/O inputs. DIFT uses operating system support to set the tag (taint) input-derived data, and then transparently

Monitor	Trigger	Action
DIFT (1-bit value tag)	ALU instructions	Tag(reg_dest) := Tag(reg_src1) or Tag(reg_src2)
	LOAD instructions	Tag(reg_dest) := Tag(mem_addr)
	STORE instructions	Tag(mem_addr) := Tag(reg_dest)
	JUMP instructions	check reg_src1 != "1"
UMC (1-bit location tag)	LOAD instructions	check Tag(mem_addr) != "0"
	STORE instructions	Tag(mem_addr) := "1"
BC (4-bit location tag and 4-bit value tag)	LOAD instructions	check Tag(mem_addr) == Tag(reg_src1)
		Tag(reg_src1) := Tag(mem_addr)
	STORE instructions	check Tag(mem_addr) == Tag(reg_src1)
		Tag(mem_addr) := Tag(mem_src1)
	ADD instructions	Tag(reg_dest) := Tag(reg_src1) + Tag(reg_src2)
	SUB instructions	Tag(reg_dest) := Tag(reg_src1) - Tag(reg_src2)
	OR instructions	Tag(reg_dest) := 0
	XOR instructions	Tag(reg_dest) := 0
	NOT instructions	Tag(reg_dest) := -Tag(reg_src1)
RC (32-bit object tag)	Create pointer	refcnt[addr] := refcnt[addr]+1
	Destroy pointer	refcnt[addr] := refcnt[addr]-1

TABLE I
TAG TYPES AND OPERATIONS FOR A SET OF RUN-TIME MONITORING FUNCTIONS.

tracks the flow of tainted information on each instruction in the monitored application. On each ALU instruction, the tag of the destination register is set if at least one of the two input operand tags is set. On each memory access instruction, the tag is copied from the source to the destination: a store copies a tag from the source register to the destination memory location, a load copies a tag from memory to a register. On a control transfer instruction, such as an indirect jump, the tag of the target address in the source operand is checked to ensure that the address is not tainted.

Uninitialized memory checking (UMC) [13]: UMC detects programming mistakes involving uninitialized variables. Eliminating these memory errors can be a very important part of the software development cycle. UMC uses 1-bit location tag per word in memory to indicate whether the memory location has been initialized since being allocated. UMC leverages software support to clear tags when memory is allocated. On each store instruction, the tag of the accessed memory word is set. On each load instruction, the tag for the accessed memory word is read and checked to detect when data is read before being initialized.

Memory bounds checking (BC) [6]: While there exist a number of run-time bounds checking techniques, in this paper, we discuss a technique that utilizes a notion of coloring both pointers and corresponding memory locations. Conceptually, this approach maintains a location tag for each word in memory and a value tag for each register and each word in memory. The location tag encodes the color for the memory location, and the value tag encodes the color for a pointer. Our implementation uses 4-bit tags. The tags are set so that pointer and memory tags match for in-bound accesses and differ for out-of-bound accesses. On memory allocation events, BC assigns the same tag value (color) to both memory locations that are allocated and the pointer than is returned. On each memory access instruction, the tag of the pointer that is used to access memory is compared to the tag of the accessed memory location. The access is allowed only when the tags match.

In addition to checking the color tags, the BC scheme also tracks the tags for pointers. On memory load instructions, the value tag is loaded from memory into the destination tag register. On memory store instructions, the value tag is copied into memory as the pointer color tag of the accessed memory location. On ALU (ADD/SUB) instructions, the value tags are propagated from the source operands to the output register to keep track of tags for an updated pointer on an pointer arithmetic operation.

Reference counting (RC) [8]: RC transparently performs reference counting in hardware to aid garbage collection mechanisms implemented in software. In this scheme, because hardware can transparently maintain reference count information, software memory allocation mechanisms can quickly find and free memory blocks that are no longer in use by the monitored application. RC uses multi-bit object tags for each object in the monitored application. In our study, we used 32-bit tags to represent integer reference counts. RC leverages compiler modifications to find instructions that create or destroy pointer references. On an instruction that creates a new pointer, the pointer value is used to look up the object tag, and the tag is incremented. On an instruction that destroys an existing pointer, the pointer value is used to look up the object tag, and the tag is decremented.

Table I summarizes the characteristics of each monitoring technique in terms of the tag type and tag operations.

III. ARCHITECTURE DESIGN

In this section, we present the design of Harmoni, which can efficiently support a broad spectrum of runtime monitoring techniques based on tagging.

A. Overview

Harmoni is designed as a parallel decoupled co-processor in a runtime monitoring architecture as shown in Figure 3. The Harmoni architecture supports fine-grained monitoring techniques by adding specialized hardware support to the processing core to forward an execution trace of selected types of instructions from the main processing core to Harmoni. The forwarded instructions are selected based on the opcode. The execution trace includes the opcode, register indexes of

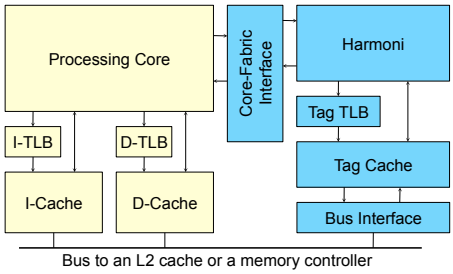


Fig. 3. High-level block diagram of the Harmoni architecture.

source and destination registers, the accessed memory address on a load/store instruction, and a pointer value for a high-level object that are being used in the main core.

