


Article

Hardware-Implemented Security Processing Unit for Program Execution Monitoring and Instruction Fault Self-Repairing on Embedded Systems

Zhun Zhang , Xiang Wang *, Qiang Hao, Dongdong Xu, Jiqing Wang, Jiakang Liu, Jinhui Ma and Jinlei Zhang

School of Electronic and Information Engineering, Beihang University, Beijing 100191, China; microzhun@buaa.edu.cn (Z.Z.); haoqiang1994@buaa.edu.cn (Q.H.); xudongdong1994@buaa.edu.cn (D.X.); wangjiqing@buaa.edu.cn (J.W.); ljiakang@buaa.edu.cn (J.L.); sy2002514@buaa.edu.cn (J.M.); zhangjinlei@buaa.edu.cn (J.Z.)

* Correspondence: wxiang@buaa.edu.cn; Tel.: +86-10-8231-3686

Abstract: Embedded systems are increasingly applied in numerous security-sensitive applications, such as industrial controls, railway transports, intelligent vehicles, avionics and aerospace. However, embedded systems are compromised in the execution of untrusted programs, where the instructions could be maliciously tampered with to cause unintended behaviors or program execution failures. Particularly for remote-controlled embedded systems, program execution monitoring and instruction fault self-repair are important to avoid unintended behaviors and execution interruptions. Therefore, this paper presents a hardware-enhanced embedded system with the integration of a Security Processing Unit (SPU) in which integrity signature checking and checkpoint-rollback mechanisms are coupled to achieve real-time program execution monitoring and instruction fault self-repairing. This System-on-Chip (SoC) design was implemented and validated on the Xilinx Virtex-5 FPGA development platform. Based on the evaluation of the SPU in terms of the performance overhead, security capability, and resource consumption, the experimental results show that, while the CPU executes different benchmarks, the average performance overhead of the SPU lowers to 1.92% at typical 8-KB I/D caches, and it provides both program monitoring and fault self-repairing capabilities. Unlike conventional hardware detection technologies that require manual handling to recovery program executions, the CPU-SPU collaborative SoC is a resilient architecture equipped with instruction tampering detection and a post-detection strategy of instruction fault self-repairing. Moreover, the embedded system satisfies a good balance between high security and resource consumption.

Keywords: embedded system; hardware security; security processing unit (SPU); program monitoring; instruction fault self-repairing



Citation: Zhang, Z.; Wang, X.; Hao, Q.; Xu, D.; Wang, J.; Liu, J.; Ma, J.; Zhang, J. Hardware-Implemented Security Processing Unit for Program Execution Monitoring and Instruction Fault Self-Repairing on Embedded Systems. *Appl. Sci.* **2022**, *12*, 3584. <https://doi.org/10.3390/app12073584>

Academic Editor: Arcangelo Castiglione

Received: 2 March 2022

Accepted: 29 March 2022

Published: 1 April 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

State-of-the-art embedded systems are increasingly employed in various applications due to their superior features of high processing performance, low power consumption, and good functional adaptation. However, in some security-critical application scenarios, the unintended behaviors of programs could jeopardize precious human lives and expensive scientific instruments. Thus, high-security capability is equally important for embedded system designs, especially for embedded processors, which are applied as the control kernels in these safety-sensitive scenes of automotive, aerospace, avionics, and railway transport [1–4].

Therefore, the hardware-enhanced security protection is an important consideration in System-on-Chip (SoC) architecture. In previous reports, the various forms of hardware-oriented attacks have been implemented from the different-level sources, and can be categorized into the two main types: hardware-level attack and software-level attack.

At the hardware-level attack, a malicious logic (well-known as a hardware Trojan) is a typical example that might be inserted into an untrusted third-part intellectual property (IP) beforehand and become activated under a specific condition to modify the processor's behaviors, cause program execution failure [5], or even create backdoors for confidential data leakage and control system hijacking to attackers [6]. On the other hand, software-level attacks mainly exploit the security vulnerabilities or bugs in software applications to disturb program executions or perform other unintended actions, such as tampering with the program code or injecting malicious code.

Up to the present, most embedded programs and applications have been developed based on the high-level programming languages of C and C++, which access memory without any valid bound checks, and this situation will bring serious security risks to remote-controlled embedded systems. For instance, a famous case is the American Lockheed Martin RQ-170 sentinel unmanned aerial vehicle (UAV) that was captured by Iranian forces through instruction tampering attacks and false coordinate injections [7]. Moreover, the software-level attacks also easily cause memory buffer overflows [8–10] via stack smashing and then take control of hardware platforms during program execution.

Various techniques have been proposed to protect SoCs against hardware-level and software-level attacks. For resisting hardware Trojan attacks, existing techniques using standard functional validation [11] and side-channel analysis [12,13] can analyze and detect hardware for a hardware Trojan. However, the large scale integrated circuit chip after manufacturing technology makes it expensive and time-consuming to analyze all IPs. Furthermore, some purposefully inserted hardware Trojans are designed to be activated only by very rare events under a specific execution condition, which is difficult to detect in the function validations.

Therefore, it is critical and challenging to eliminate the risks of these hardware-level attacks. For defending against software-level attacks, the multivariant execution technique (MET) [14], control flow integrity (CFI) [15], execute-only memory (XOM) [16], address space layout randomization (ASLR) [17], etc. have been proposed to monitor whether the program execution is following the intended behaviors. However, most protection techniques require extended instruction set architectures (ISAs) and modified compilers, which make them difficult to transplant into different embedded systems.

The instruction codes of embedded programs could be deliberately tampered with—the attacks originating from both hardware-level and software-level attacks. Although some sapiential monitoring strategies have been proposed to observe whether program execution is under attack or not, it is not yet sufficient to ensure the correct completion of program execution only by detecting unintended behaviors. A fast fault self-repairing capability after discovering the unintended behavior or instruction fault is critically important to providing a comprehensive protection for an embedded system, especially for embedded systems that are applied in remote-controlled platforms.

At present, the majority of fault self-repairing techniques rely on the checkpoint and rollback-recovery technology, in which the checkpoint is utilized to backup all the data of one correct state in the processor system, while the rollback operation reverts the process back to a recently-saved checkpoint once a processor fault is detected and then resumes the normal program execution. In addition, many researchers have proposed hardware redundancy strategies, such as dual modular redundancy (DMR) [18] and triple modular redundancy (TMR) [19] to achieve hardware fault tolerance.

However, the approach of simply replicating the structures of sensitive components will result in a high hardware complexity and resource overhead in a resource-limited embedded system. To the best of our of knowledge, attack detection and fault recovery are mostly studied independently in the existing hardware monitoring methods, and their excellent combination could help to improve the security capability of embedded system against various attacks.

After we comprehensively assessed the advantages and disadvantages of previous protections, the security designs for embedded systems still face the following three challenges:

(1) how to reduce the performance overhead induced by real-time security validation; (2) how to configure a processor with a fault self-repairing capability; and (3) how to ensure the practicality of hardware security design to be equally applicable for other different ISA platforms.

In order to overcome the above three challenges of embedded system security architecture, this paper presents a hardware-implemented security processing unit (SPU) for providing real-time program execution monitoring and instruction fault self-repairing, where the internal structure consists of an instruction monitor and a fault self-repairing module (FSRM). The checkpoint-rollback mechanism applied in FSRM is tightly coupled to the program basic block (BB) integrity signature monitoring mechanism of the instruction monitor. The specific contributions of this paper are summarized as follows.

