AFRL-RI-RS-TR-2015-143

FLEXIBLE TAGGED ARCHITECTURE FOR TRUSTWORTHY MULTI-CORE PLATFORMS

CORNELL UNIVERSITY

JUNE 2015

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2015-143 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
WILMAR SIFRE
Work Unit Manager

/ S /
MARK LINDERMAN
Technical Advisor, Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JUNE 2015	FINAL TECHNICAL REPORT	OCT 2010 – DEC 2014

4. TITLE AND SUBTITLE

FLEXIBLE TAGGED ARCHITECTURE FOR TRUSTWORTHY MULTI-CORE PLATFORMS

5a. CONTRACT NUMBER
FA8750-11-2-0025

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62788F

6. AUTHOR(S)

Gookwon Suh

5d. PROJECT NUMBER
T2ST

5e. TASK NUMBER
CO

5f. WORK UNIT NUMBER
RN

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Cornell University
Office of Sponsored Programs
373 Pine Tree Road
Ithaca, NY 14850

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2015-143

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This project aimed to realize the full potential of fine-grained tagging scheme for monitoring and detecting errors and attacks on software programs. In this context, the project developed flexible hardware tagged architecture designs that can perform a wide range of tagging techniques with low overhead. In order to enable the fine-grained tagging techniques for real-time systems, the project also developed both static analysis and dynamic enforcement techniques to provide a strong guarantee for the worst-case program execution time with tagging. The evaluation studies through simulations and FPGA prototypes suggest that the proposed tagged architecture can provide both flexibility and efficiency, and also be safely applied to hard real-time systems with strict deadlines.

15. SUBJECT TERMS
Security, tagged architecture, run-time monitoring, real-time systems

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WILMAR SIFRE**
U	U	U	UU	47	19b. TELEPHONE NUMBER (*Include area code*) **315-330-2075**

TABLE OF CONTENTS

Section		Page
List of Figures		iii
List of Tables		iv
1.0	SUMMARY	1
2.0	INTRODUCTION	1
3.0	METHODS, ASSUMPTIONS, AND PROCEDURES	3
3.1	Fine-Grained Run-Time Tagging	3
3.1.1	Tag (Meta-data) Types	4
3.1.2	Tag Operations	4
3.2	Tag-Based Run-Time Monitoring Examples	5
3.3	FlexCore: Reconfigurable Tagged Architecture	7
3.3.1	Core-Fabric Interface	8
3.3.2	Meta-Data Tag Memory Hierarchy	10
3.3.3	Reconfigurable Fabric Design	10
3.4	Harmoni: Accelerator for High-Performance Tagged Architecture	10
3.4.1	Programmable Tagging Engine	11
3.4.2	Tag Processing Pipeline	13
3.4.3	Examples	14
3.4.4	Limitations	16
3.5	Worst-Case Analysis for Tagging on Real-Time Systems	16
3.5.1	Implicit Path Enumeration	16
3.5.2	Bound for Sequential Monitoring	17
3.5.3	FIFO Model	18
3.5.4	Mixed-Integer Linear Programming (MILP) Formulation	19
3.5.5	Tool Flow	21
3.6	Opportunistic Tagging for Real-Time Systems	21
3.6.1	Measuring Slack	22
3.6.2	Dropping Tagging Operations	23
3.6.3	Hardware Support	24
4.0	RESULTS AND DISCUSSION	27
4.1	FlexCore Architecture	27
4.1.1	Methodology	27
4.1.2	Area, Power, and Frequency	28
4.1.3	Performance	29
4.2	Harmoni Architecture	30
4.2.1	Methodology	30
4.2.2	Area, Power, and Frequency	30
4.2.3	Performance	31

4.3 WCET Analysis for Real-Time Systems Tagging 32
4.3.1 Methodology 32
4.3.2 WCET Results 33
4.4 Effectiveness of Opportunistic Tagging on Real-Time Systems 34
4.4.1 Methodology 34
4.4.2 Amount of Monitoring Performed 35
4.4.3 FlexCore Results 35

5.0 CONCLUSIONS 36

6.0 REFERENCES 37

LIST OF FIGURES

1 Tag-based parallel run-time monitoring 3
2 FlexCore architecture block diagrams 7
3 High level block diagram of the Harmoni pipeline 12
4 Run-time monitoring techniques mapped to the Harmoni co-processor 15
5 Toolflow for WCET estimation of parallel tagging 21
6 Dynamic slack and total slack 22
7 An example for dynamic slack changes 23
8 Hardware modules for slack tracking and drop decisions 24
9 Metadata invalidation module (MIM) 25
10 Metadata filtering module (MFM) 26
11 Block diagram of hardware architecture for opportunistic tagging on hard real-time systems 27
12 The performance overheads of the Harmoni co-processor 32

LIST OF TABLES

1 Tag types and operations for run-time monitoring functions 6
2 The FlexCore interface between the core and the tagging fabric 8
3 The area, power, and frequency of the FlexCore architecture 28
4 The performance overhead comparisons between ASICs and FlexCore 29
5 The area, power, and frequency of the Harmoni architecture with different maximum tag sizes 31
6 Estimated and observed WCET (clock cycles) with and without monitoring 33
7 Monitoring event type breakdown in percentage 34
8 Percentage of checks that are not dropped or filtered 34
9 Monitoring event breakdown for an FPGA-based monitor (FlexCore) 35
10 Percentage of checks that are not dropped or filtered for an FPGA-based monitor (FlexCore) 35

1.0 SUMMARY

As we place more responsibilities on computing systems, secure and reliable operations are becoming increasingly important. In this context, recent studies showed that fine-grained run-time monitoring of program behaviors based on tags (meta-data) can greatly enhance the trustworthiness of computing systems through checking security policies at run-time.

Unfortunately, performing such fine-grained tagging in software is often too expensive for deployed systems. On the other hand, dedicated hardware engines can only support one particular tagging scheme even though their run-time overhead is quite low. In practice, such dedicated hardware engines are difficult to justify because of their high development and silicon costs. To address this challenge, this project developed flexible tagged architecture platforms that leverage reconfigurable fabric such as FPGAs or hardware accelerators to perform a wide range of tagging techniques with low overhead.

In addition to enabling a new trade-off point between flexibility and efficiency, this project also developed both static and dynamic techniques to apply fine-grained tagging techniques for real-time systems with strict deadlines. The static technique can analyze the impact of tagging on the worst-case execution time. The dynamic technique enforces that a monitored program meets real-time deadlines by adjusting the amount of tagging at run-time.

The proposed tagged architecture designs are prototyped on an FPGA board and shown to be able to support existing tagging techniques as well as a new tagging scheme that was developed to build a secure zero-kernel operating system. Overall, the project outcome enables fine-grained tagging techniques to be widely deployed with low overhead for both regular and real-time systems.

2.0 INTRODUCTION

Recent studies show that tagged architectures where each word in memory is associcated with a tag (meta-data) can greatly enhance the trustworthiness of computing systems through various types of fine-grained run-time checks and enforcements. As an example, Dynamic Information Flow Tracking (DIFT) is a security technique that can detect a large class of software attacks by tagging potentially malicious values from I/O as *tainted*, tracking the tainted values during execution, and checking the use of tainted values. DIFT was first introduced as a general technique to detect low-level software attacks such as buffer overflows and format string attacks [29]. Other researchers have shown that DIFT is also effective against high-level attacks such as Structured Query Language (SQL) injection and cross-cite scripting [8]. Similarly, a recent project named Trust management, Intrusion-tolerance, Accountability, and Reconstitution Architecture (TIARA) uses the tagged architecture to build strong compartments and enable zero-kernel operating systems [27].

In fact, researchers have shown that tagging is a general mechanism to enhance trust in computation even beyond security against software attacks. To name a few, fine-grained memory protection [34], array bound checking [11], software debugging support [37], garbage collection [17], and hardware fault detection [22] can all be considered as a form of tagging with run-time checks.

This project aimed to realize the full potential of fine-grained tagging by developing a hardware tagged architecture platform that can efficiently support a wide range of tagging techniques on both regular and real-time systems. Tagging techniques may also be implemented in software by inserting additional instructions for tag operations. However, fine-grained tagging such as DIFT or TIARA is simply too expensive for a wide deployment in software. For example, a software implementation of DIFT is reported to incur an average slowdown of 3.6 times [25] even in the most efficient implementation with various optimizations. On the other hand, hardware DIFT resulting in less than a 2% slowdown [29].

Unfortunately, existing tagged architecture designs suffer from a few major limitations. First of all, tagged architectures are often designed as custom hardware that can only support one particular type of tagging technique. As an example, the DIFT hardware can be used to detect a class of software attacks but is not applicable to other techniques such as memory protection, software debugging, hardware fault detection, etc. This custom tagged architecture cannot be fixed or changed after fabrication even if bugs or vulnerabilities are identified. This inflexibility along with high development costs for custom hardware poses significant challenges in deploying tagged architectures in real-world microprocessors. For example, a modern microprocessor development including verification efforts takes several years from an initial design to production. As a result, deploying a new tagging technique takes years to build new hardware. Processor vendors, especially Commercial Off The Shelf (COTS) vendors, are also reluctant to incorporate custom hardware just for one type of tagging unless the technique is widely popular.

For real-time systems, tagged architectures also introduce a new challenge that is not addressed by today's state-of-the-art. In hard real-time, the worst-case execution time of a monitored program must be guaranteed within a certain bound in order to ensure that the program produces results in time. Unfortunately, run-time tagging operations can interfere with monitored programs and change their execution time. Therefore, in order to be applicable to hard real-time systems, there needs to be new techniques to analyze and ensure the worst-case execution time of a tagged architecture framework.