Forwarded instructions trigger tag operations on the Harmoni co-processor, which update the corresponding tags and/or check tag values for errors or events in the monitored program. In order to efficiently manage location and value tags in main memory, the Harmoni architecture includes a tag TLB to translate a data address to a tag address and an L1 tag cache. The Harmoni pipeline raises an exception if a check fails; this exception is delivered over a backward FIFO to the main processing core, which invokes an appropriate exception handler to pinpoint the cause of the exception and take necessary actions.

The architecture allows the monitoring on Harmoni to be mostly decoupled from the program running on the main processing core by using a FIFO within the Core-Fabric interface to buffer forwarded instructions. Using the interface, the main processing core can queue each completed instruction that is to be forwarded to Harmoni into the FIFO and then continue execution without waiting for the corresponding check on Harmoni to complete. The Harmoni co-processor can dequeue instructions at its own pace from the Core-Fabric interface and perform tag operations for each dequeued instruction. As long as the FIFO within the Core-Fabric interface is not full, the main processor and Harmoni can effectively run in parallel. In our implementation of the Harmoni prototype, the FIFO is sufficiently sized (64 entries) to accommodate short-term differences in the throughput between the main processing core and the Harmoni co-processor.

Because the monitoring on Harmoni is performed in a decoupled fashion, an exception signal reaches the main core after the corresponding instruction has already completed. The delayed exception is sufficient if we assume a fail-stop model where the monitored program is terminated on an error. To avoid erroneous program outputs and corruption in persistent state, critical instructions such as system calls with externally visible effects are delayed until Harmoni checks are completed for all preceding instructions. If necessary, exceptions raised by Harmoni can be made precise by either utilizing standard precise exception mechanisms for modern out-of-order processors or by adding a small amount of buffering to in-order processors [14].

B. Programmability

At a high level, the Harmoni co-processor supports a wide-range of tag-based program monitoring techniques by allowing both tag type and tag operations to be customized depending on the monitoring technique. The Harmoni co-processor is designed to efficiently support the three types of tags: value, location, and object tags. The size of the tag can be statically configured to be any value that is a power of two, up to a word (32 bits). The granularity of location tags can be set to be a value that is a power of two and equal to or greater than a byte. The granularity of a single byte means that there is a tag for each byte. The object tags allow an arbitrary range in memory to be tagged.

For location tags, the Harmoni architecture supports storing a tag for each memory location using a tag memory hierarchy (TMEM). Figure 4 show the block diagram with major components in the Harmoni pipeline. The tags are stored in a linear array in main memory alongside program instructions and data. TMEM is accessed using a memory address forwarded along with an instruction from the main processing core. The mapping between the monitored program's memory address and the corresponding tag address can be done as a simple static translation in virtual addresses based on the tag size and granularity. The operating system can allocate physical memory space to program data and tags using the virtual memory mechanism. The mappings can be cached in the tag TLB, which translates memory addresses used by the monitored application into memory tag addresses. Harmoni supports the tagging of memory blocks with statically configurable sizes of one byte or larger (any power of two) with tags that can be any size that is a power of two in bits and up to a word length (32 bits). Similar to regular data accesses, the latency of tag accesses is reduced using a tag cache hierarchy. In our prototype, the cache uses write-back and write-allocate policies.

For value tags, Harmoni supports tagging each register and each value in memory using a tag register file (TRF) and tag memory hierarchy (TMEM). The TRF stores a tag for the corresponding register in the main core. The TRF is accessed using source/destination register numbers from the main core. The TMEM is accessed using the memory address from the main core on load/store instructions in the same way that the location tag is handled.

Because both location and value tags require tags in memory, the memory hierarchy in Harmoni needs to be able to deal with two tags at a time in order to allow both tag types to be used simultaneously. In case that both location and value tags are enabled, Harmoni stores a concatenation of the two tags in a linear array so that both tags can be easily accessed together. The tag cache allows reading both tags together and updating only one tag type. For example, our bounds checking prototype uses 4-bit location and value tags per word. Harmoni maintains an 8-bit meta-data per word in memory. On a load, the tag cache reads the 8-bit meta-data and splits it into two 4-bit tags. On a store, the cache only overwrites 4 bits out of

