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, any code alteration is 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 at runtime, checks whether it aligns with the expected program behavior, and signals any mismatches. Since the monitor works at a level below the instructions, the monitoring mechanism can not be bypassed by software or compromised by malicious users. We discuss the ability and limitation of such monitoring mechanism for detecting both soft errors and code injection attacks. We propose two different schemes for managing the monitor, the OS-managed and application-controlled, and design the constituent components within the monitoring architecture. Performance and area overhead of the two management schemes are evaluated and compared. Experimental results show that with an effective hash function implementation our microarchitectural support can detect program code integrity compromises at a high probability with small area overhead and little performance degradation.

Index Terms—Microarchitecture support for security and reliability, Microprocessors, ASIPs

I. INTRODUCTION

RECENT years have seen two trends driving reliability and security to become critical concerns for embedded processors. At the circuit level, as technologies scale down to the 10nm regime, transistors are getting smaller and faster. At the same time, because of lower threshold voltages and tighter noise margins, the probability of transient faults, also known as soft errors, has increased dramatically [1]. Different from permanent physical damages in circuits (hard faults), soft errors are intermittent transient faults. They can be triggered by external events like cosmic rays, and only change stored values or signal transfers temporarily. Therefore, they can escape testing and fault analysis easily. Soft errors may change program code executed in processors and cause malfunction of computer systems. At a higher system level, security has emerged as a new system design goal in addition to the traditional design metrics of performance and power consumption [2]. The vulnerabilities of systems to malicious software attacks has increased dramatically because of the proliferation of embedded software contents and pervasive networking connections. Malicious code injection through memory corruptions is a common security attack [3]. By executing the injected code, attackers can obtain the control of a system and access to sensitive information. There exist various countermeasures for such attacks, both implemented in software and hardware. However, many of them are either expensive for embedded applications, or address a specific attack and hence demonstrate to be possibly worked around once skilled attackers exploit new vulnerabilities of the system. At the instruction level, any code alteration is manifested as bit flips. Monitoring program code integrity at run-time can detect any changes made to the program code before the check point. Hence, a single light-weight code integrity monitor within processor can serve for both reliability and security purposes, and allow the system to take proper actions to defend against attacks or recover from errors.

As application-specific instruction set processors (ASIPs) have emerged as an important design choice for embedded systems [4], where both the instruction set architecture (ISA) and the underlying microarchitecture can be tailored for specific applications, they have provided an ideal experimental platform for enhancing reliability and security in architectures. Code integrity monitoring mechanisms can be incorporated into the design process as an integral step to strengthen the inherent error countermeasures of the processors. In addition, since the target applications are normally well understood at the design stage for ASIPs, useful application characteristics can be exploited for improving the performance of the monitoring architecture.

A. Paper Overview and Contributions

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

Even though there are other hardware-assisted architectural mechanisms for code integrity monitoring, their separate hardware modules are not directly coupled with microprocessors [5]. In addition, they usually require compiler support and result in considerable performance and hardware overheads [6]. In our design, both the ISA and underlying microarchitecture are enhanced for code monitoring, and the microoperations 1 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 making the monitoring mechanism more effective, flexible, and secure. As the hardware monitor is seamlessly integrated with the microprocessor pipeline architecture, both execution time and area overheads would be reduced compared to other techniques. Although some previous work presented similar schemes of in-processor code integrity checking [7], we improve the architecture and reduce the execution time overhead significantly. With our proposed monitoring architecture, we further discuss the design of each constituent component and the corresponding hardware management schemes. We propose two different approaches for handling the on-chip monitoring architecture, and compare them in terms of execution time overhead and design complexity.

The remainder of the paper is organized as follows. We first give a survey of the relevant previous work in Section II. Section III explains the design rationale behind the proposed techniques and presents the monitor architecture. Section IV discusses two management schemes for the on-chip code integrity monitor for reducing execution time overhead. Section V describes a systematic ASIP methodology to design the monitor-embedded microarchitecture and extend the ISA based on microoperations for any given application. Section VI presents experimental results and Section VII draws conclusions.

II. RELATED WORK

Monitoring program code integrity properly helps computer systems defend against malicious code injection 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.

A lot of security attacks result in injecting malicious code during the execution of an application program [3], [8]. For countermeasures, there exist both preventive mechanisms for these attacks, which prevent a vulnerability from existing, and detecting mechanisms, which prevent a vulnerability from being exploited. Software preventive approaches include employing safe programming dialects and transformations, like

---

1 Microoperations are elementary operations performed on data stored in datapath registers.
loading in IMPRES, Nakka et al. introduced a framework for providing security and reliability support in processors [37]. However, only architectural simulations are used to evaluate the proposed solution.

To counter soft errors, typical approaches rely on redundant resources of both software and hardware. Error correction coding (ECC) is one of the solutions normally used in memory hierarchy for soft error detection and correction. An operation can be 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 [38] and the states of the two processors are compared periodically. The Boeing 777 aircraft has three processors and data buses [39]. 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 [40]. 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.

In this paper, we will research and develop a practical run-time code integrity monitor which can be integrated into processor pipeline stages seamlessly. Hence, the monitoring mechanism can not be bypassed, and the execution time overhead incurred is reduced greatly. The monitor is useful for both security and reliability purpose because it detects the symptom of program code flips from any causes.

III. DESIGN RATIONAL AND CODE INTEGRITY MONITORING ARCHITECTURE

In this section, we describe our general design considerations for the monitoring mechanism and present our monitor architecture.

Since hash values are a good compact indicator of program behaviors, we monitor program code integrity by comparing two hash values of the instruction streams. As shown in Fig. 1, 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.

