Compiler-assisted Architectural Support for Program Code Integrity
Monitoring in Application-specific Instruction Set Processors*

Hai Lin†, Xuan Guan†, Yunsi Fei†, Zhijie Jerry Shi†‡
† Department of Electrical and Computer Engineering
‡ Department of Computer Science and Engineering
University of Connecticut, Storrs, CT 06269
E-mail: \{hal06002,xug06002,yfei,zshi\}@engr.uconn.edu

Abstract

As application-specific instruction set processors (ASIPs) are being increasingly used in mobile embedded systems, the ubiquitous networking connections have exposed these systems under various malicious security attacks, which may alter the program code running on the systems. In addition, soft errors in microprocessors can also change program code and result in system malfunction. 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 ASIPs, where both the instruction set architecture (ISA) and the underlying microarchitecture can be customized for a particular application domain. Based on the microoperation-based monitoring architecture that we have presented in previous work, we propose a compiler-assisted and application-controlled management approach for the monitoring architecture. Experimental results show that compared with the OS-managed scheme and other compiler-assisted schemes, our approach can detect program code integrity compromises with much less performance degradation.

1. Introduction

Reliability and security have become critical concerns for embedded processors. 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 [13]. Different from the permanent physical damages in circuits (hard faults), soft errors are intermittent transient faults. They can be triggered by external events such as cosmic rays and only change stored values or signal transfers. Thus, they can escape testing and fault analysis easily. Soft errors may change program code executed in processors and cause the malfunction of computer systems. At a higher system level, security has emerged as a new design goal in addition to the traditional design metrics of performance and power consumption [11]. The vulnerability of systems to malicious software attacks has been increased dramatically due to the increase of embedded software contents and the pervasive networking connections. Many security attacks, including buffer overflow and fault injection attacks, exploit weaknesses in a computer system and execute modified code for malicious purposes. At the instruction level, any code alteration is manifested as bit flips. Monitoring program code integrity can detect any changes made to the program code and thus allows the system to take proper actions to defend against attacks or recover from errors. A single monitor can serve for both reliability and security purposes.

We are interested in embedding program code integrity monitoring mechanisms in Application-Specific Instruction Set Processors (ASIPs), which have emerged as an important design choice for embedded systems because they combine the flexibility of software with the energy-efficiency and high performance of dedicated hardware extensions [6]. ASIPs target a specific application domain, and allow designers to customize both the instruction set architecture (ISA) and the underlying microarchitecture for the specific applications. Very often the target applications are well understood at the design stage. Hence, useful application characteristics can be extracted for improving run-time performance. ASIPs also provide a good platform for integrating code integrity monitoring mechanisms into the design process because of their flexibility in customizing both ISA and microarchitecture.

1.1. Paper Overview and Contributions

In this paper, we propose a compiler-assisted approach for reducing performance overhead of a program code integrity monitor presented in [5]. The monitor ensures that the execution of a program does not deviate from its expected behavior by comparing the dynamic execution properties collected at run-time with the expected ones. When a mismatch is detected, the monitor throws an exception to trigger appropriate remedy mechanisms.

Although there are other hardware-assisted architectural mechanisms for security purposes [4], their separate hardware modules are not directly coupled with microprocessors and thus they usually result in considerable performance and hardware overheads. In our proposed method, both the ISA and underlying microarchitecture are enhanced to incorporate the code monitoring mechanism through microoperation specification. The hardware monitor is seamlessly integrated with the processor pipeline. Therefore the area and performance overheads are small.

To further reduce the performance overhead, we propose an approach that allows applications to manage the monitoring resources, with the assistance of the compiler. With the execution characteristics of embedded

*Acknowledgments: This work was supported by NSF grant CCF-0541102 and NSF CAREER award 0644188.

†Microoperations are elementary operations performed on data stored in datapath registers.
applications collected, the monitor can be utilized more efficiently and thus incur lower performance overhead than using other OS-based or application-based management schemes.

The remainder of the paper is organized as follows. We first give a survey of the relevant previous work in Section 2. Section 3 briefly describes the monitoring architecture. Section 4 details the compiler-assisted management scheme for the monitoring architecture. Section 5 presents experimental results and Section 6 draws conclusions.

2. Related Work

Monitoring code integrity at run-time helps computer systems defend against malicious attacks and recover from soft errors. There has been a lot of work on these two problems. However, most work stays either at high level (software approach) or low level (hardware-based approach).