- A security processing unit (SPU) is constructed into the embedded system for building the proposed CPU–SPU collaborative SoC. In the SPU, both the instruction monitor and FSRM were constructed to monitor the program execution and instruction fault self-repairing in real-time, where the real-time program execution monitoring was actualized by the instruction monitor, and the instruction fault self-repairing was actualized by FSRM.
- The program basic block's (BB's) integrity checking mechanism is adopted in the instruction monitor, which is tightly coupled with the checkpoint-rollback mechanism of FSRM. Any instruction tampering behaviors caused by hardware Trojans will be detected and self-repaired by the SPU, and artificial program modification also can be detected.
- In order to reduce the performance overhead of SPU induced by integrity verification and fault self-repairing, the I/D-Caches and monitor cache (M-Cache) are felicitously configured with the optimal size of 8 KB, and the average performance overhead of SPU in running different benchmarks reduces to as low as 1.92%.
- Security capability evaluation and hardware implementation evaluation of the SPU in both the FPGA platform and ASIC design confirm that the CPU–SPU collaborative SoC achieved a good balance in high-security capability, low performance overhead, and reasonable hardware complexity.

The remainder of this paper is organized as follows. Section 2 introduces the trustworthy assumptions and threat models of SoC considered in this paper. The preparatory works related to program monitoring and fault self-repairing are described in Section 3. Section 4 presents the hardware implementations of the instruction monitor and fault self-repairing module (FSRM), as well as the performance optimizations of the real-time program monitoring mechanism. Experimental evaluations of the SPU regarding the performance overhead, security capability, and practicality comparison are implemented in Section 5. Section 6 presents SoC overall hardware implementation in FPGA and ASIC. This paper is concluded in Section 7.

2. Trustworthy Assumptions and Threat Models

Before integrating the proposed security processing unit (SPU) into an embedded system to establish the expected CPU–SPU collaborative SoC, the trustworthy assumptions and threat models aiming at embedded system should first be determined. Generally, some sensitive components in embedded system are likely susceptible to malicious attacks; thus, we should make their associated trustworthy assumptions and focus on security protections according to potential threat models. Hence, we make the following assumptions regarding the CPU–SPU collaborative SoC hardware architecture, the potential malicious attacks, and the program monitoring and instruction recovery mechanism.

We plan to integrate the hardware-implemented SPU with an open-source reduced instruction set computing (OpenRISC) processor OR1200 for constructing a secure CPU–SPU collaborative SoC architecture. In this, the central processing unit (CPU) of OR1200 is a 32-bit scalar RISC softcore processor with a Harvard micro architecture. This satisfies the GNU general public license (GPL) protocol, which is supported by the OpenCores

organization. The CPU core consists of a five-stage execution pipeline: the instruction fetching (IF) stage, instruction decoding (ID) stage, instruction executing (EX) stage, memory accessing (MA) stage, and writing back (WB) stage, which is a single-emission sequence execution pipeline.

Moreover, the CPU core is configured with instruction memory management unit (IMMU) and data memory management unit (DMMU). For decreasing the average delays in fetching instructions and data from the external memory, the instruction cache (I-Cache) and data cache (D-Cache) were configured and connected with the addressable quick memory (QMEM), the temporal locality principle and spatial locality principle of program execution in accessing cache memory can be fully exploited to keep up with CPU's execution pipeline. The CPU-SPU collaborative SoC hardware architecture is shown in Figure 1.

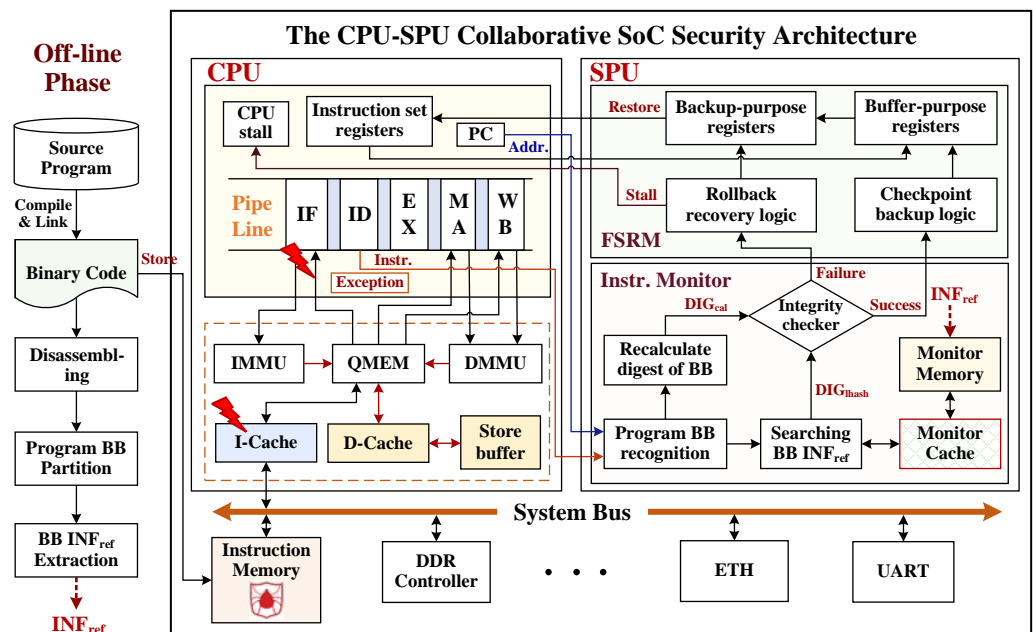


Figure 1. The CPU-SPU collaborative SoC hardware architecture for real-time security protection.

Since all the program operations are performed under the control of CPU, the components directly connected to the CPU are the critical targets of interest for inserting hardware Trojans or injecting fault attacks, and thus an erroneous instruction could pose a serious security challenge to embedded system. We assume the whole SoC architecture is an integration of IPs, and many of which are acquired from the untrusted third-party vendors for shortening the time-to-market of its applied products.

We reasonably make the trustworthy assumption regarding the proposed CPU-SPU collaborative SoC architecture that our self-designed SPU was highly tested and validated that anywhere without hidden hardware Trojans being inserted within internal control logic. However, the CPU core, cache memory, register file, main memory, etc. are sensitive components that are likely to suffer from the malicious tampering attacks during program executions.

In addition, many memory IPs (including RAM technology library) need to acquire from the third-party vendors for SoC convenient taping-out, this will further aggravate the concerns of hardware confidence. According to the different locations where program tampering attacks may occurred by hardware Trojans or fault injection, there are four types of threat model assumptions, which are common-used approaches to disturb CPU normal program execution in reported researches:

- The first situation is that instruction codes were modified or tampered with in the processor core. The authors in [20] designed a sequential hardware Trojan, which was activated by a sequence of rare events to modify instruction codes before the

ID stage. In addition, a capacitor-based triggering Trojan IP was embedded into a CPU [21], which leveraged an analog circuit to siphon charges from nearby wires and enforced the victim to flip-flop to a desired value. These two examples indicated that the instruction register (IR) and the instruction bus to IR in CPU were vulnerable to hardware Trojan attacks; therefore, their instruction code faults should be detected and repaired.

- The second situation is that instructions were maliciously modified in I-Cache instead of processor core. The hardware designers in [6,22] were developing specific hardware Trojans towards CPU cache memories to modify the cached instruction codes, here, the instruction modifications in I-Cache are manifested as the data bit flips.
- The third situation is that instruction codes were modified in the instruction memory (such as flash memory) induced by hardware Trojans, where the main memory was hidden for the sake of brevity. For example, the designers in [23,24] were leaning flash-memory-oriented hardware Trojans to tamper with the instructions in non-volatile memory.
- The fourth situation is that instruction codes that were modified by artificial intentions arose from the program code phase (including software and application) before compiling and linking. For example, the programs in C and C++ programming languages were modified via malicious code injections to take control of hardware platforms [25].