A. Granularity Level of Code Monitoring

An appropriate granularity level to characterize the program’s properties will affect the design complexity and effectiveness greatly. If too coarse-grained, serious inflicts may have been incurred before the code compromise is caught. While if too fine-grained, prompt detection is achieved at the cost of great performance degradation and nonnegligible area/power overheads. 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 execution time 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 then 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. Since most basic blocks have less than 100 instructions (on average about 10 instructions), a behavior violation can be detected within a short time. It is possible that in large basic blocks, the location of code compromise is so far from the end of basic blocks that by the time of hash comparison, damages have already been inflicted on the system. We assume large basic blocks are typically rare in programs, and more discussion about the basic-block level limitation and further improvements is presented in Section V.

B. Location of Code Monitoring

Another issue is where the code monitoring mechanism should be located. Considering the advantages of in-processor monitoring, we would like to place it in late stages, e.g., as close to the execution 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. In our work, we 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. Thus, the monitoring mechanism is harder to bypass compared to other code injection attack detection methods that protect the program code in memory. Another advantage of on-chip monitoring is that since the monitor is closely coupled with the processor pipeline, there is minimum delay between error detection and remedy, compared with other approaches that utilize a co-processor for monitoring purpose, or the Trusted Platform Module (TPM) [41] implementation that enables a software and hardware cooperative way to verify authenticity of a program at loading time.

C. Storing and Managing Hash Values

In order to compute hashes at run-time and compare them with static hashes, the microarchitecture needs to be enhanced for hash computation, storage, and management. To reduce the
execution time overhead of retrieving static hash values from memory, in our work, an internal hash table (IHT) is added on-chip to store static hashes. The IHT might be updated by the full hash table (FHT) in secure memory, which is used to store the whole set of hashes for all the basic blocks in the program, or by specific hash loading instructions. At run-time, the enhanced hardware detects the beginning of a basic block and starts computing its dynamic hash value as instructions are being fetched. When program execution proceeds to the end of the basic block, the IHT is searched to find the corresponding static hash value to compare with the dynamic one. If the basic block hash is found in the IHT and the dynamic hash matches the expected one, it is a hash hit and we consider the basic block intact. If the expected hash found in the IHT does not match the dynamic one, i.e., a hash mismatch, a violation exception signal is raised for the OS or recovery code to decide how to respond, e.g., terminating the program. If the basic block hash is not in the IHT at all, i.e., a hash miss, another exception is raised. Different actions will be taken depending on how the IHT is managed. Since the on-chip IHT has a limited size due to the area constraint of a chip, it may just hold a subset of the expected hashes. Thus, how to manage and update the IHT is important for the performance of the monitoring mechanism.

The IHT can be updated either by the OS/hardware or by applications with specific loading instructions inserted. If the IHT is managed by the OS, the on-chip IHT acts as a cache for the FHT. The expected hashes generated by the compiler at compile time are simply attached to the application code and data and loaded into a section of secure read-only memory (FHT) managed by the OS when the application is loaded. The hashes can even be computed by a special program or the OS application loader at loading time. On a hash miss at the end of a basic block execution, the FHT will be searched either by special hardware or the OS. If the basic block finds a hit in the FHT, one or more hash values will be loaded into the IHT for later use. A proper replacement policy, e.g., the Least Recently Used (LRU), will select appropriate entries to overwrite. If there is a hash miss or mismatch in the FHT, the OS may terminate the program execution. The OS management scheme has several advantages. It is invisible to instructions, and thus does not change the program code size at all. Many traditional techniques for improving cache/memory performance can be adopted here. No special hardware is added to search the FHT. However, the off-chip memory accesses for searching the FHT on hash misses managed by the OS may introduce significant execution time overhead.

Alternatively, the static hashes can be loaded into the IHT by applications in a way similar to [7]. Compilers need to insert at proper locations in programs instructions that explicitly load static hashes to the IHT, encoded as immediate [7] or transferred from the data cache or memory (from FHT in our approaches). In this scheme, the hash loading instructions have to ensure to load the expected static hashes into the IHT before they are used. Hash miss in IHT now becomes a symptom of violations. The IHT loading instructions decide when in the program execution and which hash values to load into the on-chip table, and hence program temporal locality can be utilized to reduce the execution time overhead. However, the extra loading instructions will increase the program code size. Detailed comparison of these two IHT loading mechanisms is presented in section IV.

D. Error Model

The errors that the monitor can detect are determined by the hash algorithms. Some sophisticated cryptographic hash functions, such as MD5 and SHA-1 [42], can detect many types of instruction changes, because 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 introduce long latencies. Even with dedicated high-performance hardware, it is difficult to make them keep up with the speed of processor pipelines. In this paper, we categorize the error models into three levels: single-bit error that is most commonly assumed in soft error analysis [1]; multiple-bit soft errors; arbitrary multiple-bit errors that are caused by code injection attacks. Hash functions with different computation complexity are needed corresponding to these error models. In Section V, we will discuss the hash functions and their implementation details.

E. Overview of the Monitoring Architecture

Based on the discussion above, we present our monitor architecture. The microarchitecture enhancement for program code integrity checking is embedded into the pipeline stages. Fig. 2 depicts the conceptual block diagram of the proposed monitoring architecture. The baseline processor datapath is extended with a Code Integrity Checker (CIC), where an internal hash table (IHT) is set up to capture properties of the expected program behavior, a hash functional unit (HASHFU) to compute the hashes of the program in execution, and a comparator (COMP) to compare the static and dynamic hashes to detect deviation of program execution from the permissible behavior at run-time. Exception signals will be asserted when a hash mismatch or miss is found. The control logic will notify the OS to respond with actions. There is also an FHT in secure memory for updating the IHT. Since the components in the CIC are distributed into different pipeline stages, and can operate in parallel with original pipeline operations, they do not introduce extra execution cycles for programs running on the enhanced processor.

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), each entry is a tuple associated with a basic block, \( (Add_{st}, Hash) \), where \( Add_{st} \) is the starting address of the basic block, and \( Hash \) the expected static hash of instructions in the basic block. During program execution, the HASHFU computes the hash value of the instructions within a basic block until a flow control
instruction is encountered. The dynamic and static hashes are then compared to detect any code modification within this basic block. In our approach, both the hash table look-up and entry replacement are dynamically performed during execution.

