

Microarchitectural Support for Program Code Integrity Monitoring in Application-specific Instruction Set Processors*

Yunsi Fei

Dept. of Electrical & Computer Engineering

Z. Jerry Shi

Dept. of Computer Science & Engineering

University of Connecticut, Storrs, CT 06269

E-mail: {yfei,zshi}@engr.uconn.edu

Abstract

Program code in a computer system can be altered either by malicious security attacks or by various faults in microprocessors. At the instruction level, all code modifications are manifested as bit flips. In this work, we present a generalized methodology for monitoring code integrity at run-time in application-specific instruction set processors (ASIPs), where both the instruction set architecture (ISA) and the underlying microarchitecture can be customized for a particular application domain. We embed monitoring microoperations in machine instructions, thus the processor is augmented with a hardware monitor automatically. The monitor observes the processor's execution trace of basic blocks at run-time, checks whether the execution trace aligns with the expected program behavior, and signals any mismatches. Since microoperations are at a lower software architecture level than processor instructions, the microarchitectural support for program code integrity monitoring is transparent to upper software levels and no recompilation or modification is needed for the program. Experimental results show that our microarchitectural support can detect program code integrity compromises with small area overhead and little performance degradation.

1 Introduction

Recent years have seen two trends driving reliability and security to become critical concerns for embedded processors. As technological trends continue to lead towards smaller and faster transistors with lower threshold voltages and tighter noise margins, the probability of transient faults, also known as *soft errors*, has increased dramatically [16]. As opposed to the permanent physical damages to processors (hard faults), the intermittent transient faults are caused by external events which only change stored values or signal transfers, and thus compromise program code integrity. The other trend in embedded systems has been the drastic increase of embedded software contents and the pervasiveness of networked connections. The vulnerability of systems to software attacks has thus been increased. Security has emerged as a new system design goal in addition to the traditional design constraints of performance and power consumption [12]. Typical security attacks include buffer overflow, fault injections, and data and software integrity attacks. In this paper, we focus on detection of program code compromises, no matter whether they are caused by soft errors or security attacks. At the instruction level, any code alteration is manifested as *bit flip*.

Recent decades have also seen the emerging of application-specific instruction-set processors (ASIPs) as an important design choice for embedded systems. ASIPs allow designers to customize the instruction set architecture (ISA)

for a specific application domain. They combine the flexibility of software with the energy-efficiency and high performance of dedicated hardware extensions [7]. Since ASIPs allow both the ISA and underlying microarchitecture to be tuned for specific applications, they provide a good platform for integrating code integrity monitoring mechanisms into the design process. In this work, we propose a microarchitectural support for program code integrity monitoring in ASIPs.

1.1 Paper Overview and Contributions

We will address the problem of program code integrity monitoring by ensuring that run-time program execution does not deviate from the expected behavior. A monitor should capture properties of the permissible behavior and compare it with the dynamic execution. When a mismatch is detected, the monitor throws an exception to trigger appropriate remedy mechanisms. We design a dedicated hardware architecture for this purpose.

Even though there are other hardware-assisted architectural mechanisms for security supports [6], their separate hardware modules are not directly coupled with microprocessors. In addition, they usually require compiler supports and result in considerable performance and hardware overheads [15]. In our design, both the ISA and underlying microarchitecture are customized for specific applications, and the microoperations¹ for monitoring can be incorporated in the design methodology as a design step by redefining the ISA. Since microoperations are at a lower level than machine instructions, the augmented microarchitecture is transparent to upper software layers, thus no compiler support is needed for the custom microarchitecture. The hardware monitor is seamlessly integrated with the microprocessor pipeline architecture. Hence both performance and area overheads would be small.

The remainder of the paper is organized as follows. We first give a survey of the relevant past work in Section 2. Section 3 explains the rationale behind the proposed techniques. Section 4 presents details of the monitoring architecture. Section 5 describes a systematic ASIP methodology to design the monitoring embedded ISA and microarchitecture for any given application. Section 6 presents experimental results and Section 7 draws conclusions.