The above-mentioned four threat models are assumed according to the system's potential locations where instruction tampering attacks may occur, which is induced by both human and hardware Trojans. Hence, an instruction monitor is implemented into the SPU to monitor the instruction execution security in real-time. For convenience, the instruction tampering detection can be normalized to aim at the instruction register (IR) monitoring, which is the final phase of instruction codes before entering the CPU's execution pipeline, this monitoring mechanism can not only effectively guarantee secure program executions against the above-mentioned four threat models but also facilitates hardware implementation of the instruction FSRM.

The post-detection strategy of the instruction fault self-repairing scheme is tightly coupled with the real-time instruction integrity monitoring. When a tampered instruction is executed and detected, the self-repairing is triggered and reverts to the recently-saved checkpoint for resuming normal program execution. This recovery scheme is adopted based on the strong randomness of hardware Trojan activation. It is noteworthy that source program tampering by an adversary before compiling and linking processes can also be detected and reported but cannot be self-repaired by the checkpoint-rollback scheme.

3. Preliminaries

Our proposed CPU–SPU collaborative SoC architecture focuses on achieving rapid instruction fault identification and fault self-repairing. This section presents the basics of the program execution monitoring and checkpoint-rollback mechanism in embedded systems to inspire implementation of the proposed CPU–SPU collaborative SoC architecture.

3.1. Program Execution Monitoring Based on Basic Block

Before integrating an instruction monitor into an SPU to monitor whether the CPU's executive instructions were modified or not, it is essential to preprocess instruction codes (InsCodes) for extracting the reference information of BB integrity monitoring. The offline preparation phase before the program execution contains three main stages: the partition of program basic blocks (BBs); the BB reference information extraction; and the integrity signature calculation of each BB.

3.1.1. Partition of the Program Basic Block

According to our previously-reported literature of instruction execution security [26], hardware-assisted integrity signature monitoring is an effective technique to monitor unintended program behaviors. In that scheme, the program basic block (BB) is defined as

a sequence of consecutive instructions, which starts from the program's first instruction and ends with the branch or jump instruction. Each BB block is assigned with an integrity signature for the real-time BB integrity verification. The integrity signatures of the BBs are calculated beforehand from the associated instruction information and then copied onto the monitor memory before the program is loaded for execution. During program execution, the integrity signature of each BB is recalculated from the executed instructions and compared with the previously-stored ones to validate the BB instruction integrity.

In general, the performance overhead of hardware monitoring is relevant to the integrity signature checking speed, and further, the speed of integrity validation depends on the BB granularity. Therefore, we partition programs strictly according to the transfer type instructions, where BB is a fine-grained partition scheme to monitor the potential instruction modified issues of deletion, tampering, and injection, and it can keep up with the CPU's execution pipeline.

The program execution order can be regarded as having a natural hierarchical structure according to the instruction transfer operations. Taking the six transfer instructions from the OpenRISC ISA as an example, their instruction jump operations are described in Table 1. The six transfer instructions can be conveniently categorized into the direct branch (*l.bf N* and *l.bnf N*), direct jump (*l.j N* and *l.jal N*), and indirect jump (*l.jalr rB* and *l.jr rB*) according to the instruction transfer types.

In this, when the direct branch instructions of *l.bf N* and *l.bnf N* branch or not (depending on whether the status register (SR)'s flag-bit are set or cleared), the effective branch address (EBA) after instruction operation will be sent to the program counter (PC); the direct jump instructions of *l.j N* and *l.jal N* directly send their jump target addresses to PC; and the indirect jump instructions of *l.jalr rB* and *l.jr rB* jump to the jump target address in the rB register. The *l.jalr rB* instruction stores the next instruction address of the delay instruction into the link-address register (LR).

Depending on these six transfer type instructions, the GNU Cross Compilation Toolchain of or32-elf-gcc matched with OpenRISC ISA is utilized to generate a binary executable file; and the GNU tool or32-elf-objdump is utilized to disassemble the binary executable file into a text file. We can employ regular expressions to search all the function entries, jump instructions, and target addresses; therefore, we can partition programs into BB segments from disassembling text files at the offline preparation phase.

Consequently, a sequence of consecutive instructions of each BB starts from the program's first instruction (or the last jump instruction's target address) and ends with the next transfer instruction; therefore, there is no branch or jump transfer operation in the middle of each BB. It is noteworthy that, in the five-stage execution pipeline of OpenRISC CPU, it takes at least two clock cycles between fetching instruction from the target transfer address and entering the instruction execution that will cause CPU execution pipeline discontinuity, where the branch delay slot is the wasted clock spaces following the conditional branch and jump instructions.

We considered the delay slot mechanism to reduce processor performance loss. To improve the execution efficiency of CPU, the delay slot instruction was also partitioned into each BB for filling the execution pipeline clock gap to follow the branch or jump instruction as the end boundary of each BB. This program BB partitioning strategy can minimize the processor-itself performance overhead.

Applying the fine-grained program BB partitioning strategy to the real application program, as shown in Figure 2, a segmentation of consecutive instructions was selected from the benchmark of OpenECC to illustrate the program BB partitioning details. First, the sequence of consecutive instructions can be partitioned easily into BB1, BB2, and BB3 according to the boundaries of the branch and jump instructions, where the branch and jump instructions are followed with the delay slot instruction, respectively. Secondly, the instruction transfer target address of each BB can be deduced according to the transfer-type instruction.

Table 1. The transfer type instructions in the OpenRISC instruction set architecture (ISA).

Instruction	Example	Instruction Operation	Transfer Type
<i>l.bf</i> N	<i>l.bf</i> 0x3	EA ← exts(Immediate << 2) + Bran.Instr.Addr. PC ← EA if SR[Flag] set	Direct branch
<i>l.bnf</i> N	<i>l.bnf</i> 0x6	EA ← exts(Immediate << 2) + Bran.Instr.Addr. PC ← EA if SR[Flag] cleared	Direct branch
<i>l.j</i> N	<i>l.j</i> 0x3	PC ← exts(Immediate << 2) + JumpInstr.Addr.	Direct jump
<i>l.jal</i> N	<i>l.jal</i> 0x3	PC ← exts(Immediate << 2) + JumpInstr.Addr. LR ← DelayInstr.Addr.+4	Direct jump
<i>l.jalr</i> rB	<i>l.jalr</i> r2	PC ← rB LR ← DelayInstr.Addr.+4	Indirect jump
<i>l.jr</i> rB	<i>l.jr</i> r9	PC ← rB	Indirect jump

For example, the instruction *l.bf* is a conditional branch instruction, and two potential legal branch addresses can be inferred from the analysis of branch instruction code. The absolute jump instruction *l.jr* jumps to the target address corresponding to the value of the r9 register, which is usually the returned address of the superior function. Although the value of the r9 register cannot be extracted in offline analysis phase, a new BB can be created by processing the call function entry address and traversing its target jump address.

Thus, considering the branch target addresses of two *l.bf* conditional branch instructions, BB3 and BB4 can be generated, where BB3 is an overlap with the previous BB3 and BB4 is inside BB2 from a new start address. Therefore, we only need to reserve one BB3 to avoid duplicated BB integrity checks. This will help to reduce the on-chip storage resource overhead in storing BB integrity signatures. Ultimately, the instruction stream can be partitioned into the four BBs.