IV. EFFICIENT ON-CHIP IHT MANAGEMENT SCHEMES

In this section, we discuss two different IHT management schemes in detail for efficient code integrity monitoring. Within the additional CIC architecture, the IHT table can be implemented using a content addressable-memory (CAM), or multiple registers. It is an on-chip storage element where a group of pre-computed hashes can be stored to speed up static hash access. We will investigate how the performance and area of the enhanced processor are affected by the table size and different IHT management schemes in this section.

A. OS-managed IHT Loading

The on-chip IHT table is organized as a cache for the FHT in memory. In the OS-managed scheme, the OS is involved in IHT loading, and the hash loading mechanism is transparent to applications.

At the end of a basic block execution, the IHT table is searched for the entry for the basic block. On a hash miss, the hardware would notify the OS with the address of the current basic block. The FHT in secure memory will then be searched, and some entries in the IHT will be replaced by the desired static hashes and several other hashes stored nearby in memory. If the basic block hash is not in the FHT either, or dynamic hash is different from the expected hash value, the OS will be informed and start error handling process. 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. Here we assume it is OS-managed in order to avoid the hardware overhead.

In our OS-managed approach, the replacement policy is implemented to select appropriate IHT entries for overwriting on a hash miss when the IHT is full. Instead of loading hashes on each IHT miss, another way is to leverage the existing instruction cache miss handler. When an instruction cache miss occurs, the hardware loads hashes for the basic blocks that contain the cached-in instructions simultaneously with the instructions. This method can hide the IHT hash misses handling by the instruction cache miss handling. However, it has several drawbacks. First, it requires complex hardware extensions in both instruction cache and FHT architecture which can find the corresponding hash entries in FHT for the instructions brought to cache. The hash entry locating process could be time-consuming and increase the cache miss delay. Second, this joint miss handling method could result in unnecessary IHT updates, e.g., when two consecutive cache misses are caused by instructions in the same basic block, the hash of this block would be loaded twice into the IHT, while only the first time is necessary. This causes extra power consumptions. In this paper, we handle hash misses separately from instruction cache misses.

The OS-managed scheme has several advantages compared to the early application-controlled scheme described in [7]. It is transparent to the programs in execution. With static hashes generated at compiling time or loading time, legacy code can run on the enhanced architecture without any modifications. Moreover, with this hash management scheme, the application program has no control of the hash replacement or access to the static hashes loaded. It prevents any code modification from interfering with the monitoring process. If an application makes system calls or invoke new code, either dynamically linked library or a new application, the monitor checks the integrity of the code and corresponding hash values. If these requests are from injected code, the injection is detected before OS processes the requests. Proper cache-like entry replacement policy can be used to utilize the spatial locality of program hash values to reduce the execution time overhead. However, the application execution characteristics are not exploited in this scheme, thus the temporary locality of hash values is not utilized to further reduce the loading overhead. The execution time overhead can be reduced further by our advanced application-controlled IHT loading mechanism.

B. Advanced Application-controlled IHT Loading

The on-chip monitoring IHT can also be managed by applications, with special hash loading instructions being inserted into the program by compilers. Although there is a considerable performance degradation caused by hash loading in the previous approach presented in [7], e.g., 18% for application 'apropem', we found that the inefficiency does not come from the application-controlled mechanism. At compile-time, there is plenty of application-specific information that may help achieve higher efficiency. The previous application-controlled approach, however, does not utilize this kind of information. It simply treats a program as a series of basic blocks with instruction streams in, and monitors the integrity of each basic block separately. The correlation between basic blocks in a program is not exposed at compile-time for later dynamic monitoring. In this section, we propose an approach to exploit the temporal locality of program basic blocks for better performance of the monitoring mechanism. We define a hash loading instruction that specifies explicitly which hash value in the FHT is loaded into which entry in the IHT. We then describe an effective algorithm to locate the appropriate locations for inserting hash loading instructions and find the related basic blocks that each loading covers. A bunch of hash
values are pre-loaded to the on-chip IHT for monitoring the execution in a time window. Before the program execution reaches the end of a basic block, the corresponding hash should have been placed in the IHT for integrity checking. This is very challenging. With sophisticated application-controlled IHT management, assisted by compiler and the corresponding microarchitectural support, we can monitor the program code integrity with much less performance degradation.

1) Optimal IHT Loading Policy: Previous research work has shown that loading the hash value in each basic block introduces a considerable time overhead to application execution [7]. The advanced approach differs from the previous work by exploiting application-specific information to improve the performance. The IHT is multi-entry so that we are able to load multiple hash values once into the IHT but to monitor several basic blocks that are going to execute consecutively, similar to the traditional cache prefetching techniques [43]. The IHT will be better utilized if the hash values for the basic blocks pre-loaded together are involved in frequently executed loops. Fig. 3 illustrates a simple control flow graph (CFG) for an example program, where several basic block nodes, \( bb_1 \), \( bb_2 \), and \( bb_3 \), form a loop. The edge between nodes indicates control flow, e.g., at the end of \( bb_1 \), the program control flow is transferred to \( bb_2 \). Each edge is labeled with its execution frequency obtained from profiling. For each node, the total weight of the incoming edges is equal to that of the outgoing edges, and equal to its own execution frequency. Assume the IHT table size is 4, the hash values of the three basic blocks of the loop can be loaded only once before the program execution enters the loop (i.e., in \( bb_0 \)), and all the loop iterations can be monitored thereafter. We define the cost of IHT loading on a per-entry basis, e.g., when loading the hash values for the three basic blocks in the loop once, the loading cost is 3 time units. Note that the time unit is determined by the data cache access time (normally one cycle). However, in the previous checksum register scheme, every basic block execution has to load its hash at the beginning of basic block execution, and the total loading cost for the loop is \( 3(n+1) \) time units. It is intuitive to see that bursting pre-loading utilizing application-specific locality information can reduce the execution time overhead greatly.