2 Related Work

Monitoring code integrity helps computer systems defend against malicious attacks and recover from soft errors. There has been a lot of previous work on these two problems. However, most work targets individual problem only.

To prevent security attacks that execute malicious code, checkpoints can be placed at one or multiple layers in a system. However, if checking is done too early, attacks taking place after it will not be detected. For example, the OS may

*Acknowledgments: This work was supported by an NSF grant CCF-0541142.

¹Microoperations are elementary operations performed on data stored in datapath registers.

check the integrity and authenticity of a program before loading it into memory. The code, however, can be modified in memory by attackers after the checkpoint.

Several hardware approaches have been proposed to protect the code when it is stored in memory. XOM encrypts code and allows the instructions only to be executed but not otherwise modified [8]. AEGIS encrypts both code and data stored in off-chip memory and uses hash functions to check the integrity of code and data in cache [18]. Both XOM and AEGIS focus on the memory system and assume the processor itself is secure. The instructions, however, may be changed when being transferred into the processor or when stored in the instruction window. In addition, encryption and decryption of instructions at run-time require very sophisticated cryptographic engines and often degrade the system performance. Zhang et al. proposed a separate secure co-processor for monitoring critical kernel data structures [20]. The secure co-processor, however, is too expensive to be used in low-end computing devices.

Typical approaches to counter soft errors rely on redundant resources. An operation is performed in several identical circuits simultaneously or in one circuit at different times. Any discrepancies among the results indicate soft errors. For example, TRUSS have two identical processors [17] and the states of the two processors are compared periodically. The Boeing 777 aircraft has three processors and data buses [19]. Redundancy of hardware is normally expensive, especially for embedded system design where both the size and cost are critical. SWIFT addresses the problem with software approaches [13]. An operation is performed twice with two copies of code on different registers with identical values. This method cannot detect multiple-bit faults and assumes that a processor has sufficient resources (registers and functional units) to execute redundant codes without significant performance degradation.

Recently, Arora et al. proposed a run-time monitoring mechanism implemented with hardware [6]. In addition to the integrity of instruction streams, they monitor inter-procedural and intra-procedural control flow as well. Their monitor is separated from the pipeline, introducing long latencies, and thus has a large performance overhead. Our monitor is integrated into pipeline stages seamlessly, and the monitoring operations are hidden by critical paths of the processor pipeline. Therefore our method does not slow down the processor's cycle time.

Ragel et al. proposed IMPRES to monitor processor reliability and security, in which a special register stores the expected checksum of a basic block and the value is compared with the checksum generated at run-time [10]. Their method relies on embedding extra instructions in the application code to set checksums in the special register and thus requires re-compilation and binary instrumentation which result in significant code size increase and performance degradation. Our work is much more light-weighted and has a substantial advantage over Ragel's. It requires a minimum support from the OS, and legacy code can run without modifications or recompilations.

3 Design Rationale

Since hash values are a good indicator of program behaviors, we monitor program code integrity by comparing two hash values of the instruction streams. One hash value is generated before the program starts and can be considered as the expected behavior of the program. The other hash is generated by the processor at run-time after instructions have been fetched into processor. Although the design idea is drawn from standard code integrity monitoring approaches, many issues remain to be investigated to make code checking efficient and effective. Next we discuss several salient design issues.

3.1 Granularity Level of Code Monitoring

An appropriate granularity level to characterize the program's properties will affect the design complexity and effectiveness greatly. There are several considerations as listed below in deciding the number of hash values we need to compute and the range of instructions each hash value monitors.

- 1 The expected hash should be computed statically before the execution and will match the dynamic hash if the program is not compromised.
- 2 A behavior violation can be detected promptly. Ideally, the compromised code should be stopped before any damages are inflicted.
- 3 The hardware and performance overhead involved in run-time checking should be reasonable.