In this project, we developed new hardware tagged architectures that can provide both flexibility and efficiency for a wide-range of tagging techniques. At the same time, we also developed techniques to analyze the impact of tagging on the worst-case execution time and a run-time framework to ensure that a tagged system can meet hard real-time deadlines. The tagged architecture designs are prototyped on an FPGA and applied to a new tagging scheme that was developed by an operating systems team at University of Idaho. The following summarizes the key technical contributions from this project.

- *Flexible and efficient tagged architecture*: The project developed two tagged architecture designs that can support a wide range of tagging techniques with the overhead close to dedicated hardware. In the first design, tagging techniques are implemented using reconfigurable fabric such as FPGAs. This design provides full flexibility of generic reconfigurable fabric. Yet, the clock frequency of the generic fabric does not scale up for high-performance processors running at a few GHz. The second design uses a specialized hardware accelerator to support high-performance processing cores for tagging schemes that follow certain restrictions.

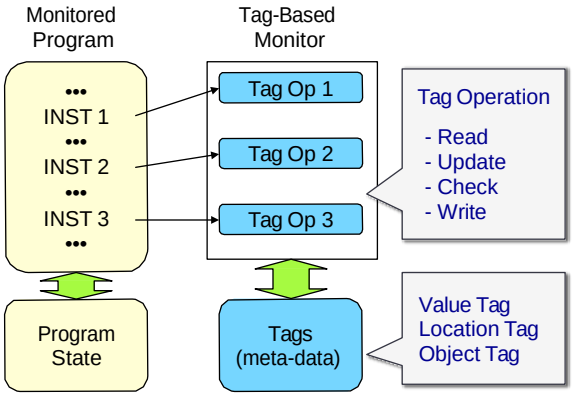


Figure 1: Tag-based parallel run-time monitoring

- *Tagging for real-time systems*: The project developed both static analysis and dynamic enforcement techniques to enable tagging on real-time systems. The static analysis enables system designers to estimate the impact of tagging on the worst-case execution time of a monitored program. The dynamic enforcement approach enables tagging for hard real-time systems even when the target system does not have enough slack for full monitoring by performing partial protection.
- *FPGA prototype systems*: The project implemented the proposed tagged architecture designs on an FPGA board for both single-core and multi-core systems. These prototypes are used to test the functionality of the proposed designs. The HDL (Hardware Description Language) code from these prototypes are also used to evaluate hardware overhead such as performance, area, and power consumption using hardware design tools.
- *Tagged real-time operating systems*: To demonstrate the flexibility of the architecture, we implemented multiple tagging techniques on the proposed tagged architecture. The study showed that the architecture can support existing tagging techniques such as array bound checks, uninitialized memory checks, dynamic information flow tracking, etc. as well as a new tagging technique that was developed during the project to support a secure zero-kernel operation system.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Fine-Grained Run-Time Tagging

The main challenge in designing a programmable and high-performance architecture for tag-based run-time monitoring schemes lies in identifying common functions for a broad class of monitors so that the programmability can be limited without significantly sacrificing functionality.

Figure 1 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” or “meta-data” 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" or "monitoring task" 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 the following discussion, we use the term "co-processor" or "monitor" to refer to the tagging engine for run-time monitoring.

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.

3.1.1 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 Arithmetic Logic Unit (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.

3.1.2 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.

3.2 Tag-Based Run-Time 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) [29]: DIFT is a security protection technique that prevents common software exploits from taking over a vulnerable program by tracking and limiting uses of untrusted inputs. For example, typical attacks that changes a program control flow can be detected by preventing 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 inputs. DIFT uses operating system support to set the tag (taint) input-derived data, and then transparently 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

Table 1: Tag types and operations for run-time monitoring functions

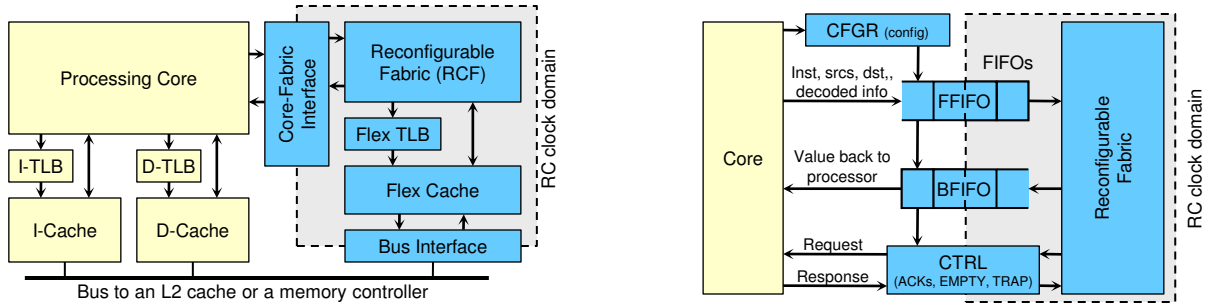
Monitor	Trigger	Action
DIFT (1-bit value tag)	ALU instructions	Tag(reg_dest) := Tag(reg_src1) or Tag(reg_src2)
	LOAD instructions	Tag(reg_dest) := Tag(mem_addr)
	STORE instructions	Tag(mem_addr) := Tag(reg_dest)
	JUMP instructions	check reg_src1 != "1"
UMC (1-bit location tag)	LOAD instructions	check 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

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) [32]: 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, here, 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



(a) Overview (b) Core-fabric interface

Figure 2: FlexCore architecture block diagrams

operands to the output register to keep track of tags for an updated pointer on an pointer arithmetic operation.

Reference counting (RC) [17]: 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.

3.3 FlexCore: Reconfigurable Tagged Architecture

To enable run-time tagging techniques in an efficient and programmable manner, we first designed a new tagging architecture based on reconfigurable fabric such as FPGAs. This architecture is named FlexCore [9]. Here, we illustrate the overall architecture in the context of a single-issue processor without multi-threading. For multi-threaded programs, the architecture needs to be augmented to handle a coherence issue between program data and meta-data [31, 18].

Figure 2(a) shows the high-level block diagram of the FlexCore architecture. The yellow (light) rectangles represent components in traditional microprocessors and the blue (dark) rectangles represent new components for FlexCore. In a high-level, the architecture closely resembles the co-processing model that is described in the previous subsection. The main processing core forwards its execution trace to the reconfigurable fabric for parallel tag operations and receives signals from the fabric through the interface module. The architecture provides a separate memory subsystem for meta-data with its own L1 cache and optionally a Translation Lookaside Buffer (TLB) if virtual memory is supported. The meta-data shares a lower level memory hierarchy such as an L2 cache and the main memory with program data.

The architecture is carefully designed to exploit the common characteristics of run-time tagging techniques without restricting the specifics of the tagging operations. For example, unlike tradi-

Table 2: The FlexCore interface between the core and the tagging fabric

Function	Module	Field	Description	Bits
Config	CFGR	FFIFO	Select a FIFO behavior for each instruction type: 1) ignore, 2) accept only if not full, 3) accept and proceed, 4) accept and wait for an acknowledgement. Contains 2 bits for each of the main 32 instruction types (SPARC prototype).	64
Core To Fabric	CTRL	PACK	Acknowledgement for a trap signal from the co-processor.	1
	FFIFO	PC	Program counter.	32
		INST	Undecoded instruction.	32
		ADDR	Address for a load/store.	32
		RES	Result of an instruction.	32
		SRCV1	Source operand 1 value.	32
		SRCV2	Source operand 2 value.	32
		COND	Condition codes that affect instruction processing.	4
		BRANCH	Computed branch direction information.	1
		OPCODE	Decoded instruction opcode.	5
		DECODE	Miscellaneous decoded signals.	32
		EXTRA	Extra processor control signals.	32
		SRC1	Decoded Source1 register number.	9
		SRC2	Decoded Source2 register number.	9
		DEST	Decoded Destination register number.	9
Fabric To Core	CTRL	CACK	Acknowledgement for FFIFO.	1
		EMPTY	A signal to indicate that there is no pending instruction in the co-processor.	1
		TRAP	Raise en exception.	1
	BFIFO	VAL	A return value on a 'read from co-processor' instruction.	32

tional FPGA co-processors, the main core forwards its execution trace without explicit intervention from software. The architecture is also optimized for *bit operations* that are common in run-time tagging. The reconfigurable fabric operates at a bit granularity and the meta-data cache supports bit-level writes.

While the fine-grained reconfigurable fabric is a good fit for typical meta-data tag computations in run-time monitoring schemes, the bit-level reconfigurability is a poor match for coarse-grained structures such as memory arrays. The FlexCore architecture handles inefficiencies in fine-grained reconfigurable fabric with a set of optimizations.

- *Specialized hardware modules*: The architecture incorporates a set of dedicated hardware units for functions that are common across many tagging schemes. These modules include the core-fabric interface, TLB, cache, and meta-data register file.
- *Filtering and pre-processing*: The core-fabric interface reduces the workload of the reconfigurable fabric by filtering out unnecessary types of instructions and performing decoding of instructions.
- *Decoupled executions*: The reconfigurable fabric is decoupled from the main core through First Input First Output (FIFO) queues. The processing core forwards its execution trace, but does not need to wait for an acknowledgment from the tagging fabric in most cases. This decoupling hides latencies from co-processing as well as communications across clock domains. The clock domain of the reconfigurable fabric is carefully selected to include the TLB and the cache so that TLB and cache hits do not require cross-domain communications.

3.3.1 Core-Fabric Interface The reconfigurable fabric communicates with the main core 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 details of the core-to-fabric interface are also listed in Table 2.