![Fig. 3. A simple CFG with a loop](image)

The problem of finding when in program execution to load hash values and what hashes to load then boils down to find an optimal loading policy for the multi-entry IHT, so that the IHT loading overhead is reduced to minimum. A strict requirement is posed on the loading policy: it should guarantee that the hash value of a basic block is already in the IHT before the basic block finishes execution. Therefore, a hash miss indicates a behavior violation, and the OS is not involved in IHT loading.

2) Basic Block Clustering Algorithm: The program CFG is analyzed at a coarser granularity than the basic block level to minimize the loading overhead. We define a basic block cluster as a set of basic blocks that have their hash values loaded to the IHT at the same time in one batch. Only when the program execution switches from one cluster to another, the contents of the IHT are changed. This is similar to context switching in process management. The loading cost for each cluster switching is the product of the edge weight and the size of the destination cluster. Fig. 4 shows an example of basic block clustering, where the hash loading instructions are inserted on the control flow transfer edges among clusters. We will seek an algorithm for clustering the basic blocks in a program to minimize the inter-cluster switching overhead, with the constraint of maximum cluster size being determined by the size of the IHT.

![Fig. 4. An example of basic block clustering](image)

The problem is formulated as follows. The CFG of a program, \( G(V, E, W) \), is obtained from profiling, where \( V \) represents the set of basic block nodes, \( v_1, v_2, \ldots, v_n \), \( E \) is the set of edges \( e_{i,j} \), with \( i \) referring to the source node index and \( j \) the destination node index, and \( W \) denotes the set of edge weights, \( w_{i,j} \). The graph is partitioned to many clusters \( c_1, c_2, \ldots, c_m \), where the number of clusters \( m \) is not fixed, and the number of nodes in each cluster does not exceed the size of the IHT, \( K \), so that the total inter-cluster switching cost is minimum.

A graph partitioning problem as described above is known to be NP-complete [44]. We develop an iterative improvement-based heuristic algorithm for this problem. Starting from an initial solution of clustering, we improve the solution through iterations of exchanging nodes among clusters in search of a better clustering result - with lower loading cost. To prevent our solution from being stuck at a local minimum, we introduce some randomization in each iteration so that new solutions can be generated from local minimum solutions and the algorithm can escape to explore more design space. We next describe the algorithm in detail. Algorithm 1 below shows the pseudo code. The input is the CFG of the program and the total number of entries in the IHT (\( K \)). We also set other two constraints for the iterative-improvement algorithm, the maximum iterations allowed (\( R \)) and the upper limit for cost increase after one iteration (\( M \)). We first construct a
simple initial clustering solution. As edges in a CFG represent
color flow transfers, and hence indicate possible execution
paths, it is intuitive to place several consecutive basic blocks
into one cluster and load their hash values at one time. Thus,
instead of randomly constructing a clustering solution
to start with, we employ a depth-first-search (DFS) method
to generate the initial solution (Line 2 in Algorithm 1). All
the nodes in a CFG are sorted in a DFS order, and nodes
that are connected are candidates for elements in multi-node
clusters. The outermost loop of the algorithm (Lines 4-36 in
Algorithm 1) is for controlling the maximum number of nodes
allowed in a cluster, num. The num starts from 2, and scales
up with a step of 1 for each round until it reaches the physical
constraint, the number of entries in the IHT (K). Through this
fine-grained control, the design space of clusterings can be
more thoroughly explored for the best solution. Within each
iteration of the outer loop, an inner loop performs cluster
ejection and merging to seek better solutions with the
constraint of maximum cluster size num (Lines 6-34). For any
clustering solution in the algorithm, two lists are maintained:
a single node list and a multi-node cluster list (Line 8). Each
improvement iteration contains two phases. At phase 1 (Lines
9-18), each multi-node cluster ejects one node which results in
the least cost increase. At phase 2 (Lines 19-28), each single
node is going to search all its connected clusters for the one
that, when merged with the node, decreases the cost most.
The splitting and merging processes generate a new solution.
To reduce the possibility of the solution searching staying in
a local minimum (e.g., at phase 1, a node is ejected from a
cluster, and at phase 2, the cluster is selected to merge with
t hat node again), we process the nodes in the single node list
in a random order at phase 2. Hence, when a node is ejected
from a cluster, there is a good chance that another node would
select to merge to the cluster in phase 2 before this node is
considered. In this way, the new solution is able to escape the
local minimum. Fig. 5 demonstrates how clusters eject and
merge nodes to explore new solutions to reduce cost within
the iteration steps. Within an inner loop, the consequence of
cluster ejecting and merging nodes is node exchanging
between clusters to reduce the total cost, as shown in the
upper dotted box of Fig. 5. When the outer loop increases the
maximum allowed size of the clusters (from the upper box to
the lower box in Fig. 5), the new round of node ejecting and
merging of clusters causes some clusters to expand or shrink,
aiming at reducing the total cost. As the iterations proceed
and the maximum allowed cluster size reaches the physical
constraint, the best solution is generated with the cluster size
ranging from 1 to the IHT table size.

The cluster switching costs for moves employed in phases 1
and 2 are evaluated separately, as illustrated in Fig. 6. The top
direction (from left graph (a) to right graph (b)) demonstrates
the case when a cluster i of size s is ejecting a node j. The
bottom direction (from (b) to (a)) is for the case of node j
being merged to cluster i'. \(\Delta cost_e\) and \(\Delta cost_m\) are defined
as the cost increase for an ejecting move and a merging move,
respectively. In graph (a), the incoming edges of cluster i can
be classified into two categories: those with their destination
as node j and those that point to other nodes in the cluster.