Considering the above requirements, we select to monitor the program code at the basic block level. It is easy to detect the range of basic blocks with hardware. The dynamic hash value of execution can be computed at run-time and compared with the statically computed expected hash. Any changes to the code will be detected at the end of basic blocks, most of which have less than 100 instructions.

3.2 Location of Code Monitoring

Another issues is where the code monitoring mechanism is located. We would like to place it in late stages, e.g., as close to the decode stage as possible, to capture more potential code changes. For example, if instructions are checked in the instruction cache, those code alterations that occur when being transferred over the bus will not be caught. We decide to incorporate the monitoring mechanism into pipelines and perform the checking in the instruction fetch (IF) and decode (ID) stages. Any alterations made before instructions are fetched into processor pipeline will be detected. Although code changes may take place after the ID stage as well, in this paper we focus on capturing the changes before instructions enter the pipeline.

3.3 Managing Hash Values

In order to compute and compare hash values, the microarchitecture needs to be enhanced. An internal hash table (IHT) (or one or more special registers) is added to store expected hash values. At run-time, the enhanced hardware detects the beginning of a basic block and starts computing its hash value as instructions are being fetched. When program execution proceeds to the end of the basic block, the IHT is searched. If a hash table entry for the basic block is found and the dynamic hash matches the expected one, it is a *hash hit* and the basic block is intact. If the basic block is found in the hash table but the dynamic hash does not match the expected one (defined as *hash mismatch*), or the basic block is not found in the hash table at all (defined as *hash miss*), an exception is raised with different signals indicating the cause. The OS will take over the control and decide how to respond.

The expected hash values can be loaded into the IHT by applications, as seen in [10], or the OS. If applications load the hash table, compilers need to insert at proper locations of programs the instructions that load expected hash values. The hash table load instructions will increase code size dramatically, and this method also increases complexity of the compiler [10].

Alternatively, the IHT can be managed by the OS. The compiler still generates the expected hashes for each block. However, in this method, all the hash values are simply attached to the application code and data and will be loaded into a section of memory managed by the OS when the application starts. The hash values can even be computed after binary code is generated, e.g., by a special program or the OS

application loader. At the end of a basic block execution, an exception caused by a hash mismatch will signal the OS to terminate the application. On an exception caused by a hash miss, the full hash table (FHT) in memory will be searched instead, and some entries in the IHT will be replaced. If the basic block is not in the FHT either, or dynamic hash is different from the expected hash value, the OS will terminate the program. Note that the search of the FHT can be done either by hardware or by software, in a similar manner as a cache miss handler.

The OS managed scheme has several advantages over the application managed one. It does not increase the complexity of compilers very much, and does not change the code size at all. In the OS managed scheme, the load of expected hash values is determined by dynamic execution paths, which are not available at compile-time. Thus the OS managed scheme may achieve a better performance than the application managed scheme.

No matter how expected hash values are computed and loaded, the underlying microarchitecture for run-time monitoring is similar. The enhanced hardware computes the dynamic hash, compares it with entries in the IHT, and raises exceptions on a hash miss or mismatch. In this paper, we will focus on microarchitecture modifications for monitoring and assume an OS is in place to handle monitoring exceptions.

The IHT acts like a cache of expected hashes stored in a FHT, which is analogous to memory. Thus many techniques for improving cache performance can be adopted here. The IHT has a limited size and may not hold all the expected hashes for a program. A suitable table entry replacement mechanism should be invoked when the table is full. In our OS managed approach, specific hardwares are designed to implement the replacement policy and select appropriate entries to overwrite when the IHT is full.

3.4 Error Model

The errors that the monitor can detect are determined by the hash algorithms. Some sophisticated cryptographic hash functions, such as MD5, SHA-1 [14], etc., can detect many types of instruction changes as they produce large size hash values and the probability of two instruction streams having an identical hash is extremely small. However, cryptographic hash functions are computationally intensive and induce long latencies. Even with dedicated high-performance hardware, it is difficult to make them keep up with the speed of processor pipelines. We start with a widely adopted assumption of simple faults: considering only a single bit flip in a basic block of program code. We employ a simple checksum function, XOR, in our experiments. The detailed fault analysis will be presented in Section 6.