The core-to-fabric interface works to enable fine-grained instruction communication between the core and reconfigurable fabric. The processing core sends its execution trace to the co-processor using the FIFO interface, so that the fabric 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 opcode, source register numbers, and a destination register number, in order to reduce the burden on the reconfigurable fabric. For comparison, we found that our DIFT prototype can run 30% faster by performing the instruction decoding for operands and control signals on the core side.

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 tagging schemes, the main core does not need to wait for the reconfigurable fabric because an exception from the co-processor does not need to be precise. For example, typical tagging schemes terminate a program if a tag check fails. There is no need to support a restart for these scheme. If an instruction requires a value from the fabric as in the "read from co-processor" instruction or if an exception from the fabric 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 reconfigurable fabric 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 waits for a completion of the Flex fabric 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 fabric provides a signal (EMPTY) to indicate whether there are any pending instructions in the co-processor. The reconfigurable fabric 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 reconfigurable fabric can easily support custom

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

instructions on the main core for each tagging scheme. For example, in order to implement an instruction to set a configuration register within the reconfigurable fabric, the fabric can be programmed to update the register on a particular instruction encoding.

3.3.2 Meta-Data Tag Memory Hierarchy For meta-data used by the reconfigurable fabric for bookkeeping, the reconfigurable fabric 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 reconfigurable fabric share the lower-level memory hierarchy such as an L2 cache and main memory.

The meta-data cache is almost identical to regular data caches except for the capability to write at a bit granularity. Meta-data cache reads return 32-bit words as in regular caches. For writes, the meta-data cache is given a 32-bit write enable mask in addition to an address and a data word, and only updates 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.

3.3.3 Reconfigurable Fabric Design The high-level FlexCore architecture is independent from the micro-architecture of the reconfigurable fabric and is applicable to various types of fabrics. In our prototype design that was evaluated, we use a standard Lookup Table (LUT) based FPGA architecture, specifically the Xilinx Virtex-5, which includes standard Configurable Logic Blocks (CLBs) with LUTs and flip-flops. The FPGA fabric is chosen over other coarse-grained fabrics because typical tagging schemes perform bit-level operations. Our reconfigurable fabric also includes an embedded meta-data register file for tags, which is implemented with custom hardware and has an 8-bit shadow register for each general-purpose architecture register in the main core. The register file provides an efficient way to keep tags for registers in a similar way that Static Random Access Memory (SRAM) banks in commercial FPGAs provide efficient memory blocks.

3.4 Harmoni: Accelerator for High-Performance Tagged Architecture

The FlexCore architecture provides an efficient and flexible platform that is capable of performing a wide range of tagging schemes. In particular, our prototype evaluations showed that FlexCore can perform tagging operations with a few tens of percent overhead for processing cores running at a moderate clock frequency (500MHz). However, today's FPGA reconfigurable fabric runs at a much lower frequency compared to processing cores, and the performance overhead of FlexCore is noticeably higher for high-performance processors running at a few GHz.

To enable programmable tagged architecture for high-performance processors, we designed a new custom accelerator architecture for tagging, named Harmoni [10]. The Harmoni architecture supports fine-grained tagging 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. Similar to FlexCore, the forwarded instructions are selected based on the opcode. The execution trace includes the opcode, register indexes of 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 updates 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 tagging operations 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.

3.4.1 Programmable Tagging Engine 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 tagging scheme. The Harmoni co-processor is designed to efficiently support 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 3 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.

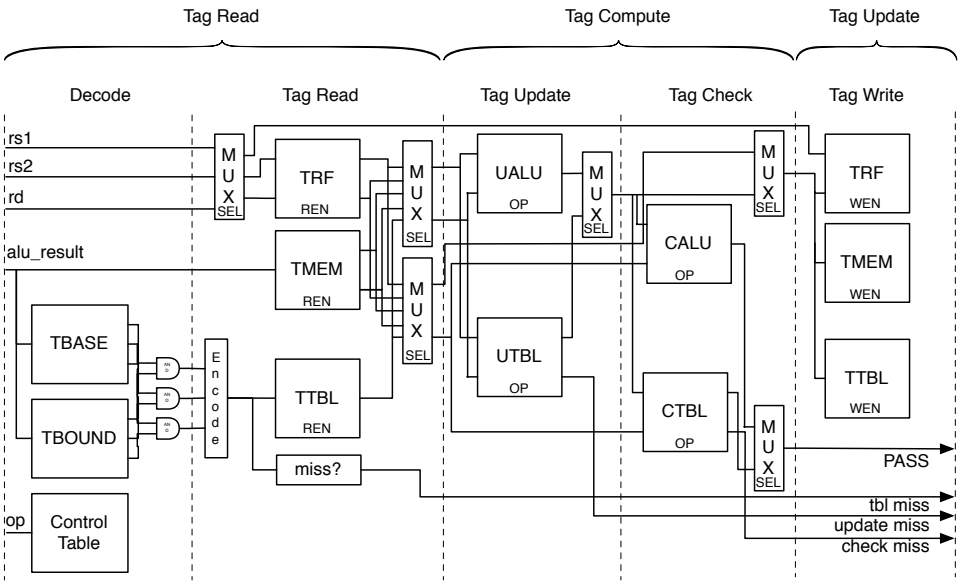


Figure 3: High level block diagram of the Harmoni pipeline

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 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 [23, 36, 30] 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.

3.4.2 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 3. 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. This control table (CONTBL) entry specifies indexes to the tag register file where tags will be read from. 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.

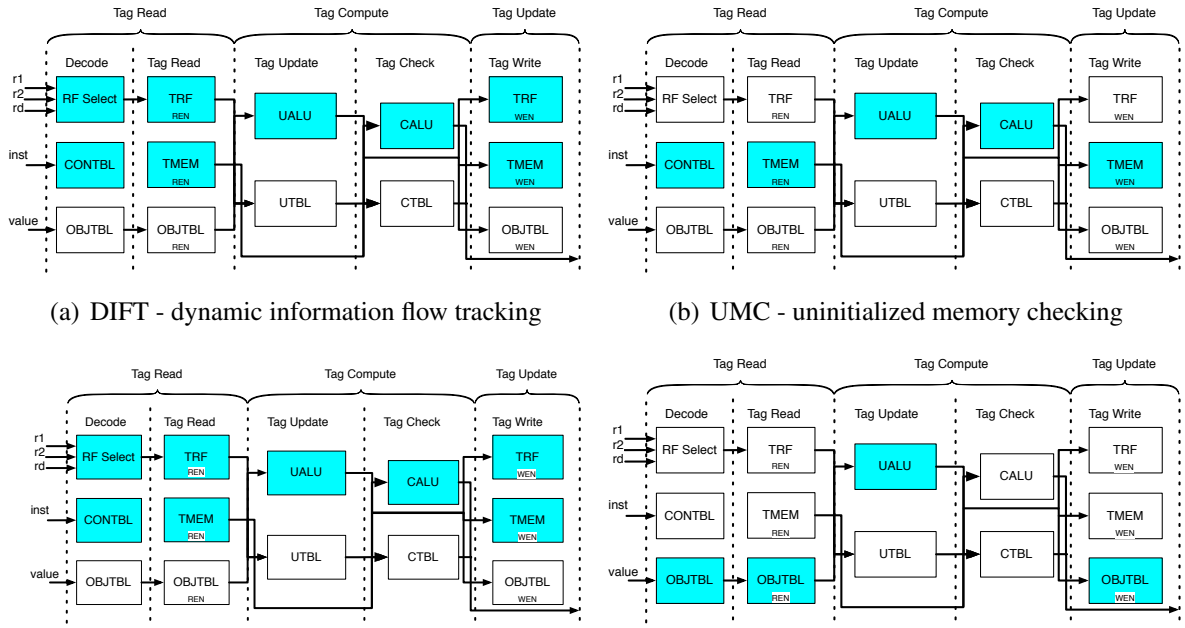
In the fourth stage of the pipeline, the tag is checked against invariants. The CALU takes the updated tag along with another tag from the TRF, the TMEM, or the OBJTBL, and performs a unary or binary comparison to determine if an exception should be raised. The CTBL uses the same input tags to perform 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.

3.4.3 Examples Figure 4 shows how the run-time tagging schemes 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 4(a). In DIFT, the tags that encode taint information are propagated on ALU and memory instructions, and 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 4(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



(a) DIFT - dynamic information flow tracking

(b) UMC - uninitialized memory checking

(c) BC - memory bounds checking

(d) RC - hardware support for reference counting

Figure 4: Run-time monitoring techniques mapped to the Harmoni co-processor

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 4(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 4(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.

3.4.4 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. 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.

3.5 Worst-Case Analysis for Tagging on Real-Time Systems

The FlexCore and Harmoni architectures enable flexible tagging schemes with low overhead. However, applying run-time tagging scheme for safety-critical real-time systems is still challenging due to the lack of timing guarantees. The development of a safety-critical real-time system requires an estimate of the worst-case execution time (WCET) of each task in order to ensure that tasks meet the system's real-time deadlines. Yet, previous studies on parallel tagging have only focused on average slowdowns with no worst-case guarantee.

Here, we present a method for estimating the impact of parallel run-time tagging operations on WCET. We first review the traditional ILP-based analysis for a sequential program execution, and show that this analysis can be extended to incorporate the overhead of tagging if the worst-case increase in execution time can be estimated for each basic block. Then, we discuss how to estimate the worst-case stalls for parallel tagging engines, which happen when the FIFO is full and cannot take a forwarded instruction.