To prevent security attacks that execute malicious code, checkpoints can be placed at one or multiple software layers in a system. However, there are always new attacks emerging to circumvent the checkpoints placed at certain stages. For example, the OS may 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 [7] [14]. They focus on the memory system and assume the processor itself is secure. Nevertheless, the instructions may be altered when being transferred into the processor or when stored in the instruction window due to soft errors or injected faults. 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 [16]. 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 indicate soft errors. For example, the Boeing 777 aircraft has three processors and data buses [15]. 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 [12]. 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 in parallel (registers and functional units) to execute redundant codes without significant performance degradation.

Recent work by Arora et al. proposed a run-time monitoring mechanism implemented with hardware [4], to monitor the integrity of instruction streams and the inter-procedural and intra-procedural control flow as well. Their monitor is separated from the pipeline, introducing long latencies, and thus has large performance overhead. Our monitor presented in [5] 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 inserting an extra instruction in every basic block to set checksum in the special register, and thus results in significant code size increase and performance degradation. Our work is similar to theirs in terms of compiler-assisted management of the monitoring architecture. However, our monitor is more capable than theirs and allows us to utilize execution characteristics of applications to achieve much smaller performance overhead.

The monitor we proposed in [5] has an internal table coupled with the processor pipeline to store expected behaviors of programs. In [5], we assumed the on-chip table is managed by the operating system (OS). Whenever the program in execution demonstrates a behavior that cannot be found in the table, the OS will take over control. It can update the on-chip expected behavior table. Although the scheme has many advantages, the involvement of OS introduces non-negligible performance overhead. In this paper, we propose an alternative approach, a compiler-assisted application-managed scheme, for managing the on-chip expected behavior table. The new approach incurs much lower performance overhead.

3. Code Integrity Monitoring Architecture

In this section, we first describe the program code integrity monitor we proposed in [5] and then discuss the internal hash table (IHT), one of the important components for storing expected behaviors of programs.

3.1. Code Integrity Monitoring Architecture

We monitor code integrity by comparing two hash values of instruction streams as hash values are a good indicator of code integrity. Any code changes between two hash computations result in different hashes with high probabilities. We compute one hash value before a program starts and consider it as the expected behavior of the program. The other hash is generated by the processor at run-time and considered as the dynamic behavior of the program. Different hashes indicate that the code has changed. The errors that can be detected are determined by the hash algorithms. In [5], we started with a widely adopted assumption of simple faults: considering only a single bit flip in a basic block of program code, and employed a simple checksum function, XOR.

The granularity level to characterize the program’s properties (compute hash in our method) affects the design complexity and effectiveness greatly. We selected to monitor the code integrity at the basic block level [5] because a) it is easy to detect the range of basic blocks with hardware, b) the hash of instructions in a basic block can be computed easily at run-time and before the program starts, and c) any changes to the code are detected promptly at the end of basic blocks, as most of them have less than 100 instructions.

The location of the code monitoring mechanism is also crucial for capturing more potential code changes. We decided to incorporate the monitoring mechanism into the processor pipeline and perform the checking in the instruction fetch (IF) and decode (ID) stages. By doing so, we can detect any code alterations that take place before instructions are fetched into the processor pipeline.

Figure 1 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 in-
struction cache, data cache, and control logic. For the purpose of code monitoring, the pipeline is extended with a Code Integrity Checker (CIC), where an internal hash table (IHT) is set up to store expected hash values, 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 mismatch is found. The OS or recovery code will be notified to respond with actions, e.g., terminating the program.

In the IHT, each entry is a tuple \((Add_{st}, HASH)\) associated with a basic block, where \(Add_{st}\) is the starting address of the basic block and \(HASH\) is the expected hash of instructions in the basic block.

**Figure 1. Block diagram of the proposed monitoring architecture**

In the IF stage, after an instruction is fetched and stored in the instruction register IR, it is also fed into HASHFU. A control signal decides whether HASHFU should start computing hash for a new basic block or continue updating the accumulated hash for the current basic block. In the ID stage, if an instruction is detected indicating the end of a basic block, i.e., a control flow transfer instruction, the IHT is searched for the expected hash value for the basic block. If the basic block is found in IHT, two hashes, the expected hash and the dynamic hash, are compared. A control signal is also generated now to notify HASHFU to start a new round of hash computation from the next instruction.

Since the CIC is incorporated in the processor pipeline, related monitoring microoperations are distributed into different stages and are hidden by the critical path of the processor. Thus, the CIC will not extend the cycle time of the processor. Moreover, since the code monitoring mechanism is working at the microoperation level, a level below the instructions, the mechanism cannot be bypassed by software or compromised by malicious users, thus providing an effective detection measure.