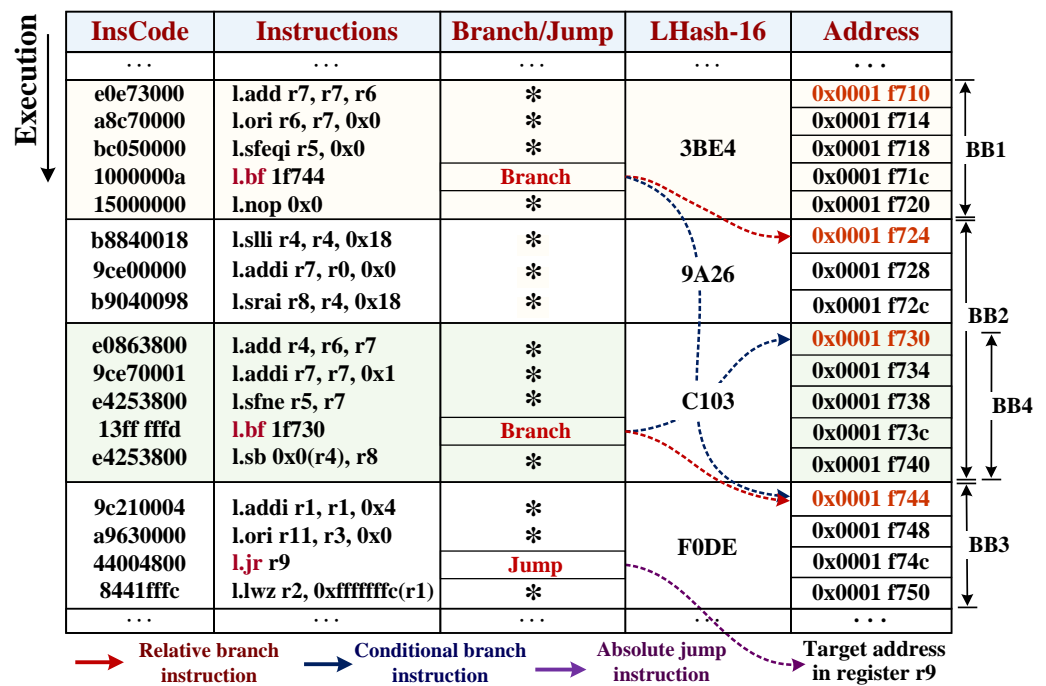


Figure 2. Example of partitioning the benchmark of OpenECC into program basic blocks.

3.1.2. Extraction of Reference Information

The intention of partitioning the instruction stream into BBs enables modular security checks for the proposed instruction monitor with minimal performance loss and great security. In the design of the instruction monitor, the reference information (INF_{ref}) of divisory BBs should be predefined to determine each BB's integrity monitoring parameter.

To satisfy the security monitoring requirements against the various forms of instruction tampering attacks and achieve a quick integrity verification, the BB INF_{inf} constitution

requires meeting three conditions: (1) it must be associated with each BB, and its value for each BB is unique so that they can be searched for quickly and accurately; (2) it must be sensitive to any damage issue so that the injection, deletion, or tampering of instruction will cause the security monitoring parameter to change; and (3) it is easy to extract from each BB—its number of bits should not be overly long while ensuring adequate security with storage-resource-limited SoCs.

After the above comprehensive consideration, we plan to extract the effective start address (ADD_{start}) and the integrity digest of each BB to constitute the expected 32-bit integrity monitoring INF_{ref} , organically. The BB digest is generated by employing a light-weight hash function (DIG_{lhash}) to calculate all the instruction codes (InsCodes) of each BB. Our adopted OpenRISC processor OR1200 has a 32-bit instruction code and target address, whose instruction and address are aligned to 4 bytes. Due to the lower 2 bits of the 32-bit instruction address in the program counter (PC) being fixed to 2'b00 (addressing RAM by word), the available value as the effective start address of each BB is PC[31:2].

In general, the width of a 32-bit address can provide 4 GB of address space, where the PC[31:2] value of the BB start address leads to large on-chip storage consumption when it constitutes the integrity reference information of INF_{ref} . For this reason, we selected the lower 16-bit effective values from PC, that is, PC[17:2], as the start address value of each BB into INF_{ref} , which can provide the applications with up to 256 KB address space. Furthermore, the size of the address space can be extended by selecting more effective bits from PC[31:2] according to the real application requirements, and its storage resource overhead will also increase.

In this work, the 32-bit INF_{ref} is composed of the 16-bit $ADD_{start}[17:2]$ and the 16-bit golden DIG_{lhash} , where the $ADD_{start}[17:2]$ is assigned to $INF_{ref}[31:16]$ for identifying the current BB and searching its corresponding INF_{inf} from the reference information table; and the 16-bit golden DIG_{lhash} is assigned to $INF_{ref}[15:0]$ for checking the BB digest integrity.

3.1.3. Integrity Signature Calculation

As described above, the extraction of INF_{inf} from each BB requires the LHash algorithm to calculate the BB digest signature for integrity monitoring. This light-weight cryptographic hash algorithm was first reported in the study [27], where an internal permutation employed the kind of Feistel-PG extended sponge structure for utilizing the permutation layers on nibbles to improve the diffusion speed. This was a hardware-friendly hash function (only 817 gate elements) and was suitable to be implemented in embedded systems to quickly transform a given sequence of instructions into a fixed bit-number of integrity signature.

As shown in Figure 2, the LHash algorithm was utilized to calculate the instruction information and generate the 16-bit integrity signatures for BB1 to BB4, respectively. Furthermore, an important consideration of applying the LHash algorithm is utilizing its sequential iteration mechanism in the sponge structure to recalculate the integrity signature of a sequence of instructions during program execution, which is then compared with the value of the previously-stored LHash-16 integrity digest in the INF_{inf} table. Consequently, this algorithm will quickly complete the integrity signature calculation under a low hardware complexity.

3.2. Fault Self-Repairing Based on Checkpoint-Rollback Scheme

The program execution monitoring was proposed to detect the embedded system whether is under attack or not. However, it is not sufficient to complete program executions only by detecting unintended behaviors caused by malicious attacks, the fast fault self-repairing capability after discovering instruction faults is also critical important to provide comprehensive protection for an embedded system. The fault self-repairing based on the checkpoint-rollback scheme contains two main stages: the object and time selections for checkpoint backup; and the buffer and backup registers for checkpoint rollback.

3.2.1. Object and Time Selections for Checkpoint Backup

As the name checkpoint-rollback recovery suggests, checkpoint backup and checkpoint rollback are two independent operations for the fault self-repairing module (FSRM); therefore, the backup object and the backup time are two critical parameters for establishing effective checkpoints. In the traditional checkpoint backup scheme, the embedded systems need to back up all the data belonging to the architectural memory components (registers and memories) for the next checkpoint-rollback [28]. This procedure requires continuously accessing various registers for replicating stored elements.

However, this type backup approach will not only occupy a large number of on-chip storage spaces but also result in a high time overhead on checkpoint backup operation to the real-time embedded system. Generally, the registers in a processor can be categorized into instruction set registers and temporary registers according to their function types, where we define the instruction set registers of OpenRISC as special meaning registers with a small number of control registers, such as the instruction register (IR), program counter (PC), general purpose registers (GPR), special purpose registers (SPR), and status registers (SR). In addition, temporary registers are defined as buffer registers during instruction execution, where the results will be saved into the instruction set registers after operation and will not affect the continuation of program.