4 Microoperation-based Monitoring Architecture

This section describes how to incorporate the code integrity monitoring mechanism into the processor pipeline. We first introduce microoperations. We then provide an overview of the proposed architectural support for run-time program code integrity monitoring, and describe in detail the microoperation-based implementation.

4.1 Microoperation

Microoperations are primitive processor operations which are performed on data stored in datapath registers. They are at a more fundamental level than processor instructions [11]. Figure 1 shows the sequence of microoperations which needs to be executed in the IF stage in a PISA processor pipeline [4]. Here *CPC* denotes the program counter register. The current PC value is read from *CPC* and used to fetch an instruction from the instruction cache (*IMAU*).

The fetched instruction is then stored into a specific register *IReg* for later use. In the end, the *CPC* is incremented for the next instruction fetch.

```

current_pc = CPC.read();
instr = IMAU.read(current_pc);
null = IReg.write(instr);
null = CPC.inc();

```

Figure 1. Microoperations for the IF stage

4.2 Overview of the Monitoring Architecture

The microarchitecture enhancement for program code integrity checking can be specified by microoperations and is embedded into the pipeline stages. Since it is working at a level below the instructions, the monitoring mechanism can not be bypassed by software or compromised by malicious users, thus providing an effective detection measure.

Figure 2 depicts the conceptual block diagram of the proposed monitoring architecture. The original processor datapath is represented by a typical in-order five-stage pipeline. The pipeline stages interact with the instruction cache, data cache, and control logic. For the purpose of code monitoring, the processor datapath is extended with a Code Integrity Checker (*CIC*), where an internal hash table (IHT_{bb}) is set up to capture properties of the expected program behavior (EPB), a hash functional unit (*HASHFU*) to compute the properties of the program in execution, and a comparator (*COMP*) to detect deviation of program execution from the permissible behavior at run-time. *Exception* signals will be asserted when a hash miss or a mismatch is found. The control logic will notify the OS to respond with actions. Since the components in the *CIC* are distributed into different pipeline stages, they do not affect the number of execution cycles for any program running on the processor.

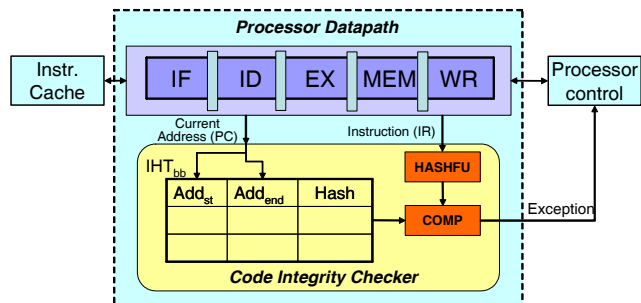


Figure 2. Block diagram of the proposed monitoring architecture

We have chosen to monitor the program behavior at the basic block level. Flow control instructions, such as branch and jump, indicate the end of a basic block, and the next instruction to be executed is the beginning of another basic block. In the internal hash table IHT_{bb} , each entry is a tuple associated with a basic block, $(Add_{st}, Add_{end}, Hash)$, where Add_{st} is the starting address of the basic block, Add_{end} the ending address, and $Hash$ the expected hash of instructions in the basic block, which is a good indicator of EPB. During program execution, the *HASHFU* computes the hash value of the instructions within a basic block until a flow control instruction is encountered. In our approach, both the hash table look-up and entry replacement are dynamically performed during execution.

4.3 Microoperation-based Basic-block Level Integrity Checking

Since the CIC is incorporated in the processor pipeline, the related microoperations are distributed into different stages. We next discuss monitoring microoperations in instruction fetch and decode stages.