### 3.2. Internal Hash Table

The IHT, where expected hash values are stored, plays an important role in the code integrity monitoring architecture. When program execution proceeds to the end of a basic block, the IHT is searched for the basic block and its expected hash. If the basic block is found in the IHT and the dynamic hash matches the expected one, it is a **hash hit** and we consider that the basic block is intact. If the basic block is found in the IHT but the dynamic hash does not match the expected one (defined as **hash mismatch**), or the basic block is not in the IHT at all (defined as **hash miss**), an exception is raised with different signals indicating the expected hash for each block. All the hash values are simply attached to the application code and data and 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 a hash miss, the FHT will be searched. If the basic block is found in the FHT, one or more hash values will be loaded into the IHT according to the selected replacement policy. If the basic block is not in the FHT either, or the dynamic hash is different from the expected value, the OS may terminate the program.

The OS managed scheme has several advantages. 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 loading of expected hash values is determined by dynamic execution paths, which are not available at compile-time. However, the off-chip memory accesses for searching the FHT on hash misses introduce significant performance overhead.

Alternatively, the expected hash values can be loaded into the IHT by applications, as seen in [10]. If applications manage the IHT, compilers need to insert at proper locations in programs the instructions that load expected hash values, which can be encoded as immediate or obtained from the data cache or memory. Although there is considerable performance overhead in the approach presented in [10], e.g., 18% for application adpcm.encode, we found that the inefficiency does not come from the compiler-assisted mechanism. Because at compile-time, there is plenty of application-specific information that may help achieve higher efficiency. The previous application-managed 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 paper, we propose an approach to exploit the temporal locality of program basic blocks for better performance. We define a hash-loading instruction that specifies explicitly what hash value is loaded into which entry in the IHT. We then propose an effective algorithm to locate the appropriate locations for inserting hash table loading instructions and find the related basic blocks that each loading instruction covers. With sophisticated application-controlled IHT management, assisted by compiler, and the corresponding microarchitectural support, we can utilize the program code integrity monitor with much less performance degradation.

Our previous work focused on microarchitectural modifications for monitoring and assumed an OS is in place to handle IHT misses and replacements [5]. In this work, we adopt the similar run-time monitoring microarchitecture. However, we study alternative application-controlled IHT management schemes.
4. Application-controlled IHT Management

We propose a compiler-assisted application-controlled IHT management scheme, where the hash values of a series of basic blocks can be pre-loaded to on-chip resources for monitoring the execution in a time window. Before the execution reaches the end of a basic block, the corresponding hash should have been placed in the IHT for run-time integrity checking. This is very challenging.

Previous research work has shown that loading the proper hash value in each basic block introduces a considerable performance overhead to application execution [10]. Our approach differs from the previous work by exploiting application-specific information for improving performance. With the multi-entry IHT, we are able to load multiple hash values once into the IHT but to monitor several basic blocks that are going to execute consecutively and repeatedly. The in-processor IHT is acting as a cache to the full hash table (FHT) in memory. Similar to the cache prefetching techniques [8], our approach exploits the application-specific execution characteristics to direct the preloading of IHT values for reducing the performance overhead. The IHT will be better utilized if the hash values for the basic blocks involved in frequently executed loops are pre-loaded together. Figure 2 illustrates one simple control flow graph (CFG) example for a program, where several nodes, $bb_1$, $bb_2$, and $bb_3$, form a loop. Each node represents a basic block, and the edge between nodes indicates a 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. Assume the number of IHT entries is 4, the hash values of the three basic blocks can be loaded only once before the loop is executed (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 in $bb_0$, 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 checksum register scheme, every basic block execution has to load its hash once, and the total loading cost for the loop is $3(n + 1)$ time units. It is intuitive to see that bursting pre-loading can reduce the performance overhead greatly.

![Figure 2. A simple CFG with a loop](image)

4.1. Problem Definition

The problem then boils down to find an optimal loading policy for the IHT with a fixed number of entries for any programs, 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 FHT is not searched by the OS. With multiple entries in the IHT, 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 of one batch. $G(V, E, W)$ denotes the graph partitioning problem, 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 for this edge 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 is less than the size of the IHT, $K$, so that the total inter-cluster switching cost is minimum (we will give the detailed cost function later).