In the five-stage execution pipeline of the CPU, the sequence of instructions of each BB enters the processor for orderly execution. The timing diagram of the BB instructions executed in the five-stage pipeline processor is shown in Figure 3. In the P1 clock cycle, when the jump instruction enters the ID stage, the instruction (Instr. 3) in front of the jump instruction enters the EX stage, and its regular operation, such as logical comparison or data operation begins to execute.

Then, in the P2 clock cycle, the instruction (Instr. 3) execution stage has completed, its operation results are stored into the instruction set registers, and the temporary registers associated with it will no longer affect the execution of subsequent instructions. The jump instruction enters the EX stage, and some relevant registers will be updated. Finally, in the P3 clock cycle, the instruction monitor will detect a delay slot instruction, which indicates the end of the current BB, and if the relevant registers for the jump instruction execution have completed updates, the integrity signature of the current BB will be verified as early as at this time.

As all the operation results were updated to the instruction set registers in the P3 clock cycle, and the values of all the temporary registers do not affect next BB program execution; we only need to back up the values of the instruction set registers to meet the checkpoint-rollback operation. Therefore, the end of P3 is the best time for the checkpoint backup.

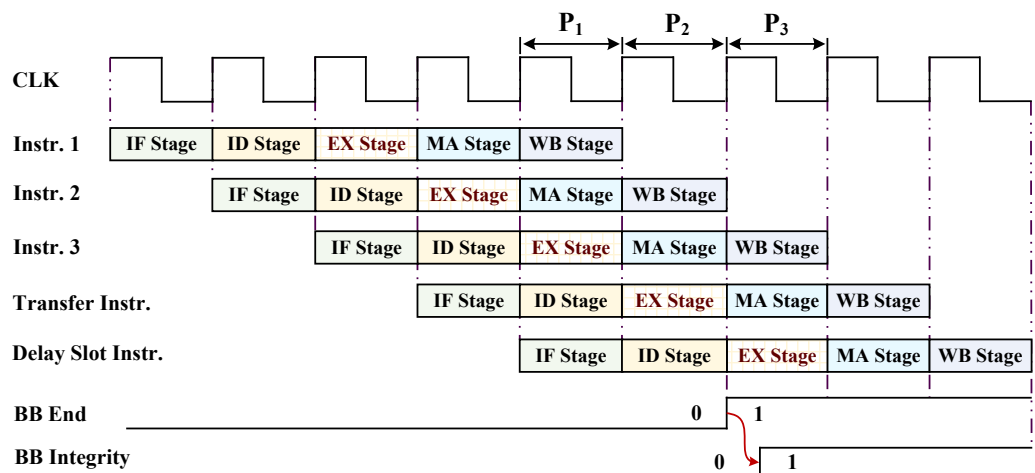


Figure 3. The timing diagram of BB instructions being executed in the five-stage pipeline processor.

3.2.2. Buffer and Backup Registers for Checkpoint Rollback

As BB integrity signature recalculation and reference information search require several clock cycles, the integrity verification of each BB will take time after the end of the P3 stage. This can cause a wrong situation that when the checkpoint-backup registers (storing the values of a previous successful checkpoint) need to be updated with the values of the current BB instruction set registers at the end of P3. Once the subsequent integrity check is invalid, checkpoint-rollback will revert back to the current BB’s checkpoint with the updated values of registers.

In order to avoid this wrong situation, we configured the two-stage registers of buffer-purpose and backup-purpose registers for the checkpoint-rollback operation. The schematic implementation of two-stage registers for the checkpoint rollback operation is shown in Figure 4. When the processor executing instructions and each BB’s integrity signature check are successful, the CPU checkpoint-rollback operation are not activated, and the instruction set registers are updated according to the current instruction execution.

At the end of the current BB, the values of instruction set registers are first buffered into buffer-purpose registers, and the values in backup-purpose registers maintain unchanged until the BB integrity signature is verified successfully. If the current BB integrity verification is invalid, the buffered values of the buffer-purpose registers will not be backed up into backup-purpose registers and will be overwritten at the next BB integrity checking pass. The CPU checkpoint-rollback operation is activated, the values of the previously-saved checkpoint in backup-purpose registers will be restored into the instruction set registers to resume the normal instruction execution of the current BB.

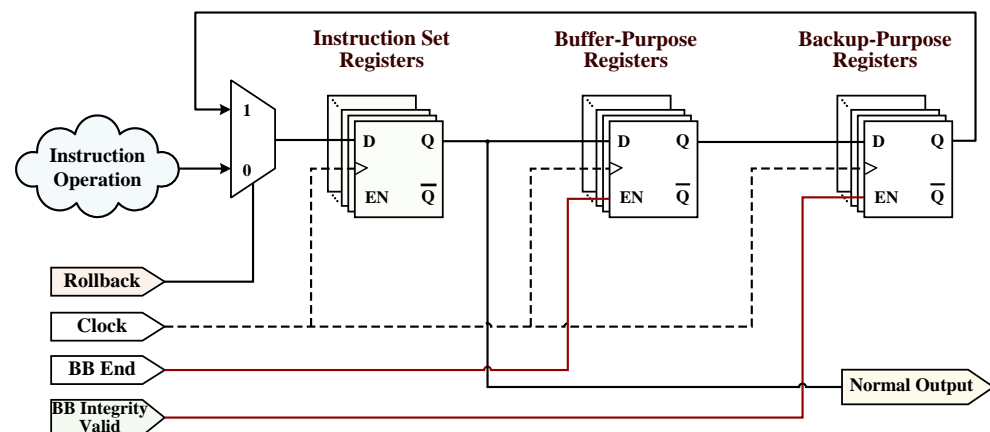


Figure 4. The schematic implementation of two-stage registers for checkpoint-rollback operation.

Applying the above two-stage registers to the CPU’s checkpoint-rollback operation is based on the contingency of instruction tampering attacks induced by hardware Trojans. Once an instruction is tampered with in program execution, it inevitably destroys the integrity signature of BB, and the checkpoint-rollback operation will revert the processor back to the initial instruction of the current BB with all saved checkpoint data for BB re-execution. In order to expatiate the proposed checkpoint-rollback operation based on two-stage registers, the specific example of an instruction stream tampering attack aiming at the benchmark of OpenECC is shown in Figure 5.

Firstly, the BB_{i-1} is already verified with the integrity of the instructions at the end of the delay slot instruction. The values of the instruction set registers are backed up into the backup-purpose registers at the checkpoint. Secondly, the CPU begins to execute BB_i , the values of instruction set registers will be buffered into the buffer-purpose registers at the end of BB_i for the waiting BB integrity verification and thus will not interfere with the execution of the following instructions of the next BB in writing the instruction set registers. Once the BB_i integrity is valid, the buffered values in buffer-purpose registers will be backed up into the backup-purpose registers as new checkpoint data.

On the other hand, if the BB_i integrity is invalid, we assumed the branch transfer instruction *lbf 1f744* was tampered with by an attack to a wrong target branch address of *lbf 1f730*, the values of buffer-purpose registers will be overwritten in the next BB execution. Finally, the checkpoint rollback is activated, and the values of backup-purpose registers are restored into CPU instruction set registers, and the CPU re-fetches the BB_i instruction order from the instruction memory for BB_i re-execution.