Algorithm 1 CFGClustering

**Input:** CFG - \(G(V, E, W)\), total number of entries in the IHT - \(K\), iteration stop cost criterion - \(M\), maximum iterations allowed - \(R\)

**Output:** optimal clustered CFG with the minimum cost

1. set the initial maximum cluster size num = 2;
2. Perform depth-first-search sorting, DFS(G, num), get the initial clustering solution \(S_0\);
3. \(S_{\text{new}} = S_0\);
4. while num < K do
5. \(\text{counter} = 0\); cost\(_{\text{pass}}\) = 0;
6. while counter < R and cost\(_{\text{pass}}\) ≤ M do
7. \(S = S_{\text{new}}\); cost\(_{\text{pass}}\) = 0;
8. get two lists for \(S\): single\_node\_list and cluster\_list;
9. for each cluster \(clr_i\) in the cluster\_list do
10. \(n_{\text{select}} = \text{null}\), \(\Delta cost_e(n_{\text{select}}, \text{null}, \text{clr}_i) = \infty\);
11. for each node \(n_j\) in \(\text{clr}_i\) do
12. if \(\Delta cost_e(n_j, \text{clr}_i) < \Delta cost_e(n_{\text{select}}, \text{clr}_i)\) then
13. \(n_{\text{select}} = n_j\); end if
14. end for
15. end for
16. eject \(n_{\text{select}}\) from \(\text{clr}_i\) to single\_node\_list, update the cluster\_list;
17. cost\(_{\text{pass}}\) = cost\(_{\text{pass}}\) + \(\Delta cost_e(n_{\text{select}}), \text{clr}_i)\);
18. end for
19. for each node \(n_j\) in single\_node\_list do
20. \(\text{clr}_\text{select} = \text{null}\), \(\Delta cost_m(n_j, \text{null}) = \infty\);
21. for each cluster \(\text{clr}_i\) that \(n_j\) is connected to do
22. if \(\Delta cost_m(n_j, \text{clr}_i) < \Delta cost_m(n_j, \text{clr}_\text{select})\) and size\(_{\text{opt}}(\text{clr}_i) < \text{num}\) then
23. \(\text{clr}_\text{select} = \text{clr}_i\); end if
24. end for
25. end for
26. merge \(n_j\) and \(\text{clr}_\text{select}\), and update the cluster\_list and single\_node\_list;
27. cost\(_{\text{pass}}\) = cost\(_{\text{pass}}\) + \(\Delta cost_m(n_j, \text{clr}_\text{select})\);
28. end for
29. a new clustering solution \(S_{\text{new}}\) is generated;
30. if cost\(_{\text{pass}}\) > \(M\) then
31. \(S_{\text{new}} = S\);
32. end if
33. counter++;
34. end while
35. num++;
36. end while
37. \(S_{\text{final}} = S_{\text{new}}\); Output the final clustering solution \(S_{\text{final}}\) and the total cost.

The total weights for these two categories of edges are \(p\) and
\(q\), respectively. Similarly, the outgoing edges of cluster \(i\) can
be put into two categories as well, with their total weights
as \(o\) and \(r\). Node \(j\) also has other internal incoming and
outgoing edges, with their weights as \(m\) and \(n\). Note that the
cluster switching cost is the product of the edge weight and the
destination cluster size. When node \(j\) is ejected, the internal
edges between \(j\) and other nodes in cluster \(i\) become inter-
cluster edges, and will introduce loading costs. In addition,
edges with weight \(q\) change their destinations (from cluster \(i\)
with size \(s\) to cluster \(i'\) with size \(s - 1\)). Edges with weight
\(p\) change their destinations from cluster \(i\) to node \(j\) (i.e., the
size changes from \(s\) to \(1\)). These four edges, \(p, q, m, n\), cause
loading cost changes. Equation (1) gives the total cost increase
for ejecting node \(j\).
locality and updates IHT in a manner without application-specific information.

For the application-controlled approaches that use explicit hash loading instructions, there is a possible vulnerability. Since the program code can be modified by the attackers, they can easily change or insert hash loading instructions. In the previous approach [7], static hash value is encoded in the program code. Hence, it requires encryption and processor-specific key to protect the hash values, otherwise it is easy for the attackers to forge hashes to deceive the monitor. However, in our application-controlled scheme, the hash loading instruction only specifies the location of the expected hash in the FHT. The IHT table can only be updated with pre-computed hashes in FHT. As long as the FHT is secure, the hash value cannot be changed by the attacker. The consequence of an illegal hash update instruction is either the hash entry could not be found in the FHT, or loading an incorrect pre-computed hash value for a basic block to the IHT. Both of the cases can be captured by the OS or the monitor.

V. DESIGN METHODOLOGY

Within the CIC, an important component is the hash function logic. In this section, we will discuss the hash functions for the aforementioned three types of error models. We will then introduce a design methodology for the enhanced processor based on microoperation extension. We will describe how the extended microoperations for CIC are implemented into the processor pipeline.

A. Hash Function and Error Detection Analysis

Our method compares the dynamic hashes generated during program execution with the expected static hash values. Thus, only the errors on the executed instructions/basic blocks can be detected. Note that some errors can be captured by the 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 detected as it is always possible to find two instruction streams that have an identical hash. We can adopt different hash functions for different error models. For single-bit soft errors, a simple “XOR” hash function is effective to detect a large probability of random errors not being detected is only $2^{-32}$, considering an instruction width of 32 bits. For multiple-bit randomly distributed soft errors, a more sophisticated Cyclic Redundancy Check (CRC) coding that is widely used in network data transfer applications is suitable. Depending on the number of bit errors that the monitor targets to detect, different types of CRC coding can be considered and their error detecting capability is well studied [45]. For security attacks resulting in multiple-bit errors within a sequence of instructions, since the bit flips are not randomly distributed but injected intentionally by skilled attackers, it is more difficult...
to detect errors using checksum functions like CRC. For this reason, more sophisticated hash functions should be considered to prevent attackers from injecting malicious instructions that have the same hash values as intended execution. A cryptographic hash algorithm like SHA-1, SHA-2 \[46\] is more effective to prevent security attacks. However, they normally have large hardware overhead and long execution latency. Thus, they are more difficult to be coupled with pipeline stages. A trade-off between security enhanced by sophisticated hash algorithms and their computation complexity should be carefully balanced.