3.5.1 Implicit Path Enumeration Most of the WCET analysis techniques today rely on an ILP (Integer Linear Programming) formulation that is obtained from implicit path enumeration techniques [21]. In this method, a program is converted to a control flow graph (CFG). From the

control flow graph, we formulate an ILP problem that maximize the following metric:

$$t = \sum_{B \in \mathcal{B}_{CFG}} N_B \cdot c_{B,max} \tag{1}$$

where $\mathcal{B}_{CFG}$ is the set of basic blocks in the control flow graph. N_B is the number of times block B is executed and $c_{B,max}$ is the maximum number of cycles to execute block B. The maximum value of t is the WCET of the task. To account for the fact that only certain paths in the graph will be executed, a set of constraints are placed on N_B. For example, on a branch, only one of the branches will be taken on each execution of the block. A variable can be assigned to each edge corresponding to the number of times that edge is taken. The number of times edges out of the block are taken must equal the number of times the block is executed. Similarly, the number of times edges into the block are taken must equal the number of times the block is executed. Various methods have been developed to create additional constraints to convey other program behavior [21, 33].

Integer linear programming is an attractive optimization technique for this problem because the solution found is a global optimum. In addition, many aspects of program and architecture behavior can be described by adding constraints to the ILP problem. Several open source and commercial ILP solvers exist which can solve the formulated ILP problem. Thus, in developing a method for estimating the WCET of parallel run-time monitoring, we look to build upon this ILP framework.

The IPET-based ILP formulation can be extended in a straightforward fashion to incorporate run-time tagging overheads if we have the maximum (worst-case) stall cycles caused by tagging operations for each basic block. This can be achieved by maximizing

$$t = \sum_{B \in \mathcal{B}_{CFG}} N_B \cdot (c_{B,max} + s_{B,max}) \tag{2}$$

Here, $s_{B,max}$ represents the maximum number of cycles that block B is stalled due to tagging. In this sense, the challenge in WCET analysis with tagging lies in determining $s_{B,max}$. The rest of this subsection addresses this problem.

3.5.2 Bound for Sequential Monitoring One way to determine a conservative bound on the worst-case stall cycles due to tagging is to consider monitoring based on tagging in a sequential manner. In this approach, the tagging operation is run in-line with the main task on the same core rather than in parallel. That is, after each instruction that would be forwarded, the tagging operation is run on the main core before the main task resumes execution. In the following discussion, we refer to this sequential approach as *sequential monitoring*.

In sequential monitoring, the WCET estimate can be obtained from a traditional method by analyzing one program that contains both main computation and tagging operations. The resulting WCET can be considered as a conservative bound for parallel tag-based monitoring because it models the case where every forwarded instruction causes the main core to stall. However, this bound is extremely conservative as it does not account for the FIFO buffering or the parallel execution of the tagging engine, which is critical for efficient run-time tagging.

3.5.3 FIFO Model To obtain tighter WCET bounds, we need to model the FIFO. The main task can continue its execution as long as a FIFO entry is available, but needs to stall on a forwarded instruction if the FIFO is full. The WCET model needs to capture the worst-case (maximum) number of entries in the FIFO at each forwarded instruction and determine how many cycles the main task may be stalled due to the FIFO being full. Here, we propose a mathematical model to express the load in the FIFO and estimate the worst-case stalls.

In this approach, the original control flow graph must be transformed so that each node contains at most one forwarded instruction which is located at the end of the code sequence represented by the node. This transformed graph is called a *monitoring flow graph (MFG)*. Intuitively, the analysis needs to consider one forwarded instruction at a time in order to model the FIFO state on each forwarded instruction and capture all potential stalls from tagging for monitoring.

To model how full the FIFO is, we define the concept of *monitoring load*. The monitoring load is the number of cycles required for the tagging engine to process all outstanding entries in the FIFO at a given point in time. The monitoring load increases when a new instruction is forwarded by the main task, and decreases as the tagging engine processes forwarded instructions. For simplicity, the increase in monitoring load for any forwarded instruction is conservatively assumed to be the worst-case (maximum) execution time among all possible tagging operations. This maximum, $t_{M,max}$, can be obtained from the WCET analysis of the tagging operations. We make this simplification because it is difficult to model the FIFO mathematically at an entry-by-entry level. With this simplification, each FIFO entry is identical and so the monitoring load fully represents the state of the FIFO. The monitoring load cannot be negative and is upper-bounded by the maximum monitoring load the FIFO can handle, l_{max}. The maximum monitoring load is the number of FIFO entries, n_F, multiplied by the increase in monitoring load for one forwarded instruction, $t_{M,max}$.

In our context, we need to determine the worst-case (maximum) monitoring load at the node boundaries in the MFG. For a given node, M, in the MFG, we define li_M as the monitoring load coming into the node and lo_M as the monitoring load exiting the node. The change in monitoring load for the node is denoted by Δl_M. The maximum Δl_M can be calculated as the difference between the WCET of a monitoring task that corresponds to M and the minimum execution cycles of the node, $c_{M,min}$:

$$\Delta l_M = \begin{cases} t_{M,max} - c_{M,min}, & \text{forwarded inst. } \in M \\ -c_{M,min}, & \text{no forwarded inst. } \in M \end{cases} \tag{3}$$

In order to ensure that the analysis is conservative in estimating the worst-case (maximum) stalls, we use the best-case (minimum) execution time for the main task here.

Because the monitoring load is bounded by zero and the maximum load that the FIFO can handle, l_{max}, the monitoring load coming out of a node is

$$\begin{aligned} lo_M = &\begin{cases} 0, & li_M + \Delta l_M < 0 \\ li_M + \Delta l_M, & 0 \le li_M + \Delta l_M \le l_{max} \\ l_{max}, & li_M + \Delta l_M > l_{max} \end{cases} \\ l_{max} = &n_F \cdot t_{M,max} \end{aligned} \tag{4}$$

The worst-case monitoring load entering node M, li_M, is the largest of the output monitoring loads among nodes with edges pointing to node M. Let $\mathcal{M}_{\text{prev}}$ represent the set of nodes with edges pointing to node M. Then,

$$li_M = \max_{M_{prev} \in \mathcal{M}_{prev}} lo_{M_{prev}} \tag{5}$$

The above equations describe the worst-case monitoring load at each node boundary. A stall from tagging occurs when a forwarded instruction is executed but there is no empty entry in the FIFO buffer. In terms of monitoring load, if a node would add monitoring load that would cause the resulting total load to exceed l_{max}, then a stall occurs. The number of cycles stalled, s_M, is the number of cycles that this total exceeds l_{max}. That is,

$$s_M = \begin{cases} 0, & li_M + \Delta l_M < l_{max} \\ (li_M + \Delta l_M) - l_{max}, & li_M + \Delta l_M \geq l_{max} \end{cases} \tag{6}$$

Then, the worst-case stall cycles due to tagging for each MFG node can be obtained by maximizing the sum of the s_M across all possible execution paths:

$$\max \sum_{M \in \mathcal{M}_{MFG}} s_M \tag{7}$$

where $\mathcal{M}_{MFG}$ is the set of nodes in the MFG. Once the worst-case stalls for each MFG node is found, the worst-case stalls for a CFG node, $s_{B,max}$, can be computed by simply summing the stalls from the corresponding MFG nodes. We note that since the monitoring load is always conservative in representing the FIFO state, no timing anomalies are exhibited by this analysis. That is, determining the individual worst-case stalls results in the global worst-case stalls.

3.5.4 Mixed-Integer Linear Programming (MILP) Formulation The proposed FIFO model requires solving an optimization problem to obtain the worst-case stalls, where the input and output monitoring loads, li_M and lo_M, and the stalls from tagging, s_M, need to be determined for each node. Here, we show how the problem can be formulated using MILP. Although the equations for lo_M and s_M are non-linear, they are piecewise linear. Previous work has shown that linear constraints for piecewise linear functions can be formulated using MILP [28]. In the following constraints, all variables are assumed to be lower bounded by zero unless otherwise specified, as is typically assumed for MILP.

First, a set of variables, lo' and s', are created to represent the unbounded versions of lo and s. For readability, the per block subscript M has been omitted.

$$\begin{aligned} s' &= li + \Delta l - l_{max},\ s' \in (-\infty, \infty) \\ lo' &= li + \Delta l,\ lo' \in (-\infty, \infty) \end{aligned} \tag{8}$$

The following piecewise linear function calculates s from s'.

$$s = f(s') = \begin{cases} 0, & s' < 0 \\ s', & s' \geq 0 \end{cases} \tag{9}$$

This function can be described in MILP using the following set of constraints.

$$\begin{aligned} a_s\lambda_0 + b_s\lambda_2 =& s' \\ \lambda_0 + \lambda_1 + \lambda_2 =& 1 \\ \delta_1 + \lambda_2 \leq& 1 \\ \delta_2 + \lambda_0 \leq& 1 \\ \delta_1 + \delta_2 =& 1 \\ b_s\lambda_2 =& s \end{aligned} \tag{10}$$

where a_s is chosen to be less than the minimum possible value of s' and b_s is chosen to be greater than the maximum possible value of s'. The choice of a_s and b_s is arbitrary as long as it meets these requirements. λ_i are continuous variables and δ_i are binary variables. In this set of constraints, s' is expressed as a sum of the endpoints of a segment of the piecewise function. The δ_i variables ensure that only the segment corresponding to s' is considered. $\delta_1 = 1$ corresponds to the $s' < 0$ segment of $f(s')$ and $\delta_2 = 1$ corresponds to the $s' \geq 0$ segment of $f(s')$. The λ_i variables represent exactly where s' falls on the domain of that segment. s can be calculated using this information and the values of the function at the segment endpoints.