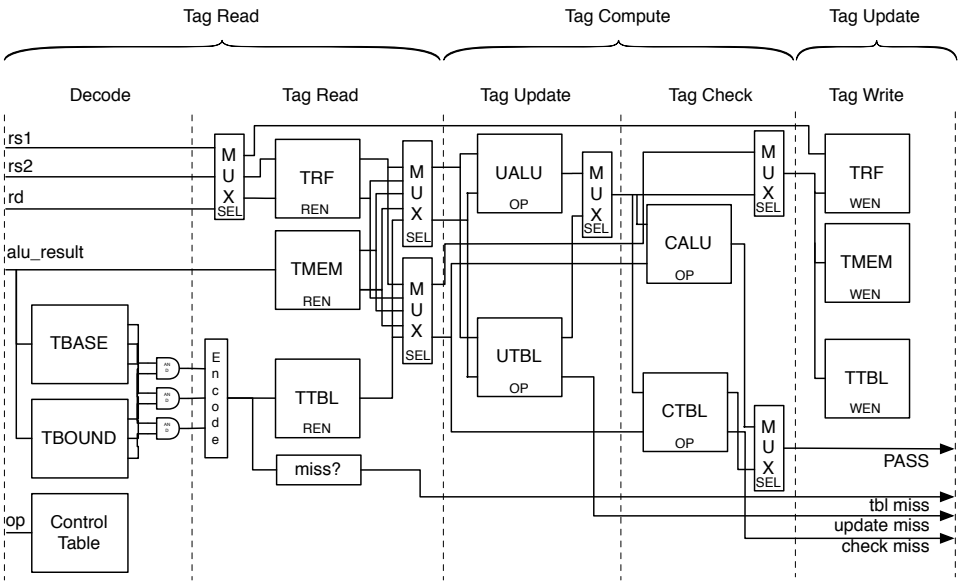


Fig. 4. High level block diagram of the Harmoni pipeline. The pipeline can be broken down into five discrete stages. The first two stages read the tags of operands used in the instruction, the third and fourth stage update and check the tags, the fifth stage writes the updated tag. The output of the control table is connected to all of the modules in the last four stages of the pipeline and determines their behavior.

the 8-bit meta-data.

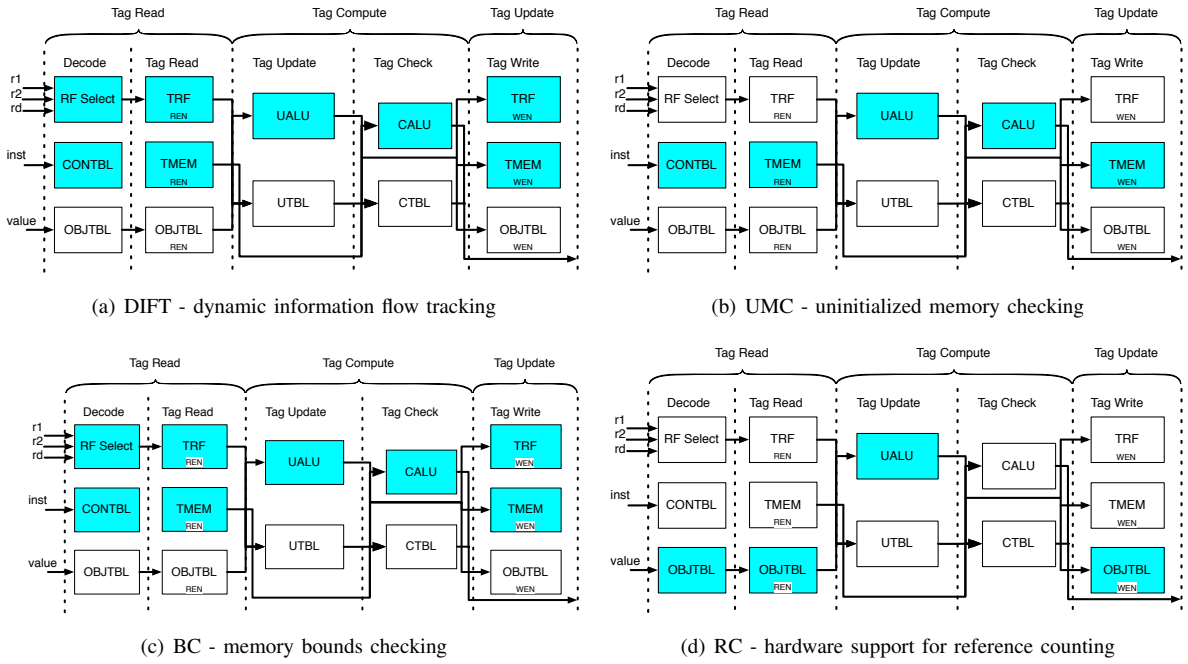
Object tags are supported with a software-controlled object table (OBJTBL), which stores a tag for recently accessed high-level program objects. Each entry in the OBJTBL contains base and bound addresses of an object along with its tag. The table is looked up in two steps using a pointer from the main core. In the first step, the table compares the base and bound addresses of each entry with the input pointer to see if there is a match. If the corresponding entry is found, the tag value is read in the second step. If the entry does not exists in the table, a object table miss exception is raised so that the table can be updated by software. In our prototype, the OBJTBL can cache up to 32 entries for object tags. Previous studies [15], [16], [17] have shown that program objects and arrays have very high temporal locality, and only a handful of entries are sufficient to cache object tags with low miss rates.

In addition to flexible tag types and sizes, the Harmoni architecture also supports programmable tag update and check operations. On each forwarded instruction from the main core, Harmoni can compute a new tag for the destination that is accessed by the instruction. More specifically, this update operation can be performed either by a tag ALU (UALU) or a software-controlled table (UTBL). The UALU can handle full 32-bit integer computations on two tags, which can be from tag registers, tag memory, or the tag object table, and is designed for monitoring techniques with regular tag update policies. For example, in reference counting, each pointer creation event results in a regular increment of the object's reference count.

The update table (UTBL) works as a cache that stores recent tag update rules and enables complex software-controlled tag update policies. The UTBL takes two input tags along with control bits that define an operation. Each entry stores a new tag value for the specified tag operation with specific input tag values. The UTBL raises an exception if an entry cannot be found for a monitored instruction that is configured to use the table. Then, software computes the new tag value and caches it in the UTBL. The updated tag can be simply read from the table if an identical tag operation with input tag values is later performed.

Similar to the update, the tag check operation can also be performed using either a check ALU (CALU) or a software-controlled check table (CTBL). The check operation can take up to two input tags and outputs a 1-bit signal indicating whether a check passes or not. One input tag comes from the output of the tag update unit, and the other input tag is from the tag register, the tag memory, or the object table. The CALU can handle a range of full 32-bit binary or unary comparison operations on one or two tags. The CTBL handles complex check policies by storing recent check results from software in the same way that the UTBL caches recent update rules. In our prototype, both UTBL and CTBL are implemented as a direct-mapped caches with 32 entries.