4.3.1 Microoperation Extensions in the IF Stage for All Instructions

Figure 1 presents a sequence of original microoperations in the IF stage for all instructions. For monitoring, the stage will be augmented with dynamic hash computation and storage. A register (*STA*) is added in the microarchitecture to store the starting address of the basic block currently in execution and a register (*RHASH*) to store the computed hash value. Figure 3 (a) illustrates the flow of operations added in the IF stage. First, the register *STA* is checked, and a value of zero indicates that it is at the beginning of a new basic block and the current PC value needs to be loaded into *STA*. After an instruction (*instr*) is fetched from the instruction cache, it is fed into *HASHFU*, together with the old hash value from *RHASH*, to compute the accumulated hash value for the basic block. The updated hash is then stored back to *RHASH*. Figure 3 (b) shows the augmented microoperations in the IF stage. The lines in *italics* are the extra microoperations embedded for code monitoring. The microoperation “*null = [start==0]STA.write(current_pc)*” is a conditional operation: only when the condition in the bracket is asserted the operation of writing *current_pc* into *STA* is performed.

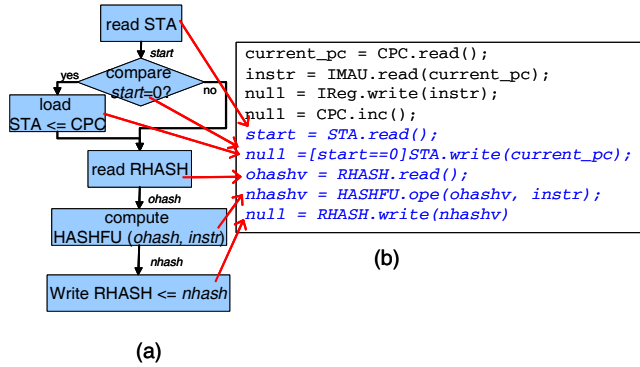


Figure 3. Flow diagram of augmented microoperations in the IF stage

Note that even though the IF stage for all instructions is extended with extra microoperations, they are simple and should not incur much latency. Normally the critical path of a processor datapath is not in the IF stage, so that we anticipate the augmentation in the IF does not affect program performance greatly.

4.3.2 Microoperation Extensions in the ID Stage for Branch/Jump Instructions

When a program execution encounters a flow control instruction, such as branch or jump, it reaches the end of the basic block that is being executed. At this point, the IHT is looked up for the basic block. Figure 4 shows the augmented microoperations in the ID stage for an “JR” instruction. Since the detection of the end of a basic block is in the ID stage of a control flow instruction (e.g., JR), it is one cycle later than when the computation of hash needs to be reset for a new basic block (i.e., in IF stage). When a JR instruction is in the ID

stage, current PC (CPC) refers to the address of next instruction following the JR instruction. Hence we need to obtain the JR instruction address from another register *PPC* (previous program counter). In our implementation, the range of a basic block is indicated by addresses stored in register *STA* and *PPC*. Computed hash value for the basic block is in the *RHASH* register. A tuple $\langle start, end, hashv \rangle$ is then used as the key to look up the table IHT_{bb} . If it is a hash hit, i.e., there is an entry with $\langle Add_{st}, Add_{end}, Hash \rangle$ equals to $\langle start, end, hashv \rangle$, the processor continues as usual and the monitor prepares to check next basic block by resetting *STA* and *RHASH*. If it is a hash miss (i.e., $found = 0$), an exception is raised to invoke the OS to search the FHT in memory. If it is a mismatch, i.e., an entry is found with the same address range but different hash value, another exception is raised to stop the program execution.

```

start = STA.read();
end = PPC.read();
hashv = RHASH.read();
<found,match> = IHTbb.lookup(<start,end,hashv>);
exception0 = [found==0] '1';
exception1 = [found==1 & match==0] '1';
null = STA.reset();
null = RHASH.reset();
target = GPR.read(rs);
null = CPC.write(target)

```

Figure 4. Augmented microoperations for the ID stage

The table IHT_{bb} can be implemented using a content-addressable-memory (CAM), or multiple registers. We will investigate how the performance and area are affected by the table size in Section 6.