Note that although the code integrity monitoring mechanism we have proposed works well for soft error detection, it has several limits for defending against security attacks. Since the monitor checks code integrity at the end of each basic block, the monitoring and error detection may not be prompt for large-size basic blocks. In addition, attackers can modify the code to perform malicious execution and extend a basic block to delay the attack detection. A solution to this problem is to set a maximum basic block size to ensure prompt error detection. If applications do have larger basic blocks, compilers can break them into smaller ones by inserting unconditional jump instructions. Hence, at run-time a basic block that exceeds this size limit indicates code modification within the basic block. The performance degradation is negligible as large basic blocks are typically rare in programs. The enhancement is transparent to the two IHT management schemes that we have proposed in the paper and may cause very little hardware overhead within our monitor architecture. In general, an in-processor code integrity monitor can ensure that the program code is not altered within each basic block. However, it does not address control flow attacks. Without changing the program code, an attacker can still redirect the program execution within the loaded program by data memory corruptions, like return-to-libc attacks. In such cases, a control flow transfer validation mechanism, e.g., the technique presented in our previous work \[47\] and other’s \[48\], can work together with our code integrity checking mechanism to defend against more security attacks. In future work, we will extend our monitoring mechanism to target larger variety of security attacks.

In the following sub-section, we start with the widely adopted assumption of simple faults: considering only a single bit flip in a basic block of program code, which is the most common scenario for soft errors. We employ a hash function, “XOR”, to demonstrate the CIC implementation. We describe how the monitor microoperations are distributed into the pipeline stages for this case. We also discuss possible implementations for other more complex hash functions.

B. Implementation of Microoperation-based Basic Block Level Integrity Checking

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

1) Microoperation: We will first introduce microoperations. Microoperations are primitive processor operations which are performed on data stored in datapath registers. They are at a more fundamental level than processor instructions \[49\]. Fig. 7 shows the sequence of microoperations which needs to be executed in the IF stage in a PISA processor pipeline \[50\]. 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.

2) Microoperation Extensions in the IF Stage for All Instructions: Fig. 7 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. Fig. 8 (a) illustrates the flow of operations 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. Fig. 8 (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.

3) Microoperation Extensions in the ID Stage for Flow Control Instructions: When a program execution encounters a flow control instruction, such as branch or jump, it reaches the end of the basic block being executed. At this point, the IHT is looked up for the basic block. Dynamically computed hash value for the basic block is in the RHASH register. A tuple <start, hashv> is then used as the key to look up the table IHTlb. If it is a hash hit, 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 corresponding handling mechanism. For the OS-managed IHT scheme, the OS would search the FHT in memory for the current basic block. For application-controlled IHT scheme, this indicates the program execution deviates from the expected behavior and there may be some compromise in the program control flow. If it is a mismatch, i.e., an entry is found for the basic block but with different hash value, an exception is raised to notify the system to perform corresponding remedy. Fig. 9 shows the augmented microoperations in the ID stage for a JR instruction for both the OS-managed and application-controlled schemes.

$$\text{start} = \text{STA.read}();$$
$$\text{hashv} = \text{RHASH.read}();$$
$$<\text{found,match}> = \text{IHT.b.lookup}(<\text{start,hashv}>);$$
$$\text{exception0} = [\text{found}==0] '1';$$
$$\text{exception1} = [\text{found}==1 \&\& \text{match}==0] '1';$$
$$\text{null} = \text{STA.reset}();$$
$$\text{null} = \text{RHASH.reset}();$$
$$\text{target} = \text{GPR.read(rs)};$$
$$\text{null} = \text{CPC.write(target)}$$

Fig. 9. Augmented microoperations in the ID stage for flow control instructions

Note that even though the IF stage for all instructions is extended with extra microoperations, for “XOR” hash function, 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 if the complexity of the hash algorithm is reasonable.

Implementing hash functions such as CRC would cause the total latency of IF stage to exceed the clock cycle. In this case, the hash computation operations can be partitioned and distributed to the following pipeline stages, e.g. with $nhashv_{-hf} = \text{HASH.ope1}(\text{ohashv, instr})$ performed at IF stage and $nhashv = \text{HASH.ope2}(nhashv_{-hf}, \text{instr})$ performed at ID stage. The augmented operations for a JR instruction shown in Fig. 9 (hash comparison and exception generation operation) can be moved to EXE stage.

If cryptographic hash algorithms are implemented, the hash computation unit itself may have more stages than the processor pipeline. In this case, there is a latency between the end of basic block and hash computation and comparison. When a new instruction is fetched in IF stage, it is added into an input buffer of the hash unit. The hash computation starts when the input buffer is full, i.e., a block of message has been accumulated, or the current basic block ends. Even if the instructions in a basic block finish execution, the hash computation continues in the pipelined hash computation unit. The hash checking is performed when hash computation is finished. Since the hash computation unit is pipelined, it starts a new hash computation before the previous computation is done. This implementation does not cause pipeline stalls, but detects errors several cycles later than the end of basic blocks. To reduce the latency, a hash that can be done in small numbers of cycles is preferred. A good choice is AES-based hash [51] as an AES encryption can be performed in 10 cycles [52]. The delay between the instruction execution and error detection is only six cycles, which is acceptable. It is so far the most suitable cryptographic algorithm to be implemented in the CIC.

Note that although we describe the hardware improvements as micro-operation extension of instructions (in IF and ID stages), the same monitoring hardware can be realized as functional additions to the processor architecture.