Similarly, lo can be bound between 0 and l_{max} by using the following set of constraints.

$$\begin{aligned} a_l\lambda_3 + l_{max}\lambda_5 + b_l\lambda_6 =& lo' \\ \lambda_3 + \lambda_4 + \lambda_5 + \lambda_6 =& 1 \\ 2\delta_3 + \lambda_5 + \lambda_6 \leq& 2 \\ 2\delta_4 + \lambda_3 + \lambda_6 \leq& 2 \\ 2\delta_5 + \lambda_3 + \lambda_4 \leq& 2 \\ \delta_3 + \delta_4 + \delta_5 =& 1 \\ l_{max}\lambda_5 + l_{max}\lambda_6 =& lo \end{aligned} \tag{11}$$

As before, a_l and b_l are chosen such that $lo' \in (a_l, b_l)$. Again, λ_i are continuous variables and δ_i are binary variables.

Finally, for each node, the input monitoring load li_M must be determined. li_M depends on the previous nodes, $\mathcal{M}_{\text{prev}}$. If there is only one edge into the node, then li_M is simply

$$li_M = lo_{M_{prev}} \tag{12}$$

When there is more than one edge into node M, one set of constraints is used to lower bound li_M by all $lo_{M_{prev}}$.

$$li_M \geq lo_{M_{prev}},\ \forall M_{prev} \in \mathcal{M}_{\text{prev}} \tag{13}$$

Then, another set of constraints upper bounds li_M by the maximum $lo_{M_{prev}}$,

$$\begin{aligned} li_M - b \cdot \delta_{M_{prev}} \leq& lo_{M_{prev}},\ \forall M_{prev} \in \mathcal{M}_{\text{prev}} \\ \sum_{M_{prev} \in \mathcal{M}_{\text{prev}}} \delta_{M_{prev}} =& |\mathcal{M}_{\text{prev}}| - 1 \end{aligned} \tag{14}$$

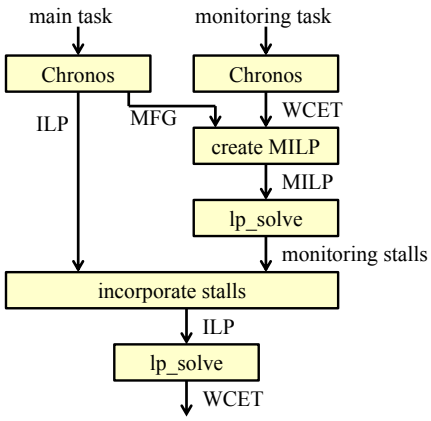


Figure 5: Toolflow for WCET estimation of parallel tagging

where b is chosen to be greater than $[\max(lo_{M_{prev}}) - \min(lo_{M_{prev}})]$ and $|\mathcal{M}_{\text{prev}}|$ is the number of nodes with edges pointing to M. δ_i are binary variables. The use of the binary variables δ_i and the second constraint ensure that li_M is only upper bound by one of the $lo_{M_{prev}}$. In order for all constraints to hold, this must be the maximum $lo_{M_{prev}}$. Together with the lower bound constraints, these constraints result in $li_M = \max(lo_{M_{prev}})$.

3.5.5 Tool Flow Our toolflow for the proposed WCET estimate method is shown in Figure 5. We first use Chronos [20], an open source WCET tool, to estimate the WCET for the main task and the monitoring tasks. We also modified Chronos to produce a MFG of the main task. This MFG and the monitoring task WCET are used to produce an MILP formulation. This MILP problem is solved using lp_solve [4], which produces the worst-case monitoring stall cycles for each forwarded instruction. These stalls from tagging are combined into the ILP formulation that is originally generated for the main task to estimate the overall WCET with parallel run-time tagging. Although we use Chronos and lp_solve for our implementation, these components can be replaced with any WCET estimation tool and LP solver respectively.

3.6 Opportunistic Tagging for Real-Time Systems

The WCET estimate method in the previous subsection provides a way to analyze the impact of run-time tagging schemes on a real-time system. Using the analysis, a system designer can obtain a worst-case execution time guarantee for tagged systems. Unfortunately, tagging may significantly increase the worst-case execution time (WCET) bound. For example, our experiments showed that tagging can increase the WCET of a program by a factor of 3.5x. As a result, applying tagging on real-time systems requires a large increase in the time allocated to monitored tasks. If a real-time system cannot support this extra utilization, then monitoring cannot be applied to the system.

To address this limitation, we have developed a system that greatly reduces the amount of time that must be statically allocated to each task in the worst case (i.e., WCET) in order to enable tagging. Specifically, our system exploits dynamic slack in order to opportunistically perform as much tagging as possible within the time constraints of the system.

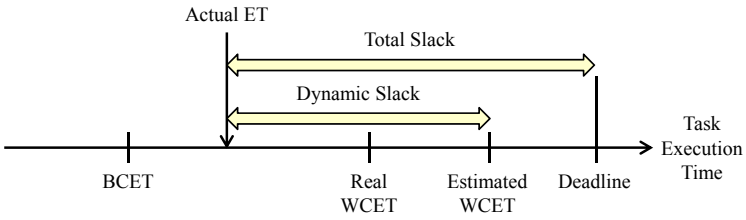


Figure 6: Dynamic slack and total slack

The approach is based on three key observations:

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

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

Here, we present a framework for opportunistically performing tagging in such a way as to ensure that the main task's execution time does not exceed its WCET bound. For each monitored event that triggers a tagging operation, our system checks whether there is enough slack to account for the worst-case impact of tagging on the main task's execution time. If there is enough slack, then the tagging operation proceeds. If there is not enough slack, then instead the event is dropped.

There are two main challenges in this approach. The first is how to measure slack at run time and decide when it is necessary to drop tagging operations. Once it is decided to drop a tagging event, the second issue is how to drop the tagging operation in such a way as to maintain the correctness of the tagging scheme without incurring false positives.

In the following discussions, we describe how our framework addresses these challenges using a multi-core system as an example, Here, one core, called *main core*, runs a main computation task and another core, called *monitoring core* performs tagging operations for monitoring. The tagging operation is also called *monitoring task*. The instructions on the main core that trigger a tagging operation for run-time program monitoring is referred to as *monitoring events* or *tagging events*.

3.6.1 Measuring Slack In order to decide when it is possible to perform a tagging operation, a system must be able to measure the dynamic slack available. Dynamic slack is defined as the difference between a task's expected worst-case execution time (WCET) and it's actual execution

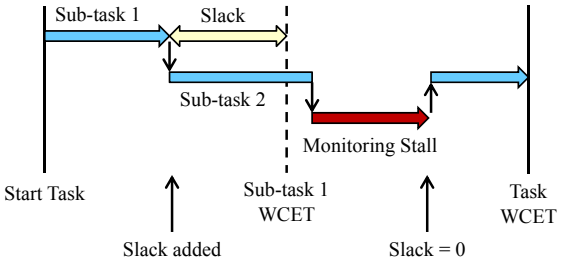


Figure 7: An example for dynamic slack changes

time [1]. This is only a portion of the total slack which is the difference between a task's finish time and its deadline (see Figure 6). Although the dynamic slack only accounts for a portion of the total slack, we only focus on dynamic slack because this is the portion of slack that is specific to a task. Additional slack in the schedule could be assigned to a specific task to be used for tagging by the system designer or scheduler. For brevity, we will use the term slack to refer to dynamic slack.

In order to perform tagging operations as the task runs, we need to be able to measure dynamic slack as the task runs. We can track dynamic slack by setting a number of checkpoints throughout the task. These checkpoints effectively divide the task into a number of sub-tasks. For each of these sub-tasks, the sub-task's WCET is determined. At run-time when a sub-task finishes, the difference between its actual run-time and its WCET is the slack generated by the sub-task. Specifically, we insert code to mark each sub-task boundary. At the beginning of a sub-task, the WCET of a sub-task is loaded into a timer. Each cycle, the timer decreases. At the end of a sub-task, the remaining value in the timer is the slack generated by the sub-task. This value is added to the current slack. An example of this process is shown in Figure 7. In our experiments, the division of a task into sub-tasks was done by hand but this process could be automated to divide a task using function boundaries, code length, or some other criteria. In addition to the slack accumulated while running, a portion of *headstart slack* can be initially assigned by the designer or scheduler to the task. For example, if the designer knows that static slack exists in the schedule, this slack can be added to the initial dynamic slack of a task to be used for tagging.

By accumulating this slack as the task runs, we can determine whether a tagging operation can be performed while still meeting the real-time constraints. If the worst-case impact of the tagging operation on the main task is less than the accrued slack, then the tagging operation can execute. If running the tagging operation causes the main task to stall, then slack is consumed. Slack was initially generated since the main task was running ahead of its WCET, so stalling up to the slack time will not cause the main task to exceed its WCET (see Figure 7). In the best case, the tagging operation executes entirely in parallel and does not affect the main task and thus consumes no slack. On the other hand, if the worst-case impact of the tagging operation on the main task's execution time is greater than the slack, then, conservatively, the tagging operation cannot be run. Instead, it must be dropped in order to guarantee that the main task finishes within its WCET.