5 Design Methodology

Figure 5 presents the ASIP design flow to incorporate a program code integrity checker. An automatic synthesis tool - ASIP Meister [1] is used. The tool captures target processors’ specification using a GUI, “Architecture design entry system,” and generates RTL processor descriptions for logic synthesis. Selection of target instructions for a particular application is beyond the scope of this paper [9]. After the target ISA has been specified, corresponding resources (such as general purpose register file, ALU, registers, etc.) are selected from a resource library. Meanwhile, extra hardware modules for monitoring (e.g., *RHASH*, *HASHFU*, etc., as described in Section 4) are selected as well. The monitoring microoperations are then embedded into proper instructions, such as branch, jump, etc. Synthesizable VHDL code for the custom ASIP are generated from the ASIP Meister HDL generator. The associated retargetable software toolset including a compiler, simulator, and assembler is also automatically generated for the customized processor.

6 Experimental Results

In this section, we present experimental results on evaluating the system overheads of the program code integrity checker and analyze the effectiveness of our approach in fault detection.

6.1 Performance Impact of the Checker

Since the size of the hash table is limited, very often not all the expected hashes of a program can fit in the IHT. We

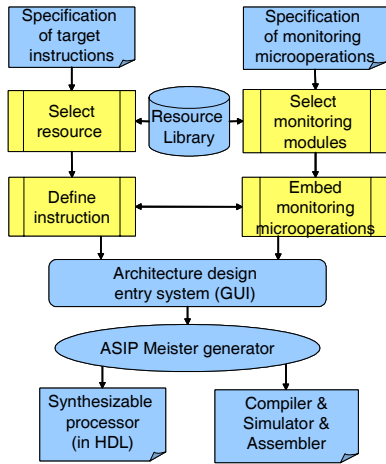


Figure 5. Design methodology for self-monitoring ASIPs

will have hash misses caused by capacity, which will incur FHT search and cause performance overheads. We will look into a suite of benchmark programs to see if a proper hash table size can be identified for most programs.

The number of basic blocks varies for different programs. For example, `stringsearch` has 25 basic blocks executed while `susan` has 93 basic blocks. Thus, it is hard to find a good IHT size for all applications. Another application-specific factor, the execution pattern of basic blocks, will also affect the performance greatly. If the execution of basic blocks in a program has very good temporal locality, i.e., an hash table entry referenced by a program at one point in time will be referenced again sometime in the near future, the hash misses rate would be low. However, the locality characteristic of programs also varies a lot.

We assume that the OS handles monitoring exceptions with a least-recently-used (LRU) replacement policy. On each hash miss, the OS replaces half of the entries with hash records from the FHT. Figure 6 lists the miss rate of nine applications in MiBench [2] for different hash table sizes (from 1 to 32 entries). For several applications, such as `dijkstra`, `patricia`, `blowfish`, and `bitcount`, a hash table of 8 entries can reduce the miss rate greatly. We see a significant reduction for all the applications when the entry size is 32, which, however, may result in considerable area overhead. Overall, hash table miss rate highly depends on the behavior of programs.

Since each hash miss will incur FHT searching and OS managed table replacement, the number of execution cycles will increase. We assume each OS exception handling takes 100 cycles. Table 1 gives the cycle number overhead for code monitoring. Column 2 reports the total number of execution cycles for the original baseline architecture, a single-issue 6-stage PISA processor. Column 3 and 4 presents the number for the enhanced architectures with a CIC of 8 entries and 16 entries, respectively. Column 5 and 6 are the cycle number overheads. The average clock cycle overhead over the nine applications is 14.7% for 8 entries and 7.7% for 16 entries. The `stringsearch` application has relatively higher overheads due to its poor temporal locality of basic block execution.