In addition to the complexity of hash functions, another factor that affects the hash processing latency is the processor issue-width. In the above we have discussed a single-issue processor where the hash function logic in the IF stage handles one instruction at each clock cycle. For multi-issue processors like VLIW and Superscalar architecture, several instructions are fetched together and thus need to be processed simultaneously. To make the monitoring mechanism keep up with the instruction level parallelism, the dynamic hash computation should be extended to take the multiple instructions in at the same stage and produce a hash in one cycle. This is possible depending on the hash function, and would require more hardware support, like parallel hash computation logic. In this way, the dynamic hash computation latency does not increase for multiple instructions. Another issue with hash function logic is when the processor clock cycle is reduced, either because of multi-cycle execution or application performance requirement, the allowed time for hash computation in a clock cycle is reduced. There exists a threshold clock cycle, below which the overhead of monitoring could not be hidden. One way to avoid limiting lowest clock cycle is to split the hash computation into pipelined multi-cycle, in a similar way as handling complex hash functions, which will cause extra pipeline registers and control logic. However, the monitoring mechanism remains the same, and the design complexity only increases slightly.

C. ASIP Design Flow

Fig. 10 presents the ASIP design flow to incorporate a program code integrity checker. An automatic synthesis tool - ASIP Meister [53] 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 [54]. After the target ISA has been specified, corresponding resources (such as general purpose register file, ALU, and registers) are selected from a resource library. Meanwhile, extra hardware modules for monitoring (e.g., RHASH and HASHFU as described in Section III) are selected as well. The monitoring microoperations are then embedded into proper instructions, such as branch and jump. Synthesizable VHDL code for the custom ASIP is 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.

VI. EXPERIMENTAL RESULTS

In this section, we present experimental results on evaluating system overheads of the code integrity monitor, and compare
them with results of other previous approaches.

A. Performance Impact of the Checker

As described above, different IHT loading schemes would cause different performance overhead. For the OS-managed IHT loading scheme, very often not all the expected hashes of a program can fit in the IHT. Thus, it encounters hash misses caused by capacity limit, which will incur FHT search and IHT loading and thus execution time overhead. We 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 execution time overhead greatly. The cache-like OS-managed IHT loading scheme is capable of utilizing the spatial locality of basic blocks to reduce the hash miss rate. However, the locality characteristic of programs also varies a lot among programs. In the experiment, we assume that the OS handles hash miss exceptions with a least-recently-used (LRU) replacement policy. To effectively exploit the spatial locality of the hashes in FHT while keeping the hash loading time tolerable, on each hash miss, the OS replaces half of the entries with hash records from the FHT. Fig. 11 lists the miss rate of nine applications in MiBench [55] for different IHT sizes (from 1 to 16 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. If the program is very complex, the limited on-chip IHT table will incur many hash miss. It also shows the necessity to utilize the programs’ execution characteristics to further reduce the execution time overhead.

Since each hash miss will incur OS-managed FHT searching and IHT table replacement, the number of execution cycles will increase. We assume each OS exception handling takes 100 cycles for both FHT searching and IHT table update. Table I gives the cycle number overhead for code monitoring. Column 2 reports the total number of execution cycles for

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Clock cycle (10^6)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No CIC</td>
<td>CIC8</td>
</tr>
<tr>
<td>basicmath</td>
<td>158</td>
<td>149.89</td>
</tr>
<tr>
<td>susan</td>
<td>25.58</td>
<td>25.63</td>
</tr>
<tr>
<td>dijkstra</td>
<td>54.79</td>
<td>57.6</td>
</tr>
<tr>
<td>patricia</td>
<td>133</td>
<td>146.64</td>
</tr>
<tr>
<td>blowfish</td>
<td>32.01</td>
<td>32.32</td>
</tr>
<tr>
<td>rijndael</td>
<td>37.6</td>
<td>45.4</td>
</tr>
<tr>
<td>sha</td>
<td>13.21</td>
<td>15.65</td>
</tr>
<tr>
<td>stringsearch</td>
<td>4.43</td>
<td>6.65</td>
</tr>
<tr>
<td>bitcount</td>
<td>43.62</td>
<td>43.62</td>
</tr>
<tr>
<td>Average</td>
<td>14.7</td>
<td>7.7</td>
</tr>
</tbody>
</table>

For the advanced application-controlled IHT loading scheme, the application would load the IHT with static hashes in a bursting manner. We vary the IHT size from 1 to 16 to examine the overhead reduction trends for each application with the proposed basic block clustering algorithm. With explicit hash loading instructions inserted, no FHT searching procedure is needed and the OS is not involved for miss handling. We assume a hash loading instruction takes one cycle to load a hash into the IHT. Table II gives the detailed statistics for run-time program execution. Column 2 reports, for each test application, the total number of cycles executed for the baseline architecture without code integrity monitor. Columns 3-7 give the number of extra cycles executed for the enhanced architectures with an IHT of 1, 2, 4, 8, and 16 entries, respectively. Columns 7-12 give the corresponding execution time overhead. The overhead varies a lot with different applications because of different CFGs and average number of instructions within a basic block. The average loading overhead of our application-controlled IHT management scheme over the nine applications is only 2.1% for 8 entries, and 1.7% for 16 entries, much smaller than that of the OS-managed scheme.

As our approach degenerates to the previous application-controlled scheme presented in [7] when the IHT size is 1, we compare our approach with the previous one. Fig. 12
TABLE II
CYCLE NUMBER OVERHEAD FOR PROGRAM CODE INTEGRITY CHECKING UNDER THE APPLICATION-CONTROLLED IHT LOADING SCHEME