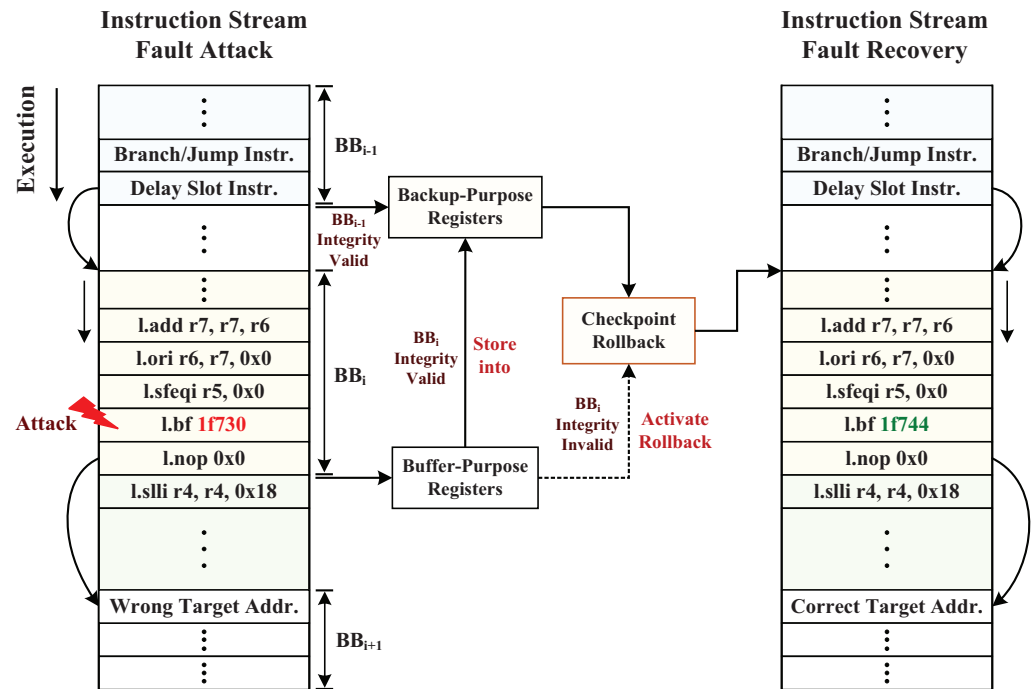


Figure 5. Example of instruction tampering attack and fault recovery aim at the benchmark of OpenECC.

4. Hardware Implementation of CPU–SPU Collaborative SoC

This section presents the hardware implementation details of the CPU–SPU collaborative SoC. We expatiate the hardware implementations of the instruction monitor and FSRM. In addition, the performance optimization methods will be implemented to reduce the performance overhead of the SPU.

4.1. Hardware Implementation of Instruction Monitor

After we complete the offline preparation works of the program BB partition and the BB’s INF_{ref} extraction from the instruction codes (InsCodes), our following phase is the hardware implementation of the instruction monitor to provide a high-efficiency instruction violation detection relying on the proposed fine-grained BB integrity monitoring. As the instruction codes will be executed in the five-stage pipeline processor in a sequential order, the instruction monitor will verify the BB integrity signatures according to the sequential order of program execution. The hardware architecture details of the proposed instruction monitor connecting with the CPU are illustrated in Figure 6.

The CPU exports the physical addresses of instructions from the PC, the instruction codes from the ID stage, and the delayed tags from a synchronizer logic, respectively. These three parameters are the input signals of our hardware-implemented instruction monitor. Then, a finite state machine (FSM) IP is configured to keep track of the branch and jump instruction executions with their control-state transitions, which can also identify the BB boundary of the start address (also being the target address of previous transfer instructions) and end address with a delay slot instruction after the branch or jump operation.

Under the control of FSM IP, when the start address (ADD_{start}) of each BB is input into the LHash engine as an activated signal, the InsCode streams will be continuously

pumped into the LHash engine to recalculate a 96-bit LHash digest after following the BB’s InsCodes executions. Moreover, we provide a selection logic of 16-bit RNG bit-selected numbers to generate the 16-bit LHash digest (DIG_{cal}).

In another path, an additional monitor cache (M-Cache) is configured to buffer parts of INF_{ref} blocks from the monitor memory. The instruction monitor searches the cache lines of M-Cache according to the received ADD_{start} of each BB. If the M-Cache hits, the corresponding INF_{ref} block is input an intercept logic for obtaining the 16-bit BB integrity digest of $INF_{ref}[15:0]$ as the golden LHash (DIG_{lhash}); if M-Cache misses, the instruction monitor starts to search the ADD_{start} in the monitor memory. If it succeeds, a two-input multiplexer (MUX) controlled by the states of hit/miss receives the $INF_{ref}[15:0]$ after the intercept logic; if it fails to search, the monitor asserts an invalid signal of BB absence to the processor.

When M-Cache hits or memory hits, the recalculated DIG_{cal} will be compared with the prestored DIG_{lhash} in the integrity checker. The instruction monitor asserts the BB as a valid status when their compared result is equal. Otherwise, the instruction monitor asserts the BB as an invalid status, and we preset the LHash value error with the invalid status “01”, and the start address error with the invalid status “10” (BB absence).

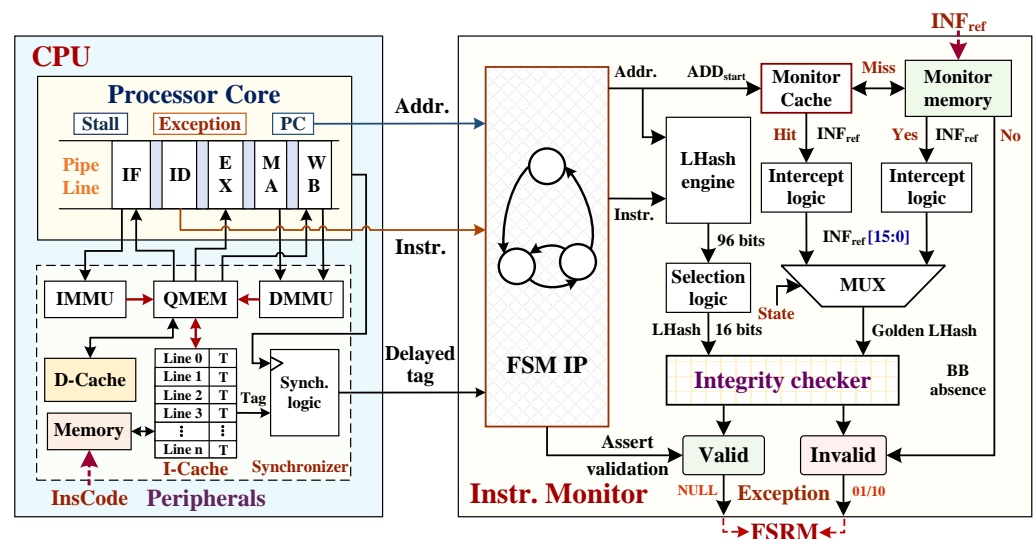


Figure 6. The hardware implementation details of the instruction monitor connecting with the CPU.

Ultimately, the BB’s invalid signal will be sent not only to the Exception module of the CPU when the instruction monitor detects a violation of BB integrity but also to the FSRM in the SPU to activate checkpoint-rollback operation, and hold the CPU_Stall signal to suspend the execution pipeline. In the CPU–SPU collaborative SoC architecture, the exception signal is nonmaskable to trigger both a fast-response and the self-repairing mechanisms of the CPU and FSRM. While FSRM repairs the instruction fault, the rollback recovery logic still asserts the CPU_Stall signal until the successful checkpoint-rollback operation. It is noteworthy that the above delayed tag signal is configured to improve the checking efficiency of BB integrity; it helps to reduce the performance overhead of the instruction monitor.