6.2 Area Overhead

With the customized processor containing monitoring routines generated, we use ModelSim to simulate the VHDL

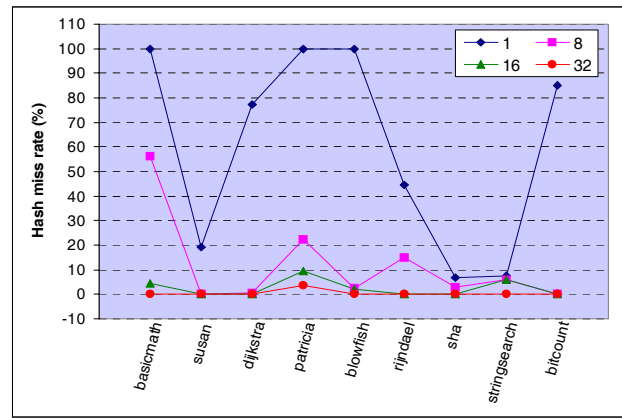


Figure 6. The IHT miss rate of different applications for different table size

Table 1. Cycle number overhead for program code integrity checking

Benchmarks	Clock cycle (10^6)			Overhead (%)	
	No CIC	CIC ₈	CIC ₁₆	CIC ₈	CIC ₁₆
basicmath	158	174.89	159.35	10.7	0.9
susan	25.58	25.63	25.58	0.2	0
dijkstra	54.79	57.6	54.81	5.1	0
patricia	133	146.64	138.81	10.2	4.4
blowfish	37.07	43.32	42.53	16.9	14.7
rijndael	37.6	45.4	37.6	20.7	0
sha	13.21	15.65	13.25	18.5	0.2
stringsearch	4.43	6.65	6.62	50.1	49.4
bitcount	43.62	43.62	43.62	0	0

code [3]. The functionality of applications is verified.

The custom processors were synthesized into technology-mapped gate-level netlists using Synopsys Design Compiler [5] with TSMC's 0.18 μ CMOS standard cell library. We have implemented both the baseline architecture and a number of enhanced processors consisting of the code integrity checker. The comparison results are presented in Table 2. Column 2 reports the minimum cycle time of the implementations, and column 3 the cycle time overhead compared with the baseline architecture. Column 4 reports the cell area of the implementations, and column 5 the area overheads. It shows that our method involves some area overhead, which is almost linearly dependent on the size of the table. The cycle time of the processor is not changed at all, because normally the critical path of a single-issue pipeline processor is in the execution stage and the extended microoperations are added in the IF and ID stages.

6.3 Fault Analysis

Our method compares the dynamic hashes generated during instruction execution with the expected hash values. Thus, only the errors on the executed instructions/basic blocks can be detected. Note that some errors can be detected by baseline microarchitecture itself, including invalid opcode, invalid opcode/operand combinations, etc.

The mechanism described in this paper intends to detect any changes to instructions before they are fetched and stored in instruction registers. However, some errors may not be de-

Table 2. Cycle time and area overheads for program code integrity checking

Designs	Minimum period(ns)	Cycle time overhead (%)	Cell area	Area overhead (%)
Baseline	37.90	-	2136594	-
With a 1-entry table (register)	37.93	0.1	2193510	2.7
With an 8-entry table	37.82	-0.2	2489737	16.5
With a 16-entry table	38.10	0.5	2750976	28.8

tected as it is always possible to find two instruction streams that have an identical hash. For a cryptographic hash algorithm like MD5 and SHA-1, the probability of an error not being detected is extremely small, e.g., 2^{-80} for SHA-1. Although we employed a simple XOR checksum algorithm, the probability of errors not being detected is still small. As long as the number of bits changed is odd, the XOR checksum can always detect the error. Hardware implementation of other more sophisticated cryptographic algorithms can be adopted in our monitoring architecture. The effectiveness of the XOR checksum may also be improved with a process-dependent random value.