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Total # of cycles executed (no IHT) (10^6)</th>
<th># of extra cycles executed with different IHT size (10^6)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(cycles)</td>
<td>(cycles)</td>
<td></td>
</tr>
<tr>
<td>basicmath</td>
<td>158</td>
<td>0.3 0.27 0.27 0.27 0.11</td>
<td>0.2 0.2 0.2 0.2 0.07</td>
</tr>
<tr>
<td>susan</td>
<td>23.58</td>
<td>1.88 0.79 0.79 0.79 0.79</td>
<td>1.3 3.1 3.1 3.1 3.1</td>
</tr>
<tr>
<td>dijkstra</td>
<td>54.79</td>
<td>7.94 7.94 7.94 7.94 7.94</td>
<td>13.3 14.5 14.5 14.5 14.5</td>
</tr>
<tr>
<td>patricia</td>
<td>133</td>
<td>0.6 0.6 0.45 0.45 0.42</td>
<td>0.4 0.4 0.3 0.3 0.3</td>
</tr>
<tr>
<td>blowfish</td>
<td>37.07</td>
<td>2.71 1.67 1.67 1.67 1.67</td>
<td>7.3 4.5 4.5 4.5 4.5</td>
</tr>
<tr>
<td>rijndael</td>
<td>37.6</td>
<td>0.52 0.27 0.27 0.27 0.27</td>
<td>1.4 0.7 0.7 0.7 0.7</td>
</tr>
<tr>
<td>alia</td>
<td>17.21</td>
<td>0.82 0.12 0.12 0.12 0.12</td>
<td>0.2 0.9 0.9 0.9 0.9</td>
</tr>
<tr>
<td>stringsearch</td>
<td>4.43</td>
<td>0.016 0.016 0.002 0.002 0.002</td>
<td>0.4 0.4 0.4 0.4 0.4</td>
</tr>
<tr>
<td>bitcount</td>
<td>43.62</td>
<td>8.9 8.62 3.37 3.37 3.37</td>
<td>20.4 19.7 7.5 7.5 7.5</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td>6.5 4.9 3.6 2.1 1.7</td>
</tr>
<tr>
<td>Overhead reduction rate</td>
<td></td>
<td></td>
<td>0% 24.6% 55.4% 67.6% 73.8%</td>
</tr>
</tbody>
</table>

Fig. 12. The normalized program execution time overhead for different IHT sizes under the application-controlled IHT loading scheme

These results show that with the proposed clustering algorithm to utilize application characteristics, the advanced application-controlled IHT loading scheme on our monitor architecture reduces the execution time overhead compared to both previous simple application-controlled checksum loading mechanism and the OS-managed mechanism we have proposed. However, this approach has noticeable static code size increase compared to OS-managed mechanism. Fig. 13 shows the code size increase of nine applications in MiBench for different hash table sizes (varying from 1, 2, 4, 8, to 16), normalized to the original code size without the monitoring mechanism (no IHT). The average code size increase rate for the case of 1-entry IHT is 8.5%. The reason for the small increase rate is that the loading instructions are only inserted in each basic block, which has an average size of 10-100 instructions. The number of extra instructions equals to the number of basic blocks in the CFG. While the dynamic execution time overhead decreases greatly as the IHT table size increases, the static code size normally increases, as shown in Fig. 13. This is because when the table size is larger than one, the basic block clustering technique is applied. To ensure all the basic blocks in a cluster have their hashes loaded in the IHT when program execution enters the cluster, hash loading instructions are inserted at the sites of inter-cluster edges. With multiple input edges for a cluster on average, many basic block hashes get loaded multiple times at different edges, which causes the same loading instruction appearing multiple times in the program. Thus, for many applications, the program code size increases as the IHT table size increases, and for the case of a 16-entry IHT, the average code size increase is 20.1%.

Fig. 13. The normalized code size increase of applications under the application-controlled IHT loading scheme

B. Area Overhead

With the customized processor containing monitoring routines generated, we use ModelSim to simulate the VHDL code [56]. The functionality of applications is verified.

The custom processors were synthesized into technology-mapped gate-level netlists using Synopsys Design Compiler [57] with TSMC’s 0.18µ CMOS standard cell library. We have implemented both the baseline architecture and a number of enhanced processors consisting of the code integrity checker. With a simple “XOR” hash function implemented above, the comparison results are presented in Table III. Column 2 reports the minimum cycle time of the implementations, and column 3 the cycle time overhead compared with the baseline architecture. Considering that it is reasonable that the critical path delay of the synthesis results from Synopsys Design Compiler has a slight variance, the cycle time of the processor presented in the table can be regarded as not
changed at all. This is because normally the critical path of a single-issue pipeline processor is in the execution stage and the extended microoperations with our proposed hash function are added and distributed in the IF and ID stages. Column 4 reports the cell area of the implementations, and column 5 the area overheads normalized to the baseline processor core without IHT, which does not include on-chip caches, i.e., only the datapath, controller, and the register file. These results show that our method involves some tolerable area overhead, which is almost linearly dependent on the size of the table.

<table>
<thead>
<tr>
<th>Designs</th>
<th>Minimum period (ns)</th>
<th>Cycle time overhead (%)</th>
<th>Cell area (%)</th>
<th>Area overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>37.91</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With a 1-entry table (register)</td>
<td>37.93</td>
<td>0.1</td>
<td>2179281</td>
<td>2.0</td>
</tr>
<tr>
<td>With an 8-entry table</td>
<td>37.82</td>
<td>-0.2</td>
<td>2461453</td>
<td>12.4</td>
</tr>
<tr>
<td>With a 16-entry table</td>
<td>38.10</td>
<td>0.5</td>
<td>2597381</td>
<td>21.6</td>
</tr>
</tbody>
</table>

VII. CONCLUSIONS

In this paper, we have proposed architectural support for monitoring the integrity of program code running on embedded processors. We choose the monitoring level of basic blocks and propose an approach to check the instruction stream within each basic block at run-time. The monitor is incorporated into the processor pipeline seamlessly by augmenting the pipeline stages of critical instructions with microoperations. The area overhead is reasonable for an IHT with 8 or 16 entries. The maximum clock frequency from synthesis report does not change at all. Two different IHT loading schemes are proposed and compared, where the OS-managed one allows legacy code to run on the enhanced architecture with larger performance overhead, and the application-controlled one achieves better performance but involves more program instrumentation. 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, combining the two proposed IHT management mechanisms together to achieve advantages from both sides, and also experimenting with more secure yet efficient hash algorithms. Considering the limitation of our monitoring technique in detecting broader range of security attacks, we will also extend our work and combine with other techniques to enhance the system security further. Meanwhile, it will be interesting to design the recovery mechanism either at the architectural or OS level.

REFERENCES