3.6.2 Dropping Tagging Operations Dropping a tagging operation implies that some functionality of the tag-based monitor has been lost. This may cause either false negatives, where an

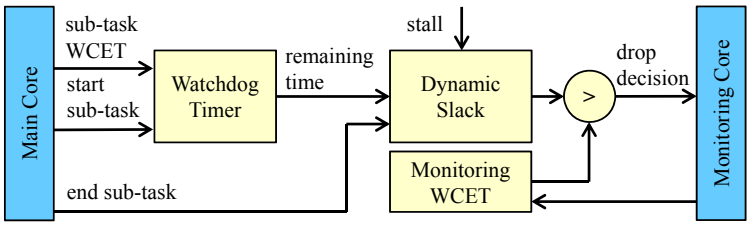


Figure 8: Hardware modules for slack tracking and drop decisions

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

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

3.6.3 Hardware Support Here, we present hardware mechanisms that eliminates the overhead of the proposed opportunistic tagging framework on the main task's WCET. There are two main sources that affect the main task's WCET when tagging operations are dropped purely in software: the additional code for keeping track of slack and the impact of the dropping task. By performing these two operations in hardware, we can ensure that the throughput of the monitored program is not affected by tagging even in the worst case.

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

The currently measured dynamic slack is used to determine whether the tagging operation can run

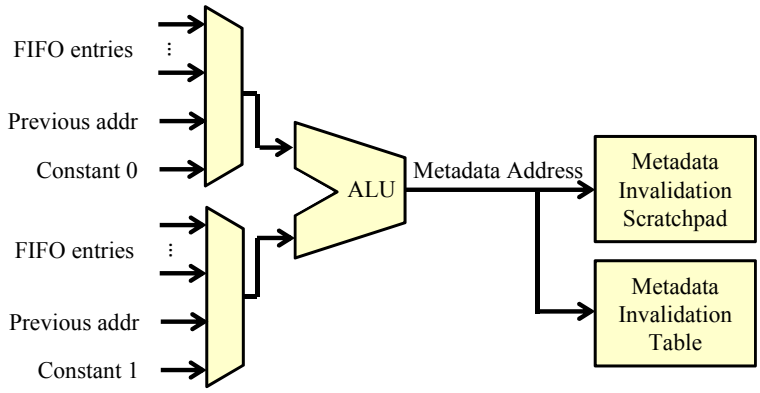


Figure 9: Metadata invalidation module (MIM)

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

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

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

In order for this dropping operation to match the throughput of the main core (i.e., up to one monitoring event per cycle), the metadata invalidation information is stored on-chip. The MIT is

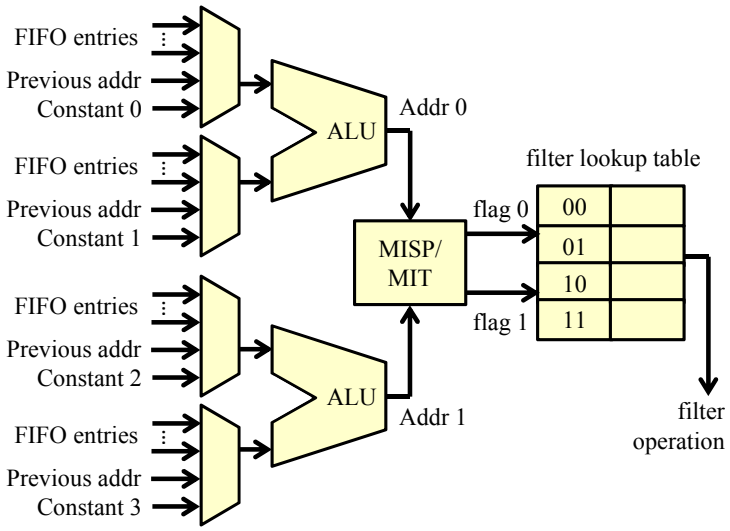


Figure 10: Metadata filtering module (MFM)

implemented as a small on-chip memory, similar to a cache. This memory stores invalidation flags and is indexed using part of the metadata address. It stores the remaining portion of the address as a tag. Unlike a cache, if an access misses in the MIT, there is no lower-level memory structure to go to. This is done in order to ensure that the MIM can handle a monitoring event every cycle. Instead of backing the MIT with lower-level memory, if writing to the MIT would force an eviction, we instead mark the corresponding cache set as "aliased". From this point on, we are forced to conservatively consider any metadata that would be mapped to this cache set as invalid, regardless of the tag corresponding to its metadata invalidation address. This reduces the amount of useful monitoring that can be done, but guarantees that the dropping hardware can match the main core's throughput.

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

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

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

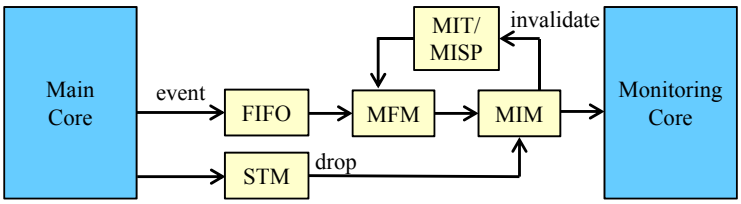


Figure 11: Block diagram of hardware architecture for opportunistic tagging on hard real-time systems

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

Full Hardware Architecture Figure 11 shows a block diagram of the complete architecture. Tagging events from the main core are first enqueued in a FIFO. The events in the FIFO are dequeued and processed by the metadata filtering module. The MFM checks the MIT and/or MISP to decide whether the event should be filtered because of invalid metadata tags. If the event is not filtered, then the metadata invalidation module decides whether to drop the event based on the slack tracking module. If the event is dropped or filtered, then the MIM marks invalidation flags using the MIT/MISP. Instead, if the event is not dropped or filtered, then it is forwarded to the monitoring core to perform the tagging operation.

4.0 RESULTS AND DISCUSSION

4.1 FlexCore Architecture

The proposed FlexCore architecture is developed in order to enable a wide range of run-time tagging schemes with low overhead on a single hardware platform. Here, we study the overhead of this flexibility by comparing the FlexCore architecture with dedicated Application-Specific Integrated Circuit (ASIC) implementations of each tagging scheme.

4.1.1 Methodology To estimate the area, power consumption, and operating frequency of the hardware modules for a typical ASIC flow, we used Synopsys Design Compiler (DC) with a 65nm IBM technology library. To estimate the same metrics for tagging schemes implemented on the FPGA fabric on FlexCore, we used Synplify Pro and Xilinx ISE. The tools were used to map each

Table 3: The area, power, and frequency of the FlexCore architecture

	Extension	Description	Max Freq (MHz)	Area		Power	
				μm^2	overhead	mW	overhead
Baseline	-	Unmodified Leon3 w/ 32KB L1	465	835,525	-	365	-
ASIC	UMC	Leon3 w/ UMC	463	932,118	11.6%	388	6.3%
	DIFT	Leon3 w/ DIFT	456	960,558	15%	388	6.3%
	BC	Leon3 w/ BC	456	996,894	19.3%	393	7.7%
	SEC	Leon3 w/ SEC	463	836,786	0.15%	364	-
FlexCore	Common	Leon3 w/ dedicated FlexCore modules	458	1,106,967	32.5%	418	14.6%
	UMC	UMC on Flex fabric (FPGA)	266	90,384	10.8%	21	5.8%
	DIFT	DIFT on Flex fabric (FPGA)	256	123,471	14.8%	23	6.3%
	BC	BC on Flex fabric (FPGA)	229	203,364	24.3%	27	7.4%
	SEC	SEC on Flex fabric (FPGA)	213	390,588	46.7%	36	9.9%

tagging scheme to the Virtex-5 FPGA, which was also manufactured in a 65nm technology. Virtex-5 was chosen to match the technology node that we use in the ASIC flow. Each tagging scheme on an FPGA was fully placed and routed, without dedicated components such as the core-fabric interfaces and caches. This synthesis provided the estimates for the operating frequency and the number of LUTs. To compute a rough estimate of the area, we used an estimate of CLB tile area from the model by Kuon and Rose [19]. The model reports that the area of a CLB tile with 10 6-input LUTs in the 65nm technology node is approximately 8,069μm^2. We used this estimate of 807μm^2 per LUT and multiplied it by the total number of LUTs used in our design to generate an area estimate. To compute an estimate of the power of the reconfigurable fabric, we used the Virtex-5 Power Spreadsheet [35] using the frequency and number of LUTs from FPGA synthesis. The area and power consumption of the shadow register file are obtained from a memory compiler and included in the estimates for the dedicated FlexCore modules. The power estimates currently use a fixed toggle rate of 0.1 and static probability of 0.5 for both ASIC and FPGA to provide rough comparisons.

To evaluate the performance overhead, we performed RTL simulations of the entire system including the processing core, caches, a tagging scheme, a memory controller, and off-chip SDRAM. The default configuration includes a Leon3 core with a single-issue 7-stage pipeline, 32-KB L1 instruction and data caches with 32-B lines, a 4-KB meta-data cache with 32-B lines, and a 64-entry core-to-fabric FIFO. The Leon3 caches use a write-through with no allocate policy. The simulations ran a set of benchmark programs from MiBench [15] and small kernels.

In the evaluation, we used four tagging schemes as examples: UMC, DIFT, BC, and SEC. UMC, DIFT, and BC were discussed earlier as examples. SEC stands for Soft Error Checking. In this scheme, the reconfigurable fabric is used to detect an error in ALU computations by using checksums or recomputing simple operations.