7 Conclusions

In this paper, we have presented a microarchitectural support for monitoring the integrity of program code running on embedded processors. We choose the monitoring level of basic blocks and formulate a mechanism to check the instruction stream within each basic block at run-time. The monitor is incorporated into the processor pipeline seamlessly by augmenting the IF and ID stages of critical instructions with microoperations. The area overhead is reasonable for an IHT with 8 or 16 entries. The maximum frequency from synthesis report does not change at all. The number of execution cycles is slightly increased due to OS exception handling. Our studies reveal that the proposed architecture is capable of detecting a wide range of program code integrity compromises, no matter they are caused by security attacks or transient soft errors.

Future work will include refining the entry replacement policy for the IHT to make the methodology more effective, and experimenting with more secure yet efficient hash algorithms. Meanwhile, it will also be interesting to design the recovery mechanism either at the architectural or OS level.

8 Acknowledgments

The authors would like to thank Prof. Sri Parameswaran and Roshan Ragel of the University of New South Wales, Australia for their helpful discussions and suggestions on ASIP Meister usage.

References

- [1] ASIP Meister. [<http://www.eda-meister.org/asipmeister>].
- [2] MiBench. [<http://www.eecs.umich.edu/mibench/>].
- [3] ModelSim Simulator. [<http://www.model.com/>].
- [4] SimpleScalar Portable Instruction Set Architecture (PISA). [<http://www.simplescalar.com/>].
- [5] Synopsys Design Compiler. [<http://www.synopsys.com/>].
- [6] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha. Secure embedded processing through hardware-assisted run-time monitoring. In *Proc. Design Automation & Test Europe Conf.*, pages 278–283, Mar. 2005.
- [7] T. Gokler and H. Meyr. *Design of energy-efficient application-specific instruction set processors*. Kluwer Academic Publishers, Norwell, MA, 2004.
- [8] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proc. Int. Conf. on Architectural Support for Programming Languages & Operating Systems*, pages 168–177, Nov. 2003.
- [9] J. Peddersen, S. L. Shee, A. Janapsatya, and S. Parameswaran. Rapid embedded hardware/software system generation. In *Int. Conf. on VLSI Design*, Jan. 2005.
- [10] R. Ragel and S. Parameswaran. IMPRES: Integrated monitoring for processor reliability and security. In *Proc. Design Automation Conf.*, July 2006.
- [11] R. G. Ragel, S. Parameswaran, and S. M. Kia. Micro embedded monitoring for security in application specific instruction-set processors. In *Int. Conf. Compilers, Architecture, & Synthesis for Embedded Systems*, pages 304–314, Sept. 2005.
- [12] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady. Security in embedded systems: Design challenges. *ACM Trans. on Embedded Computing Systems: Special Issue on Embedded Systems and Security*, 3(3):461–491, Aug. 2004.
- [13] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proc. Int. Symp. on Code Generation & Optimization*, 2005.
- [14] B. Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley and Sons, 1996.
- [15] Z. Shao, Q. Zhuge, Y. He, and E. H.-M. Sha. Defending embedded systems against buffer overflow via hardware/software. In *Proc. Annual Computer Security Application Conf.*, pages 352–361, Dec. 2003.
- [16] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effects of technology trends on the soft error rate of combinational logic. In *Proc. Int. Conf. Dependable Systems & Networks*, pages 389–399, June 2002.
- [17] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzky. Fingerprinting: Bounding soft-error-detection latency and bandwidth. *IEEE Micro*, 24(6):22–29, Nov./Dec. 2004.
- [18] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proc. Int. Conf. on Supercomputing*, pages 160–171, June 2003.
- [19] Y. Yeh. Design considerations in Boeing 777 fly-by-wire computers. In *Proc. IEEE Int. High-Assurance Systems Engineering Symposium*, pages 64–72, Nov. 1998.
- [20] X. Zhang, L. Doorn, T. Jaeger, R. Perez, , and R. Sailer. Secure coprocessor-based intrusion detection. In *Proc. ACM SIGOPS European Workshop*, Sept. 2002.