4.2. Hardware Implementation of the Fault Self-Repairing Module

As previously described, the instruction fault self-repairing operation of the FSRM via the checkpoint-rollback mechanism is tightly coupled with the BB integrity checking mechanism of the instruction monitor. The overall control status diagram of the SPU real-time instruction monitoring and checkpoint-rollback operation is illustrated in Figure 7. When the instruction monitor detects a delay-slot instruction following transfer instruction into the execution of the CPU five-stage pipeline, the status of BB end triggers the

checkpoint-backup operation, which duplicates the values of instruction set registers into buffer-purpose registers.

Then, the instruction monitor asserts the BB valid status when the BB integrity label verification has passed; otherwise, it asserts the BB invalid status. At the status of BB valid, a new checkpoint begins to be established, which backs up the values of buffer-purpose registers into the backup-purpose registers. After checking that the status is BB invalid, the checkpoint recovery is activated, and all the values of backup-purpose registers are restored into the instruction set registers. When the register value restoration has completed, the CPU_Stall signal will be released and the CPU resumes the first instruction execution of the current BB. This two-stage checkpoint-rollback can minimize the time consumption required for instruction fault recovery.

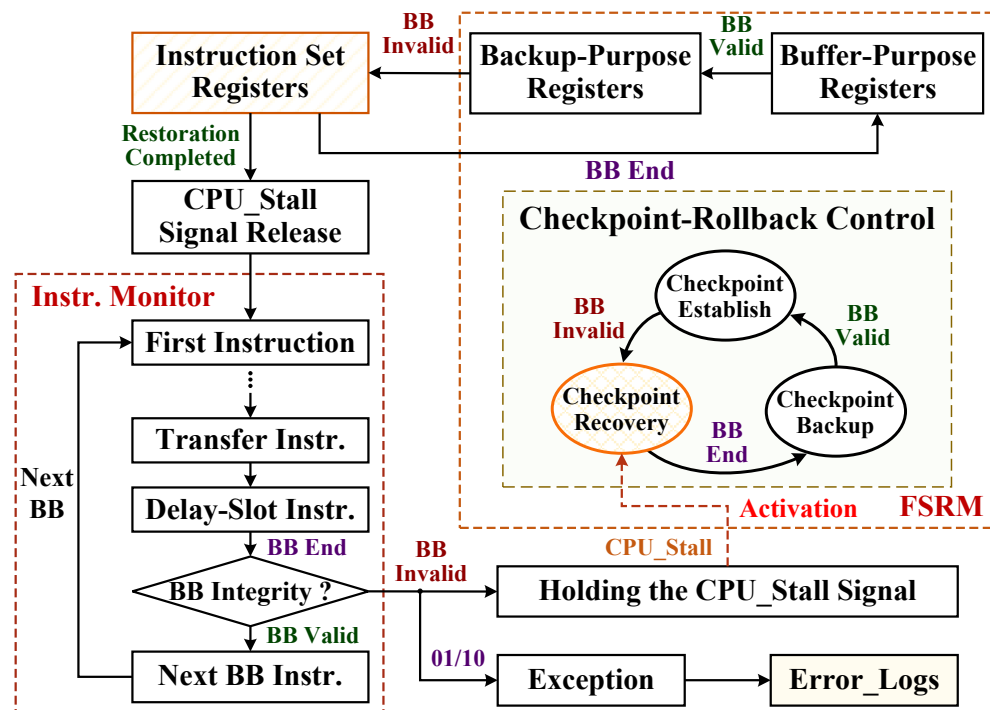


Figure 7. The overall control status diagram of instruction monitoring and checkpoint-rollback.

The hardware implementation of two-stage registers for the checkpoint-rollback operation is presented in Figure 8, which also is a specific hardware implementation detail of Figure 4. We additionally configured the proposed two-stage registers for buffering and backing up all the values of the instruction set registers. In each stage, thirty-two 32-bit registers are configured to back up all the values of the instruction set registers (including the IR register).

In some practical applications of the OpenRISC processor, only twenty-six instruction set registers are actually utilized in the realistic checkpoint-backup operation. Increasing register configurations of the thirty-two 32-bit registers will fully meet the requirements of future more-complicated checkpoint-rollback operations in different application platforms. For a better clarification, the implementation flow of the program execution monitoring and instruction fault self-repairing is presented in Algorithm 1.

It should be considered that the CPU_Stall signal will suspend the execution pipeline of instruction when the BB's integrity is invalid, and then SoC enters the instruction fault self-repairing phase, which will make the performance overhead of FSRM impossible to be calculated by the performance indicator of Cycles-Per-Instruction (CPI). Therefore, the checkpoint-rollback operation as the post-detection technique will not be considered in the next performance overhead evaluation.

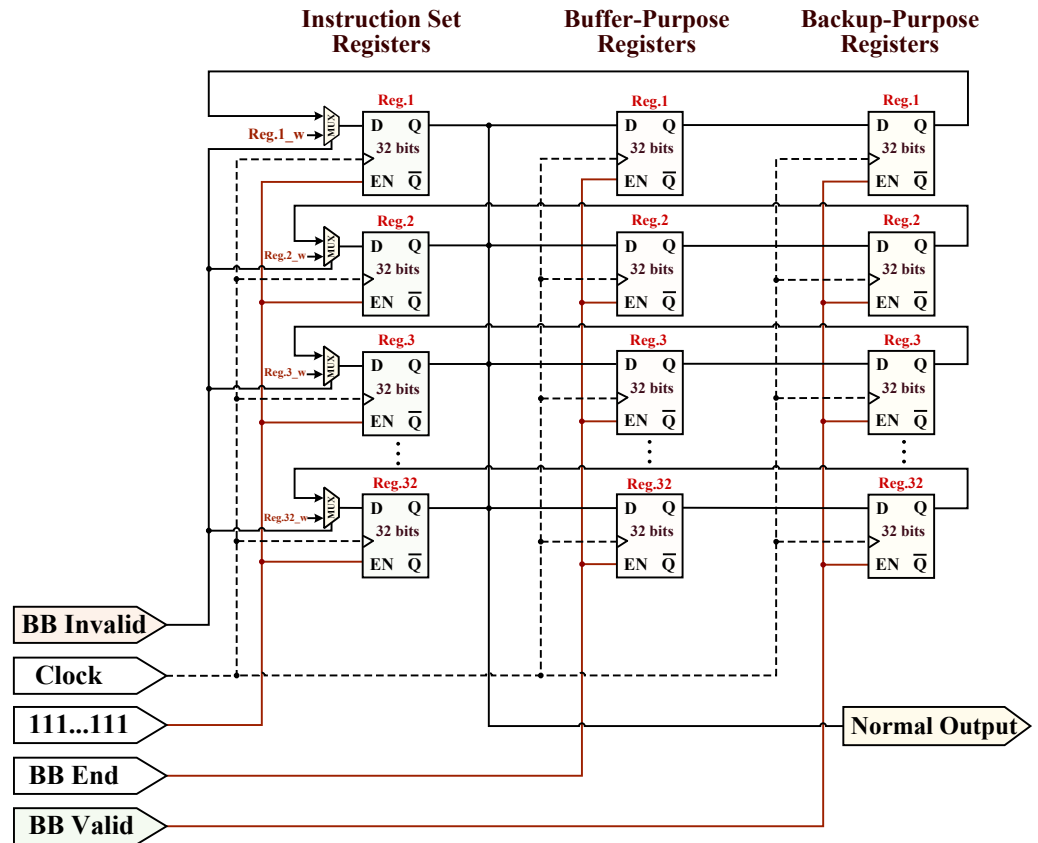


Figure 8. The hardware implementation of two-stage registers for checkpoint rollback.