To configure the tag operations, Harmoni uses a statically-programmed look-up table for pipeline control signals (CONTBL). The CONTBL is indexed by the opcode of the forwarded instruction and holds one set of control signals for each opcode type. Our prototype supports 32 instruction types. The control signals from the CONTBL determine where tags are read from, how the tag update and check should be performed, and where the updated tag should be written to. As an example, for the tag update operation, the CONTBL signals specify whether the computation will be handled by the UALU or the UTBL, which tag values are used as inputs (up to two from the tag registers, up to two from the tag memory, up to one from the object table), and what the UALU or UTBL operation should be.



(a) DIFT - dynamic information flow tracking

(b) UMC - uninitialized memory checking

(c) BC - memory bounds checking

(d) RC - hardware support for reference counting

Fig. 5. Run-time monitoring techniques mapped to the Harmoni co-processor.

C. Tag Processing Pipeline

Having described the Harmoni architecture at a high-level, we now describe the Harmoni pipeline in more detail, which is shown in Figure 4. The Harmoni pipeline can be broken down into five stages. The first two stages read the relevant tags for the monitored instruction, the third stage updates the tag, the fourth stage performs a tag check, and the fifth stage writes the updated tag back to the tag register file, the tag memory, or the object tag table.

In the first stage of the pipeline, the instruction is "decoded". The CONTBL is accessed using the opcode of the forwarded instruction. The tag register file indexes to read tags from are specified in this control table. At the same time, the stage looks up the OBJTBL by checking the base and bound addresses with the pointer address from the main processing core.

In the second stage of the pipeline, tag information is accessed from the tag register file (TRF), the tag memory (TMEM), and the software-controlled table (OBJTBL). Up to two tags are read from the TRF, the TMEM is accessed for up to two tags corresponding to the memory content or address (value or location tag), and one object tag is read from the OBJTBL.

In the third stage of the pipeline, the updated tag is computed. Up to two tags are used by either the UALU or the UTBL to calculate the updated tag. The UALU allows broad range of typical tag processing operations including bit-wise logic operations (AND, NOT, OR, XOR), integer arithmetic operations (Add and Sub), bit-shifting operations (shift and rotate), and propagation of either operand. The UTBL caches software specified tag update results in order to perform a more complex tag update. The output of either the UALU or the UTBL is selected using control signals from the CONTBL at the end of the third stage and propagated to the next stage of the Harmoni pipeline.

In the fourth stage of the pipeline, the tag is checked against invariants. The CALU takes the updated tag along with another tag from the TRF, the TMEM, or the OBJTBL, and performs a unary or binary comparison to determine if an exception should be raised. The CTBL uses the same input tags to perform a complex tag check. The CONTBL selects which module, the CALU or the CTBL, should drive an exception signal back to the main processing core.

In the fifth and final stage of the pipeline, the updated tag is written back to the tag register, the tag memory, and/or the object tag table. The updated tag is sent on a broadcast bus to these three structures, and the writing of this tag for each module is controlled by a set of control signals generated from the CONTBL.

D. Monitor Examples

Figure 5 shows how the run-time monitoring techniques that we discussed in Section II can be mapped to Harmoni. The figure highlights the modules that are used by each monitoring scheme in block diagrams.

The modules used by dynamic information flow tracking (DIFT) are shown in Figure 5(a). In DIFT, ALU instructions propagate taint information between registers, memory instructions propagate taint information between registers and memory, and taint is checked on control transfer instructions. For ALU instructions, the CONTBL enables reading from the TRF and register tags are sent to the UALU. The UALU is programmed to propagate the tag based on the input tags and the instruction opcode by performing an OR operation, and the result is written back to the TRF. For load instructions, the CONTBL enables reading a tag from the TMEM and sends the tag to to the UALU. The UALU passes through the taint tag unaltered and this result is written to the TRF using the

destination register index for the load. For store instructions, the CONTBL enables reading of the tag from the TRF. This tag is propagated through the UALU and into the TMEM. For indirect jump instructions, the tag of the jump target address is read from the TRF, propagated through the UALU, and checked in the CALU. If the tag is non-zero, an exception is raised.

The modules that are used by uninitialized memory checking (UMC) are shown in Figure 5(b). In UMC, the location tag of the memory that is accessed is read and checked on a load, and the location tag of the accessed memory address is set on a store. For load instructions, the CONTBL enables reading of the tag of the accessed memory location from the TMEM. This tag is propagated through the UALU unchanged, and checked in the CALU to confirm that the accessed memory location was initialized (the tag is set). For store instructions, the control table sets the UALU to output a constant "1", which is stored to the TMEM at the address from the store.

The modules used by bounds checking (BC) are shown in Figure 5(c). In bounds checking, explicit instructions set and clear a value tag (pointer tag) and location tags (corresponding locations) on memory allocation and deallocation events, the pointer tags are propagated on an ALU instruction and a load/store operation, and then the pointer and location tags are compared on each memory access instruction to ensure in-bound accesses. In our prototype, we implemented the scheme using 4-bit tags, which represent 16 colors. For ALU instructions, the value tags (pointer colors) of source operands are read from the TRF and propagated to the UALU. The UALU calculates the tag for the result, and this tag is written to the TRF for destination register. For memory load instructions, the CONTBL enables both the TRF and the TMEM in the second state to read both the value tag of the load address (TRF) and the value and location tags of the accessed memory location (TMEM). Then, the pointer tag of the memory address is compared with the memory location tag from the TMEM in the CALU to ensure that they match. The tag of the loaded memory value is then written back to the TRF. For memory store instructions, the pointer tag of the accessed address is read from the TRF and compared with the memory location tag from the TMEM as in the load case. The tag of the value that is being stored is then stored to the TMEM. To improve the accuracy of the bounds checking scheme, the pointer tag propagation can be complemented by the UTBL so that software can make more intelligent decisions on exceptional cases.