4.1.2 Area, Power, and Frequency Table 3 summarizes the estimated area, power consumption, and operating frequency for the Leon3 processor with and without various tagging schemes. The overheads in silicon area and power consumption are shown relative to the baseline Leon3. 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 tagging scheme 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

Table 4: The performance overhead comparisons between ASICs and FlexCore

Benchmark	UMC			DIFT			BC			SEC		
	(1X)	(0.5X)	(0.25X)	(1X)	(0.5X)	(0.25X)	(1X)	(0.5X)	(0.25X)	(1X)	(0.5X)	(0.25X)
sha	1.01	1.01	1.01	1.01	1.06	1.16	1.03	1.07	1.15	1.00	1.33	1.50
gmac	1.01	1.01	1.09	1.01	1.15	1.34	1.02	1.17	1.37	1.00	1.20	1.47
stringsearch	1.03	1.05	1.12	1.16	1.46	1.89	1.22	1.45	1.84	1.00	1.00	1.11
fft	1.01	1.01	1.01	1.02	1.05	1.31	1.02	1.03	1.35	1.00	1.15	1.45
basicmath	1.01	1.01	1.01	1.03	1.08	1.34	1.04	1.07	1.37	1.00	1.14	1.43
bitcount	1.04	1.06	1.07	1.08	1.36	1.69	1.13	1.27	1.64	1.00	1.19	1.48
geomean	1.02	1.02	1.05	1.05	1.18	1.43	1.07	1.17	1.44	1.00	1.16	1.40

tag 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 a tagging scheme in full ASIC implementations results in a slightly lower operating frequency because the tagging schemes tap into internal pipeline signals.

For the FlexCore implementations, the table separately shows the estimates for each tagging scheme on the Flex fabric and the estimates for the dedicated (ASIC) modules common for all FlexCore implementations including the FlexCore interface and 4-KB meta-data cache. Similar to the ASIC implementations, the synthesis results show that the addition of the FlexCore interface that taps into the main core pipeline slows down the frequency slightly by 1.5%. The dedicated FlexCore modules (the interface and the meta-data cache) add about 32.5% more silicon area and 14.6% more dynamic power compared to the baseline Leon3 processor. These overheads are higher than the ASIC implementations because the FlexCore interface is more general. In the FlexCore implementations, the Flex fabric adds noticeable overheads in addition to the meta-data tag cache and the interface. The tagging schemes require the Flex fabric to be 0.09 to $0.39mm^2$, which represent 11 to 47% additional area overheads, and consume 6 to 10% additional power.

While the relative area overheads are noticeable for the Leon3 processor, which is a tiny embedded processor, the results demonstrate that the FlexCore architecture is far more energy-efficient than running a tagging function on another processing core. Also, we note that the absolute area and the power consumption for the FlexCore modules are quite small if compared to higher-end microprocessors. For example, each processing core in UltraSPARC T2 occupies $12mm^2$ in the 65nm technology. MIPS R14000, which is fabricated in the 0.13μ technology, occupies $204mm^2$ and consumes 17W at 500MHz. For these modern processors, the relative overheads of FlexCore will be insignificant.

4.1.3 Performance Table 4 presents the normalized execution time for each tagging scheme. The execution time is normalized to the baseline Leon3 without any tagging. The ASIC implementations where a tagging scheme runs at the same clock frequency with the monitored core show at most 7% performance overheads. The Flex fabric runs at half the clock frequency as the main core for DIFT, UMC, and BC, and runs at one quarter of the main core frequency for SEC. The overheads come from two sources, stalls from a full forward FIFO and contention for the shared memory bus. A cache miss on a meta-data tag access forces the tagging scheme to stall while meta-data is refilled from memory. During this time the forward FIFO fills up with waiting instructions and may stall the main core if it becomes full. Also, meta-data refills from memory hog the memory bus shared by the meta-data cache and the main core caches. Therefore, the main core

suffers from increased memory access latencies for its own cache misses.

The performance overheads of the FlexCore implementations are estimated by running RTL simulations with two separate clocks for the main core and the FPGA fabric. The clock frequency for the FPGA fabric is set based on the frequency estimates from the synthesis results. BC, UMC, and DIFT run at half the frequency as the main core (0.5X in the table) while SEC runs slower (0.25X). For UMC, the performance of FlexCore is virtually identical to the ASIC performance despite running at half of the core frequency because only a small portion of instructions are forwarded to the reconfigurable fabric. BC and DIFT have a slight higher performance overheads of 18% for FlexCore because the fabric needs to process a larger percentage of main core instructions and access the memory for meta-data. However, the FlexCore overheads are still quite low. SEC has the highest performance overheads at 40% because it processes a large number of instructions at a low clock frequency.

The experimental results demonstrate that the FlexCore extensions are far more efficient than software implementations with comparable capability when the main core's clock frequency is moderate. For DIFT, previous software implementations have placed the performance overhead as high as 37 times [24]. Even a highly optimized implementation on high-end superscalar processors reported an average slowdown of 3.6 times [25]. For UMC, Purify [16] performs similar checks by adding state bits to each byte in memory, and was reported to be up to 5.5 times slower. The array bound check in software can also incur a noticeable slowdown in memory intensive programs up to 1.69 times even with extensive optimizations [12].

4.2 Harmoni Architecture

4.2.1 Methodology To evaluate the Harmoni architecture, we implemented a prototype system based on the Leon3 microprocessor [13] and compared the overhead with the FlexCore prototype. The prototype 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. Because 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. Similar to FlexCore, 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.

4.2.2 Area, Power, and Frequency 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.

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

Table 5: The area, power, and frequency of the Harmoni architecture with different maximum tag sizes

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%

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 [7] 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.

4.2.3 Performance 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 [15] 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 [9].

Figure 12 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 per-

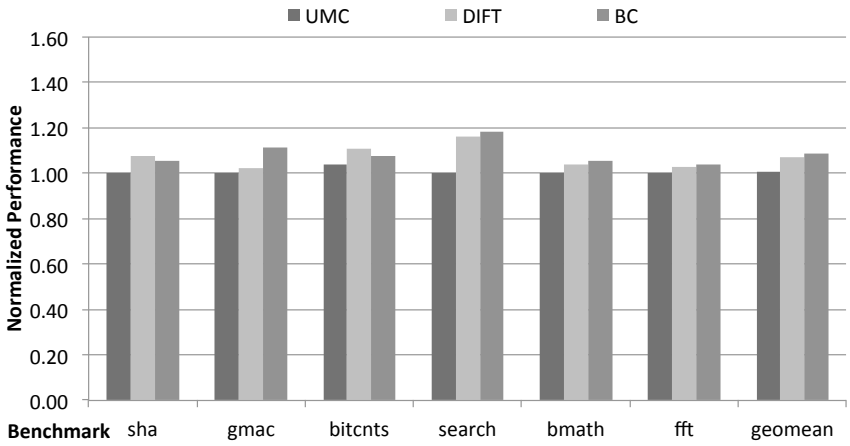


Figure 12: The performance overheads of the Harmoni co-processor

formance 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 5 is capable 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 [9] 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 12 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.

4.3 WCET Analysis for Real-Time Systems Tagging

4.3.1 Methodology To evaluate the effectiveness of our WCET estimation scheme, we compared its estimate with a simple WCET bound from sequential monitoring as well as simulation results using the SimpleScalar tool [3]. In addition to the WCET estimates with tagging, we also compared the results with the WCET of the main task without tagging, using both Chronos and simulations. Because our WCET infrastructure is based on processing cores, this study assumed that another core is used for tagging operations instead of the reconfigurable fabric in FlexCore or

Table 6: Estimated and observed WCET (clock cycles) with and without monitoring

Monitoring	Experiment	Benchmark						
		cnt	expint	fdct	fibcall	insertsort	matmult	ns
None	wcet-none	64531	3483	1805	245	598	133668	5951
	sim-none	62931	2293	1805	245	598	133668	5951
UMC	sequential-umc	103052	3591	4382	257	2489	357453	10338
	wcet-umc	64550	3498	3035	245	2083	256120	5953
	sim-umc	62931	2297	2564	245	1864	235120	5951
CFP	sequential-cfp	151732	11669	1976	794	1174	231507	18623
	wcet-cfp	93544	8984	1805	547	677	133668	13614
	sim-cfp	72540	5247	1805	382	598	133668	9824

the Harmoni accelerator.

For the experiments, we configured Chronos and SimpleScalar to model simple processing cores that execute one instruction per cycle for both the main core and the tagging engine. We assumed an 8-entry FIFO between the two cores. This configuration represents typical embedded microcontrollers, and is designed to focus on the impact of parallel run-time monitoring by removing complex features such as branch prediction and caches. In the evaluation, we used seven benchmarks from the Mälarden WCET benchmark suite [14] and two monitoring techniques: uninitialized memory checks (UMC) and control flow protection (CFP). UMC detects a software bug that reads memory without a write. CFP protects a program's control flow by checking a target address on each control transfer [2]. In this technique, a compiler determines a set of valid targets for each branch and jump in the main task. This information is stored on the monitoring core. On a branch or jump, the monitoring core ensures that the target is contained in the list of valid targets.

4.3.2 WCET Results Table 6 shows the experimental results for each benchmark under different configurations. The first set of rows show the WCET estimate from Chronos (`wcet-none`) and actual run-times from simulations (`sim-none`) without monitoring. The remaining rows show the WCET for the UMC and CFP tagging schemes. The results are shown for three different approaches: a bound from sequential monitoring (`sequential`), our approach (`wcet`), and simulations (`sim`). The numbers indicate the number of clock cycles.

If we compare the WCET estimates from ILP or MILP formulations with the worst-case simulation cycles for each monitoring setup, the results show that the analytical WCET estimates from our proposed scheme are larger than the observed WCET by 0% to 52% for UMC and 0% to 71% for CFP, depending on the main task. This difference is comparable to the case without parallel run-time tagging, where the analytical WCET from Chronos is larger than simulation results by 0% to 52%. In fact, for `expint`, the majority of the difference is from the WCET estimate of the main task rather than the effects of tagging. This result suggests that our WCET approach is not significantly more conservative than the baseline WCET tool for the main task.