4.2. Algorithm Description

A graph partitioning problem as described above is known to be NP-complete. We develop an iterative-improvement based heuristic algorithm for this problem. 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 our 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, and our algorithm will improve the solution through iterations. The outermost loop of the algorithm (Lines 4-36 in Algorithm 1) is for controlling the maximum number of entries in the IHT ($K$). We also set other two constraints for our 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, and our algorithm will improve the solution through iterations. The outermost loop of the algorithm (Lines 4-36 in Algorithm 1) is for controlling the maximum number of entries 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.

As edges in a CFG represent control 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.
For a clustering solution, two lists can be generated: a single node list and a multi-node cluster list. Each improvement iteration contains two phases (Lines 6-34). 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 which, when merged with the node, decreases the cost most. 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 this node again), we process the nodes in the single node list in a random order at phase 2.

The costs for moves employed in phases 1 and 2 are evaluated separately, as illustrated in Figure 3. 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 $j$. And 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. 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 $a$ and $r$. Node $j$ also has other internal incoming and outgoing edges, with their weights as $m$ and $n$. 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. Also, 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 $j$ (size change 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 $j$, where each inter-cluster loading cost is the product of edge weight and the size of the destination cluster.

![Figure 3. Costs of different moves](image)

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. $S_{new} = S_0$;
4. while $num < K$ do
5.   $counter = 0$;
6.   while $counter < R$ and $cost_{pass} < M$ do
7.     $S = S_{new}$; $cost_{pass} = 0$;
8.     get two lists for $S$: single_node_list and cluster_list;
9.     for each cluster $clr_i$ in the cluster_list do
10.    $n_{select} = null$, $\Delta cost_e(n_{select}, clr_i) = \infty$;
11.    for each node $n_j$ in $clr_i$ do
12.       if $\Delta cost_e(n_j, clr_i) < \Delta cost_e(n_{select}, clr_i)$ then
13.           $n_{select} = n_j$;
14.       end if
15.     end for
16.     eject $n_{select}$ from $clr_i$ to single_node_list, update the cluster_list;
17.     $cost_{pass} = cost_{pass} + \Delta cost_e(n_{select}, clr_i)$;
18.     end for
19.     for each node $n_j$ in single_node_list do
20.        $clr_{select} = null$, $\Delta cost_m(n_j, null) = \infty$;
21.        for each cluster $clr_i$ that $n_j$ is connected to do
22.           if $\Delta cost_m(n_j, clr_i) < \Delta cost_m(n_j, clr_{select})$ and sizeof(clr_i) < num then
23.              $clr_{select} = clr_i$;
24.           end if
25.       end for
26.       merge $n_j$ and $clr_{select}$, and update the cluster_list and single_node_list;
27.       $cost_{pass} = cost_{pass} + \Delta cost_m(n_j, clr_{select})$
28.     end for
29.     a new clustering solution $S_{new}$ is generated;
30.     if $cost \geq M$ then
31.         $S_{new} = S$;
32.     end if
33.     $counter++$;
34.     end while
35.     $num++$;
36. end while
37. $S_{final} = S_{new}$; Output the final clustering solution $S_{final}$ and the total cost.
By applying many such ejecting and merging moves in a pass, each iteration can possibly reduce the loading cost. The cost increase for an iteration is defined as \( \text{cost}_{\text{pass}} \). For each round with certain \( \text{num} \) value, the iteration loop stops when either \( \text{cost}_{\text{pass}} \) reaches a pre-set value (normally set to 0 for convergence) or the number of iterations has reached the maximum allowed value (Line 6 in Algorithm 1). After \( K - 1 \) rounds, the final clustering solution and the total loading cost are output.

The complexity of the algorithm is \( N^2 \cdot (K - 1) \cdot M \), where \( N \) represents the total number of nodes in the CFG, \( K \) is the size of the IHT, and \( M \) is the maximum allowed iterations. Our simulation results show that with a reasonable number of iterations, the clustering algorithm can achieve a significant reduction in the loading overhead compared to the scheme of loading checksum values at the beginning of every basic block.

5. Experimental Results

In this section, we first briefly describe the design methodology for incorporating a program code integrity monitor into ASIPs, then present experimental results on evaluating system overheads of the code integrity monitor, and finally compare them with results of other approaches.

5.1. Hardware/Software Co-design Methodology for the Code Integrity Monitor

We employed an automatic synthesis tool - ASIP-Meister [1] to incorporate a program code integrity monitor in an embedded processor. The ISA can be specified to include an IHT loading instruction, and the corresponding regular resources for the datapath and extra hardware modules for the monitor are selected. Synthesizable VHDL code for the custom ASIP are generated from the ASIP Meister HDL generator. The associated reconfigurable software toolset, including a compiler, simulator, and assembler, is also automatically generated for the customized processors. Due to the space limit, we skip the implementation details [5, 9].