The modules used by hardware reference counting (RC) are shown in Figure 5(d). In the reference counting, specialized instructions that create or overwrite a pointer explicitly send the pointer that was created or overwritten to the co-processor. The pointer is compared to a stored list of object base and bound addresses in the OBJTBL to determine the reference count (tag) that needs to be updated. If the pointer does not lie within the base and bound addresses of any objects in the OBJTBL, an exception is raised so that software on the main processing core can update the OBJTBL. For instructions that create a pointer, the object that the created pointer points to is looked up and the reference count for that object is incremented in the UALU. This updated reference count is written back to the OBJTBL. For instructions that overwrite a pointer, the object that the overwritten pointer points to is looked up, the reference count for that object is decremented in the UALU, and this updated reference count is written back to the OBJTBL. The reference counts can be read by the main core to quickly determine if a certain object can be removed.

Leon3 Processor	
Pipeline	7-stage, in-order
Instruction cache	32 KB, 4-way set-associative
Data cache	32 KB, 4-way set-associative
Cache block Size	32 B
Cache write policy	write-through
Register file	144 registers, 8 windows
Harmoni Pipeline	
Control table	32 entries (28 bits per entry)
UTBL	32 entries
CTBL	32 entries
Harmoni Support Structures	
Core-Harmoni FIFO	64 entries
Tag cache	4KB, direct-mapped
Tag cache block size	32B
Tag cache write policy	write-back

TABLE II
ARCHITECTURE PARAMETERS.

E. Limitations

The Harmoni architecture targets to support a broad range of tag-based monitoring techniques efficiently through carefully trading off programmability and efficiency. While we found that many monitoring techniques can be mapped to the current Harmoni design, the architecture is not Turing complete and certain monitoring techniques may not work well. Here, we briefly discuss the limitations of the current architecture, which we plan to investigate in the future.

One main limitation of the current Harmoni design is that it only allows a single tag operation for each monitored instruction. Therefore, tagging techniques that require a sequence of operations for a single monitored instruction cannot be efficiently supported. We plan to investigate the possibility of expanding the control table to allow multiple operations per opcode or even simple control instructions. Another limitation comes from the limited interface to the main processing core. Currently, the architecture is designed to only work on tags but not data. Therefore, a monitor that checks data values of the monitored program such as soft error detection is not supported by the architecture.

IV. EVALUATION

To evaluate the Harmoni architecture, we implemented a prototype system based on the Leon3 microprocessor [18]. Leon3 is a synthesizable RTL model of a 32-bit processor compliant with the SPARC [19] instruction set. The Leon3 processor includes a single-issue in-order seven stage integer pipeline and 32KB of on-chip L1 instruction and data caches. Completed instructions are forwarded from the exception stage of the integer pipeline to the Harmoni co-processor. Since

Description	Max Freq (MHz)	Area		Power	
		μm^2	overhead	mW	overhead
Leon3 Processor - 32KB IL1/DL1	465	835,525	-	365	-
Harmoni (32-bit)	465	156,517	18.7%	46	12.46%
	1250	187,255	22.4%	120	32.9%
Harmoni (16-bit)	465	82,552	9.9%	24	6.6%
	1250	94,289	11.3%	63	17.3%
Harmoni (8-bit)	465	46,319	5.5%	14	3.8%
	1250	50,974	6.1%	35	9.6%
Support structures: FIFO, cache, etc.	458	271,442	32.5%	53	14.6%

TABLE III
THE AREA, POWER, AND FREQUENCY OF THE HARMONI ARCHITECTURE WITH DIFFERENT MAXIMUM TAG SIZES. THE OVERHEADS IN SILICON AREA AND POWER CONSUMPTION ARE SHOWN RELATIVE TO THE BASELINE LEON3 PROCESSOR.

the opcode in the SPARC ISA can come from different parts of the instruction and are irregular in size, we divide instructions in the SPARC ISA into 32 custom categories. Only instructions from categories that are relevant to the monitoring function being performed on Harmoni are forwarded from the Leon3 processor to Harmoni and the CONTBL is indexed by the instruction category. The Harmoni architecture was evaluated with the Harmoni pipeline and support structures that include the Core-Harmoni FIFO and a 4-KB on-chip tag cache. Table II summarizes the parameters that we used in the evaluation. To evaluate the area, power, and maximum frequency of this architecture, we synthesized Leon3, Harmoni, and corresponding hardware support structures in Synopsys Design Compiler using Virage 65nm standard cell libraries. The power estimates currently use a fixed toggle rate of 0.1 and static probability of 0.5 to provide rough comparisons. Table III shows the results of this analysis.

Even without extensive optimizations, the Harmoni pipeline can run up to 1.25 GHz, which is more than 2.5 times the maximum frequency of the Leon3 processing core. This result shows that Harmoni can keep pace with processing cores that have much higher operating frequencies and application performance. The rest of the Harmoni architecture, including the FIFO interface from the main core and a tag memory system, is synthesized with the Leon3 core and shown to have a minimal impact on the core's clock frequency even with the additional signals that are required for forwarding instructions and supporting an exception.