Now let us compare the bound from sequential monitoring and the WCET from our proposed method. For UMC, our approach shows up to a 74% reduction in WCET estimates over the simple bound. Similarly, for CFP, our method shows up to a 73% improvement. These results demonstrate that modeling the FIFO decoupling between the main computation and tagging operations is important for obtaining tight WCET estimates of parallel tagging.

Table 7: Monitoring event type breakdown in percentage

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

Table 8: Percentage of checks that are not dropped or filtered

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

4.4 Effectiveness of Opportunistic Tagging on Real-Time Systems

4.4.1 Methodology We implemented our opportunistic tag-based monitoring architecture by modifying the ARM version of the gem5 simulator [5]. In order to explore the generality of the architecture for different monitors, we implemented the three different monitors: uninitialized memory check (UMC), return address check (RAC), and dynamic information flow tracking (DIFT). UMC and DIFT were discussed earlier as example monitoring schemes. Return address check (RAC) protects against certain security attacks, such as return-oriented programming [26], by checking that the address returned to after a function completes matches the address that originally called the function. We tested our system using several benchmarks from the Mälardalen WCET benchmark suite [14].

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

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

Table 9: Monitoring event breakdown for an FPGA-based monitor (FlexCore)

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

Table 10: Percentage of checks that are not dropped or filtered for an FPGA-based monitor (FlexCore)

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

4.4.2 Amount of Monitoring Performed Table 7 shows the amount of monitoring events whose tagging operations were performed, dropped, and filtered as a percentage of the total number of monitoring events. These results are shown for zero headstart slack. We find that a portion of monitoring can still be done without exceeding the main task's WCET. This is due to the dynamic slack that is gained during run time. On average, UMC can perform 17% of its monitoring tasks. RAC can perform 66% of its monitoring and DIFT can perform 31% of its monitoring. No results are shown for `insertsort` and `nsichneu` for RAC because these benchmarks do not make any function calls.

As expected, a false positive never occurred in our experiments. However, false negatives can occur due to dropping tagging operations. Specifically, dropped or filtered check-type operations can result in false negatives. Table 8 shows the number of checks that are performed as a percentage of the total checks. This acts as a measure of the coverage achieved by the monitors. The coverage for UMC is 15% on average and the coverage for RAC is 66% on average. The average coverage for DIFT is 59% which is much higher than the percentage of monitoring tasks that are not dropped or filtered. This implies that only a portion of the tagging operations performed by DIFT actually affect the checks. For example, DIFT propagates taint metadata on every ALU, load, and store instruction. However, only instructions that eventually propagate metadata to an indirect jump instruction affect the coverage.

4.4.3 FlexCore Results In addition to the parallel tagging on multi-core systems, we also evaluated the effectiveness of opportunistic tagging on the FlexCore architecture, which uses on-chip FPGA fabric for tag-based monitoring. We model the FPGA-based monitor as being able to run at 250 MHz and handle up to one tagging event each cycle [9]. Note that this means that the FPGA-based monitor can process a tagging event in two cycles of the main core which runs at 500 MHz. Table 9 shows the number of monitored, dropped, and filtered events on FlexCore with no headstart slack. UMC and RAC are able to run 67% of their monitoring tasks on average, while

DIFT is able to run 72% of its monitoring tasks on average. Table 10 shows the coverage achieved by these monitoring schemes on FlexCore. RAC shows similar numbers to processor-based monitoring because the number of calls and returns in these benchmarks is relatively small. However, for UMC and DIFT, we see that FlexCore based allows much more monitoring to be done without adding additional headstart slack. The coverage for UMC increases from 15% to 62% while the coverage for DIFT increases from 59% to 86%.

5.0 CONCLUSIONS

This project aimed to realize the full potential of fine-grained tagging scheme for monitoring and detecting errors and attacks on software programs. In this context, the project developed flexible hardware tagged architecture designs that can perform a wide range of tagging techniques with low overhead. In order to enable the fine-grained tagging techniques for real-time systems, the project also developed both static analysis and dynamic enforcement techniques to provide a strong guarantee for the worst-case program execution time with tagging. The evaluation studies through simulations and FPGA prototypes suggest that the proposed tagged architecture can provide both flexibility and efficiency, and also be safely applied to hard real-time systems with strict deadlines.

6.0 REFERENCES

[1] Aravindh Anantaraman, Kiran Seth, Eric Rotenberg, and Frank Mueller. Enforcing Safety of Real-Time Schedules on Contemporary Processors using a Virtual Simple Architecture (VISA). In *Proceedings of the 25th International Real-Time Systems Symposium*, 2004.

[2] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Secure embedded processing through hardware-assisted run-time monitoring. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 178–183, Washington, DC, USA, 2005. IEEE Computer Society.

[3] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 2002.

[4] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve Version 5.5. `http://lpsolve.sourceforge.net/5.5/`.

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 Simulator. *SIGARCH Computer Architecture News*, 2011.

[6] James Clause, Ioannis Doudalis, Alessandro Orso, and Milos Prvulovic. Effective memory protection using dynamic tainting. In *Proceedings of the* 22^{nd} *International Conference on Automated Software Engineering*, 2007.

[7] Intel Coporation. Intel Atom Processor Z510, 2008.

[8] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the* 34^{th} *International Symposium on Computer Architecture*, June 2007.

[9] Daniel Y. Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G. Edward Suh. Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric. In *Proceedings of the 43rd International Symposium on Microarchitecture*, 2010.

[10] Daniel Y. Deng and G. Edward Suh. High-performance Parallel Accelerator for Flexible and Efficient Run-time Monitoring. In *Proceedings of the 42nd International Conference on Dependable Systems and Networks*, 2012.

[11] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve 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*, pages 103–114, 2008.

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

[13] Jiri Gaisler, Edvin Catovic, Marko Isomaki, Kristoffer Glembo, and Sandi Habinc. GRLIB IP Core User's Manual, 2008.

[14] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks – Past, Present and Future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

[15] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. *Workload Characterization, Annual IEEE International Workshop*, 0:3–14, 2001.

[16] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors in C and C++ programs. In *Proceedings of the Winter 1992 USENIX Conference*, page 125138, 1992.

[17] Jos Joao, Onur Mutlu, and Yale Patt. Flexible reference-counting-based hardware acceleration for garbage collection. In *Proceedings of the* 36^{th} *International Symposium on Computer Architecture*, 2009.

[18] Hari Kannan. Ordering decoupled metadata accesses in multiprocessors. In *ACM/IEEE* 42^{nd} *International Symposium on Microarchitecture (MICRO-42)*, December 2009.

[19] Ian Kuon and Jonathan Rose. Area and delay trade-offs in the circuit and architecture design of FPGAs. In *Proceedings of the 2008 ACM/SIGDA* 16^{th} *International Symposium on Field Programmable Gate Arrays*, 2008.

[20] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudury. Chronos: A Timing Analyzer for Embedded Software. *Science of Computer Programming*, 2007.

[21] Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd Conference on Design Automation*, 1995.

[22] Albert Meixner, Michael E. Bauer, and Daniel Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Proceedings of the* 40^{th} *Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[23] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve 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*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[24] James Newsome and Dawn 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.

[25] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng 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.

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

[27] Howard Shrobe, Adnre DeHon, and Thomas Knight. Trust-management, intrusion-tolerance, accountability, and reconstitution architecture (TIARA), 2009.

[28] Gerard Sierksma. *Linear and Integer Programming*, pages 237–239. Marcel Dekker, Inc., 2002.

[29] G. Edward Suh, Jaewook Lee, David X. Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the* 11^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, October 2004.

[30] Mohit Tiwari, Banit Agrawal, Shashidhar Mysore, Jonathan Valamehr, and Timothy 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*, MICRO 41, pages 94–105, Washington, DC, USA, 2008. IEEE Computer Society.

[31] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *Proceedings of the* 14^{th} *International Symposium on High Performance Computer Architecture*, 2008.

[32] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 273–284, Washington, DC, USA, 2007. IEEE Computer Society.

[33] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems*, 2008.

[34] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian memory protection. In *Proceedings of the* 10^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, 2002.

[35] Xilinx. Virtex-5 Power Spreadsheet, 2010.

[36] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 225–240, Berkeley, CA, USA, 2008. USENIX Association.

[37] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iwatcher: Efficient architectural support for software debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS

ALU	Arithmetic Logic Unit
ASICs	Application-Specific Integrated Circuits
BC	Bound Checking
BFIFO	Back First Input First Output
CACK	Co-processor Acknowledgement
CALU	Check Arithmetic Logic Unit
CFG	Control Flow Graph
CFGR	Configuration Register
CLB	Configurable Logic Blocks
CNTL	Control
CONTBL	Control signals
COTS	Commercial Off The Shelf
CTBL	Check Table
DIFT	Dynamic Information Flow Tracking
FIFO	First Input First Output
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
ILP	Integer Linear Programming
LUT	Lookup Table
MFG	Monitoring Flow Graph
MFM	Metadata Filtering Module
MILP	Mixed-Integer Linear Programming
MIM	Metadata Invalidation Module
MISP	Metadata Invalidation Scratchpad
MIT	Metadata Invalidation Table
OBJTBL	Object Table
RAC	Return Address Check
RC	Reference Counting
ROB	Re-Order Buffer
SEC	Soft Error Checking
SQL	Structured Query Language
SRAM	Static Random Access Memory
STM	Slack Tracking Module
TIARA	Trust management, Intrusion-tolerance, Accountability, and Reconstitution Architecture
TLB	Translation Lookaside Buffer
TMEM	Tag Memory Hierarchy
TRF	Tag Register File
UALU	Update Arithmetic Logic Unit
UMC	Uninitialized Memory Checking
UTBL	Update Table
WCET	Worst Case Execution Time