5.2. Program Code Size Impact

We adopted the SUIF [3] tool set for program profiling and obtained the CFG for our clustering algorithm. Given the number of entries in the IHT, our clustering algorithm can determine where in the program to insert loading instructions, and how many instructions are inserted together for a cluster of basic blocks. Both the code size increase and loading time overheads are evaluated at the end of our algorithm.

Figure 4 shows the code size of eight applications in MiBench [2] 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). When the number of entries is 1, each basic block has its own loading instruction, and our approach degenerates to the checksum loading method as presented in paper [10].

The average code size increase rate for the case of 1-entry IHT is 5.13%. 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. When the table size is larger than one, loading instructions are inserted at each inter-cluster edge. Thus, for many applications, the program code size increases slightly as the table size increases.

5.3. Performance Impact

The program performance will be degraded with the integrity checking mechanism because additional instructions are executed at run-time to load hash values. Table 1 gives detailed statistics for run-time program execution. Column 2 reports the total number of program instructions executed for the baseline architecture without code integrity monitor, i.e., a single-issue 6-stage PISA processor. Columns 3-7 give the number of extra instructions executed for the enhanced architectures with an IHT of 1, 2, 4, 8, and 16 entries, respectively. Columns 7-12 give the corresponding performance overhead. Note that the average loading overhead over the eight applications is only 3.6% for 8 entries, and 2.3% for 16 entries with the basic block clustering algorithm.

As our approach degenerates to the method presented in [10] when the IHT size is 1, we next make comparisons between our approach and the previous one. Figure 5 demonstrates the normalized performance overhead as the IHT size varies from 1 to 16. The performance overhead is defined as the total number of loading instruction executions normalized to the number for the case of 1-entry IHT. Different curves represent different benchmarks, and each dot on a curve corresponds to a IHT table size. Our experimental results show that for all the benchmarks, the performance overhead decreases as the table size increases, especially when the IHT entry number reaches up to 8 or 16. The average performance overhead reduction for a 8-entry IHT is 49.5%, and 66.4% for a 16-entry IHT. With a considerable cluster size (IHT table size), many loops of basic blocks can be grouped into clusters. Thus, the loading overhead is greatly reduced.

The area overhead and the fault coverage and analysis are the same with the OS-managed scheme. Please refer to the previous work [5] for more details.

6. Conclusions

In this paper, we present an approach of compiler-assisted architectural support for monitoring the program code integrity in embedded processors. A monitoring mechanism is formulated to check the integrity of the instruction stream within each basic block at runtime. The integrity monitor is incorporated into the processor pipeline seamlessly by augmenting the IF and ID stages of critical instructions with microoperations.
Table 1. Run-time Loading Overhead

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Total # of instr. executed (no IHT)</th>
<th># of extra instr. executed with different IHT size</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>qsort</td>
<td>6117769</td>
<td>548070</td>
<td>548070</td>
</tr>
<tr>
<td>sha</td>
<td>26731742</td>
<td>915786</td>
<td>813904</td>
</tr>
<tr>
<td>dijkstra</td>
<td>97874391</td>
<td>801906</td>
<td>7141306</td>
</tr>
<tr>
<td>bitcount</td>
<td>27226971</td>
<td>1550107</td>
<td>450009</td>
</tr>
<tr>
<td>stringsearch</td>
<td>51779</td>
<td>4217</td>
<td>4217</td>
</tr>
<tr>
<td>blowfish</td>
<td>78914820</td>
<td>2151427</td>
<td>1893357</td>
</tr>
<tr>
<td>patricia</td>
<td>6192945</td>
<td>615541</td>
<td>615541</td>
</tr>
<tr>
<td>susan</td>
<td>26843455</td>
<td>753002</td>
<td>499758</td>
</tr>
</tbody>
</table>

Average                                                                 6.2  5.4  5.0  3.6  2.3

Figure 5. The normalized program performance overhead for different IHT sizes

To reduce performance overhead, we propose a novel compiler-assisted clustering algorithm to group related basic blocks into clusters and insert hash-loading instructions in the program. At run-time, the program is directed to load hash values of basic blocks into the IHT in a bursting fashion. Compared to previous approaches that load only one hash value each time, the new method reduces the performance overhead greatly because hash values are loaded at the coarser granularity level of clusters and stay in the IHT for repeated uses. 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.

References