The Harmoni architecture does show noticeable area and power consumption compared to the Leon3 processor. The total area that includes the forwarding FIFO, the tag cache, and the full 32-bit Harmoni pipeline makes up an additional 55% in area compared to the baseline Leon3 processor. The area overhead can be mitigated by limiting the maximum size of the tags that Harmoni can support. By going to 16-bit and 8-bit pipelines for tag updates and tag checks, the respective area overheads of Harmoni can be reduced to 44% and 39%. Furthermore, we note Leon3 is a very small and simple embedded processing core. The performance of the Harmoni architecture allows it to be easily coupled with much larger and higher-performance processing cores that runs at a few GHz. For example, the Intel Atom processing core [20] is more than 25 times larger than Leon3 while running at a comparable clock frequency with Harmoni. The full 32-bit Harmoni pipeline would present an area overhead of less than 3% for Atom. Moreover, we note that the Harmoni architecture is far more energy efficient compared to an approach that utilizes a regular processing core as a monitor. At 465MHz, Harmoni is estimated to consume 46mW, which is less than 15% of the baseline processor power consumption. This is far more efficient than consuming twice the power using two identical cores for both computation and monitoring.

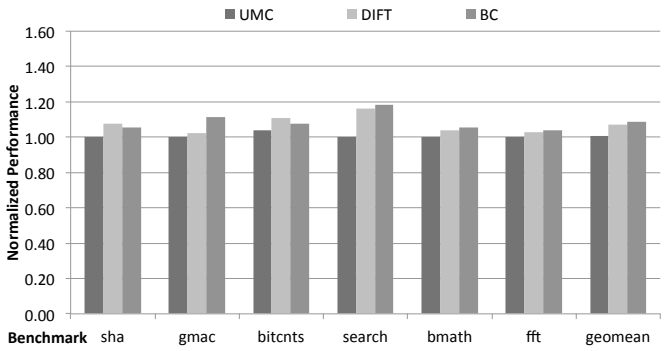
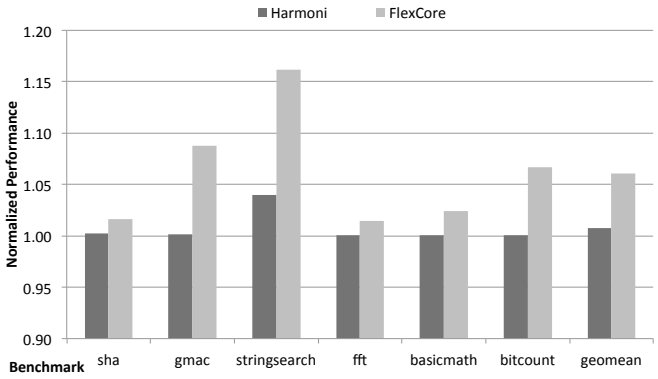


Fig. 6. The performance overheads of run-time monitoring on the Harmoni co-processor. The Y-axis shows normalized performance relative to an unmodified Leon3 processor. The X-axis shows the names of benchmarks used in the evaluation.

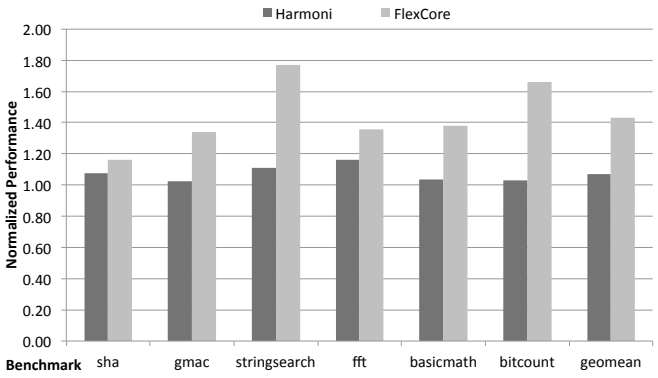
To evaluate the performance overheads of the Harmoni architecture, we performed RTL simulations of the architecture with three different monitoring techniques for several benchmarks. Benchmarks include programs from the MiBench [21] benchmark suite as well as two kernel benchmarks for SHA-256 and GMAC, which are popular cryptographic standards. We compared the execution time of these benchmarks between an unmodified Leon3 processor, Leon3 with a hardware monitor mapped to Harmoni, and Leon3 with a hardware monitor mapped to the FPGA fabric as in FlexCore [12].

Figure 6 shows the normalized execution time of benchmarks on Harmoni with respect to an unmodified Leon3 processing core. We implemented three monitoring techniques on Harmoni, including uninitialized memory checking (UMC), dynamic information flow tracking (DIFT), and array bounds checking (BC). The results show that run-time monitoring on Harmoni has low performance overheads on the monitored program. In fact, the Harmoni performance is almost identical to that of custom hardware monitors because most overheads come from tag accesses to memory, which is identical in both cases.

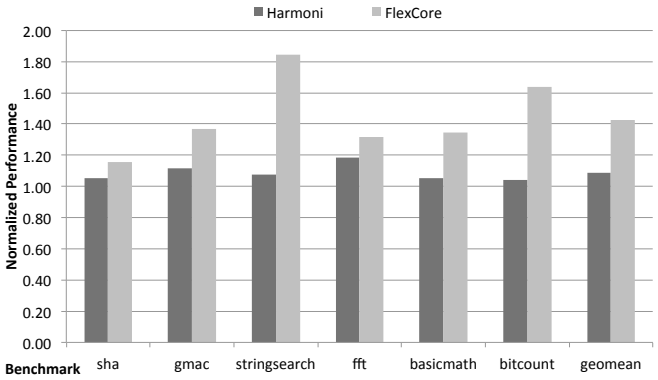
The Harmoni architecture as shown in Table III is capable



(a) UMC



(b) DIFT



(c) BC

Fig. 7. Normalized performance overheads of run-time monitoring on the Harmoni co-processor and the FPGA-based co-processor (FlexCore) for a main processing core running at 1GHz.

of running at a high clock frequency. The high performance and energy efficiency distinguish Harmoni from previously proposed run-time monitoring platforms. For example, FlexCore [12] provides flexibility using on-chip FPGA fabric, but reports that monitors on the FPGA fabric can only run at a couple of hundred MHz. Figure 7 shows the normalized performance of Harmoni on a main processing core with a high clock frequency (1GHz) and compares the result with the FlexCore approach with an on-chip FPGA fabric, which can only run at roughly one-fourth of the main core's clock frequency. Because Harmoni can match the main processing core's clock frequency, its performance impact is fairly low. On the other hand, due to its low clock frequency, monitoring on an FPGA introduces significant overheads for techniques that require frequent tag operations. For the majority of benchmarks that we ran in the evaluation, the performance overheads of Harmoni were much lower than FlexCore with a slow on-chip FPGA. While not shown here, we found that Harmoni can keep pace even with a main processing core running at twice its maximum frequency of 2.5GHz with small overheads. Because not all instructions on the main core triggers a tag operation, running the tag pipeline at a slower clock frequency does not directly translate to performance overheads for most monitoring techniques. The higher frequencies achievable by Harmoni allows it to be used with high performance processing cores running at a Gigahertz and more.

V. Related Work

Here, we briefly discuss how the Harmoni architecture is related and different from the previous work in run-time monitoring. In a high-level, Harmoni represents a unique design point in the context of the inherent trade-off between programmability and efficiency; Harmoni is much more general and programmable compared to dedicated hardware solutions for a single run-time monitoring technique while providing much higher performance and power efficiency compared to fully programmable approaches.

A. Hardware-Based Run-Time Monitoring

Run-time monitoring and tagging as methods to ensure various program properties have been extensively studied. This work uses the previously proposed monitoring techniques and also borrow ideas such as tagging, decoupled monitoring architecture, and software-controlled tables from existing hardware implementations. However, the goal of Harmoni is to enable a broad range of monitoring techniques on a single platform rather than realizing one particular monitor.

Feustel [22] argued that a complete tagged architecture can make software and systems more cost effective for performing practical computations and can deal with type and program safety issues in computer systems in a natural and transparent manner. This previous work introduced the notion of tagging, but only in the context of a fixed hardware extension.

Recently, the dynamic information flow tracking (or dynamic taint analysis) has been widely studied in the context of building more secure systems. DIFT [1] used a single-bit hardware tagging mechanism to track the flow of untrusted information in a system and to prevent program hijacking attempts that take advantage of program vulnerabilities. Using simple hardware support and tag memory management optimizations, DIFT was able to perform information flow tracking with negligible performance overheads. Similarly, Minos [23] tags individual words of memory data and implements Biba's low-water-mark integrity policy [24] on the use of memory data to stop attacks that attempt to corrupt control data in order to hijack program control flow.

Raksha [3] expanded upon DIFT by using multi-bit tags to support programmable and concurrent information flow security policies along with low-overhead security handlers that allow software to better manage detected errors. FlexiTaint [25] proposed a fully flexible DIFT implementation where a software-controlled table can be used to propagate and check taint information on each instruction. In that sense, FlexiTaint can be considered as supporting value tags in a way that the propagation and check policies are completely flexible. However, FlexiTaint is only designed to support a dynamic information flow tracking with flexible policies. The Harmoni architecture borrows the idea of software-controlled tables. However, Harmoni processor supports a wider range of monitoring techniques by enabling value, location, and object tags and also allowing them to be combined together.

Tiwari et al [26] extended DIFT to the hardware gate level. By restricting the ISA, using predicated execution, bounded loops, and an iteration-coupled load/store architecture, the authors designed a provably-sound information flow tracking system that is capable of tracking all explicit and implicit information flows within a computer system. Harmoni targets to monitor program-level behaviors and does not handle gate-level information flows.

The early DIFT implementations added tagging capabilities directly into a processing core pipeline. However, performing invasive hardware modifications to existing processor designs presented a major obstacle in deploying information flow tracking in practice due to the high hardware design and verification costs. The DIFT co-processor [27] proposed to reduce these costs by performing DIFT in a small decoupled co-processor. The DIFT co-processor was shown to be able to provide the same degree of security as the most complete integrated DIFT architecture, had lower performance overheads, and required fewer invasive changes to the baseline processing core. Harmoni uses a similar decoupled co-processor architecture, but support a wide range of monitoring schemes in addition to DIFT.

In addition to DIFT, a number of run-time monitoring schemes have been proposed to enable many types of capabilities such as fine-grained memory protection [4], array bound checking [5], [6], software debugging support [7], managed language support such as garbage collection [8], hardware error detection [9], etc. Harmoni targets to support many of them with a single hardware platform.

B. Programmable Monitoring Platforms

There have been recent efforts to build programmable run-time monitoring architectures, which can enable more than one monitoring scheme. Compared to these proposals, the Harmoni architecture is either more flexible or provides higher performance.

MemTracker [13] is a runtime monitoring approach for memory bug detection in which a hardware state machine uses a memory tag and memory operation to update the memory tag and check for memory bugs on each memory access instruction. In comparison to MemTracker, Harmoni is not restricted to monitoring for memory bugs and can perform more sophisticated monitoring functions by keeping track of tags for a larger portion of program state. Effectively, MemTracker only supports memory location tags with very simple update and check rules.

LBA [28] proposes to utilize a large number of processing cores in future microprocessors for run-time monitoring. More specifically, LBA augments each processor with hardware support for logging a main program trace and delivering it to another (otherwise idle) processing core for inspection. A software program running on this other core executes the monitoring task. Compared to LBA, Harmoni is less general, yet far more efficient and have better throughput because it performs monitoring entirely in hardware and avoids the area and power overheads of running general purpose instructions on a separate processing core.

FlexCore [12] is a hybrid architecture that combines a general-purpose processing core with a decoupled on-chip, bit-level reconfigurable fabric. The fabric can be reconfigured to perform a range of runtime monitoring functions in hardware to detect reliability and security errors. However, the low throughput of the bit-level programmable fabric used in the FlexCore architecture can cause it to have very high performance overheads on high-performance processing cores that can run at a high clock frequency. This work addresses this performance challenge by narrowing the scope of run-time monitoring functions and optimizing the architecture for tagging techniques.

VI. Conclusion

This paper proposed the Harmoni architecture, a high-performance and reconfigurable co-processor that can be used to implement program monitoring techniques based on tagging in hardware. We showed how a variety of runtime monitoring techniques can be mapped to the Harmoni hardware so as to check for memory bugs, security violations, and support the management of system resources. Harmoni presents a new design point on the spectrum between performance and flexibility for runtime monitoring approaches; by matching the common characteristics of monitoring approaches based on tagging, Harmoni can achieve very high performance without restricting the capabilities of the monitoring approaches. We evaluated the overheads of the Harmoni co-processor by building an RTL model and evaluated the application performance with Harmoni monitoring using RTL simulations. The evaluation results demonstrated that the Harmoni coprocessor takes moderate silicon area and has low overheads on program performance for a range of monitoring approaches.

Acknowledgments

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.

REFERENCES

[1] G. E. Suh, J. W. Lee, D. 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*, ser. ASPLOS-XI. New York, NY, USA: ACM, 2004, pp. 85–96. [Online]. Available: http://doi.acm.org/10.1145/1024393.1024404

[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*, February 2005.

[3] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *Proceedings of the 34th annual international symposium on Computer architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 482–493. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250722

[4] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-X. New York, NY, USA: ACM, 2002, pp. 304–316. [Online]. Available: http://doi.acm.org/10.1145/605397.605429

[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* 13^{th} *international conference on Architectural support for programming languages and operating systems*, 2008, pp. 103–114.

[6] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic, "Effective memory protection using dynamic tainting," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 284–292. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321673

[7] 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*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 224–. [Online]. Available: http://dl.acm.org/citation.cfm?id=998680.1006720

[8] J. A. Joao, O. Mutlu, and Y. N. Patt, "Flexible reference-counting-based hardware acceleration for garbage collection," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 418–428. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555806

[9] A. Meixner, M. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, dec. 2007, pp. 210 –222.

[10] 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* 39^{th} *International Symposium on Microarchitecture*, 2006.

[11] 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* 35^{th} *International Symposium on Computer Architecture*, June 2008.

[12] 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 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 137–148. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2010.17

[13] 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*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 273–284. [Online]. Available: http://dl.acm.org/citation.cfm?id=1317533.1318083

[14] D. Deng and G. Suh, "Precise exception support for decoupled run-time monitoring architectures," in *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, oct. 2011, pp. 437 –438.

[15] 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*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 245–258. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542504

[16] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware enforcement of application security policies using tagged memory," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 225–240. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855757

[17] M. Tiwari, B. Agrawal, S. Mysore, J. Valamehr, and T. Sherwood, "A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 94–105. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2008.4771782

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

[19] I. SPARC International, "The SPARC Architecture Manual Version 8," 1992.

[20] I. Coporation, "Intel Atom Processor Z510," 2008.

[21] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *Annual IEEE International Workshop on Workload Characterization*, 2001.

[22] E. A. Feustel, "On the advantages of tagged architecture," *Computers, IEEE Transactions on*, vol. C-22, no. 7, pp. 644 –656, july 1973.

[23] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 37. Washington, DC, USA: IEEE Computer Society, 2004, pp. 221–232. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2004.26

[24] K. Biba, M. C. B. MA., and U. S. A. F. S. C. E. S. Division, *Integrity Considerations for Secure Computer Systems*. Defense Technical Information Center, 1977. [Online]. Available: http://books.google.com/books?id=lAa4SgAACAAJ

[25] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, feb. 2008, pp. 173 –184.

[26] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '09. New York, NY, USA: ACM, 2009, pp. 109–120. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508258

[27] H. Kannan, M. Dalton, and C. Kozyrakis, "Decoupling dynamic information flow tracking with a dedicated coprocessor," in *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, 29 2009-july 2 2009, pp. 105 –114.

[28] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser, "Log-based architectures for general-purpose monitoring of deployed code," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ser. ASID '06. New York, NY, USA: ACM, 2006, pp. 63–65. [Online]. Available: http://doi.acm.org/10.1145/1181309.1181319