Algorithm 1 Implementation flow of the program execution monitoring and instruction fault self-repairing.

Inputs: ADD_{start} , Instruction

Outputs: Exception, BB End, BB Valid, BB Invalid

- 1: $BB_i \leftarrow$ set of the program basic blocks, BB_i , $1 \leq i \leq m$, where m denotes total of BBs.
- 2: $Instruction_j \leftarrow$ set of instructions in each BB, $Instruction_j$, $1 \leq j \leq n$, where n denotes total number of instructions in each BB.
- 3: **begin:** Program execution, SPU BB integrity monitoring and fault self-repairing;
- 4: **for all** BB_i ($i = 1; i++; i \leq m$) **do**
- 5: **if** ADD_{start} of BB_i is detected **then**
- 6: **for all** $instruction_j$ ($j = 1; j++; j \leq n$) **do**
- 7: $DIG_{cal} = f_{LHash}(instruction_1, instruction_2, \dots, instruction_j)[16\text{-bit}]$;
- 8: $DIG_{lhash} = INF_{ref}[15:0]$ of BB_i from monitor memory;
- 9: **end for**
- 10: **if** $DIG_{cal} = DIG_{lhash}$ **then**
- 11: Exception = null ("00"); /* BB Valid */
- 12: **else** Exception = assertion ("01"/"10"); /* BB Invalid */
- 13: **end for**
- 14: **if** BB End **then**
- 15: Checkpoint backup: Buffer-purpose registers \leftarrow Instruction set registers;
- 16: **else if** BB Valid **then**
- 17: Checkpoint establish: Backup-purpose registers \leftarrow Buffer-purpose registers;
- 18: **else** BB Invalid **then**
- 19: Checkpoint recovery: Instruction set registers \leftarrow Backup-purpose registers;
- 20: **end**

4.3. Performance Optimizations of SPU Security Monitoring

During program execution, the performance overhead induced by the BB integrity verification is an important consideration in the hardware-enhanced architecture design. As previously described, the integrity signature recalculation and reference information search require some clock cycles. This is the reason for the two-stage register configuration, as it is possible that the comparative result of the BB integrity verification is not yet asserted after all the instructions of the BB being executed. This situation requires the CPU execution pipeline to be suspended every time before each BB integrity checking pass, which will cause the CPU to have a large performance overhead. Therefore, we configured the M-Cache to improve the searching efficiency of the INF_{ref} block, and an optimized I-Cache to reduce the number of times in the BBs integrity verification. In addition, the worst situation regarding the maximum performance overhead in completing BB integrity checking is depicted.

4.3.1. M-Cache Searching Efficiency

During program execution, the determining factor of performance impact in the proposed instruction monitor is the searching efficiency of INF_{ref} blocks. When the instructions of each BB enter the CPU execution pipeline in sequence, if there is no M-Cache, the instruction monitor needs to search the corresponding INF_{ref} block from the monitor memory according to the ADD_{start} of each BB, which will frequently access the monitor memory and search all the table of INF_{ref} blocks. In order to improve the searching efficiency of BB INF_{ref} blocks, the M-Cache with a depth of 256 cache lines is configured to buffer partial blocks of the BBs INF_{ref} table from monitor memory.

Hence, both the temporal locality principle and spatial locality principle of the M-Cache in searching nearby BB INF_{ref} blocks can be fully exploited to reduce the memory's searching time. The M-Cache content-searching method with a pointer and cache internal structure with partial BB INF_{ref} table are shown in Figure 9. The pointer is described as a double ring buffer that is constructed with an 8-bit register to indicate the address of the appointed cache line.

The content-searching circuit and the storage parts of ADD_{start} are fully interconnected; therefore, the hit/miss status of M-Cache can be acquired within two clock cycles. As the content searching method is according to the effective ADD_{start} from the PC, the cache line is appointed by comparing the BB start address segment of $ADD_{start}[17:2]$ with the value of cache line $INF_{ref}[31:16]$.

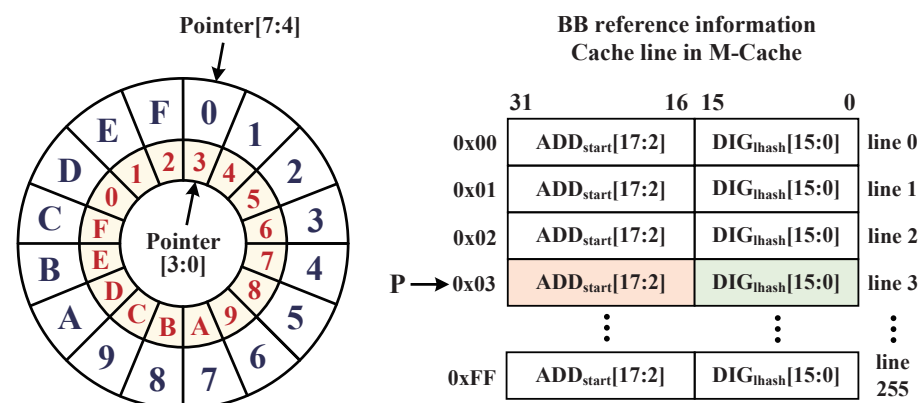


Figure 9. The M-Cache searching method and internal structure of the BB reference information table.

4.3.2. I-Cache Structure Optimization

In the SoC architecture, improving the searching efficiency of the INF_{ref} block is not the only strategy in performance overhead optimizations. Moreover, reducing the number of times on BBs integrity checking is another approach to decrease the performance overhead caused by LHash recalculation and searching the INF_{ref} table. In order to better

utilize the locality principle of I-Cache directly mapping main memory (InsCode segment), we tagged the instructions that had been cached in I-Cache and validated for integrity signature during the other BB executions as security statuses.

Figure 10 presents the internal implementation mechanism of the I-Cache with a size of 8 KB. The structure consists of two parts of IC_TAG and IC_RAM. When CPU sends the request 32-bit physical address (ADD_{phy}) to the I-Cache for fetching an instruction, then, the I-Cache searches the high 19-bit $ADD_{phy}[31:13]$ to identify whether it is within its address space range, while the indexed cache line is appointed by $ADD_{phy}[12:4]$ (where the cache-line depth is 512).

Afterwards, the physical address can accurately find the target address of the request instruction according to line block offset address of $ADD_{phy}[3:0]$. The *Validity (V)* mark bit in the appointed cache line is “1”, which indicates that the I-Cache hit. If the I-Cache misses, then CPU fetches instruction via accessing the external InsCode memory.

As shown in the IC_RAM of I-Cache structure, a cache line has four instruction words, and when the four instructions at the same cache line are read for execution, the *Tag (T)* bit in the cache line turns from “0” to “1” to indicate that the instructions in current cache line were verified for integrity. From the partition principle of program BBs, one BB contains at least three instructions and occupies one or two cache lines. The long BB occupies several cache lines.

Therefore, the I-Cache outputs the tag signal of security when all the instructions of the current BB are cached in I-Cache, also the tags of cache lines they occupied are all signed with “1” (for logic AND). Then, the delayed Tag from a synchronizer is input into the FSM controller, and the instruction monitor directly asserts the valid status to the processor (as shown in Figure 6). This optimized approach plays an important role in reducing the number of BB integrity checks in the situation that the BB overlaps with the other BBs.

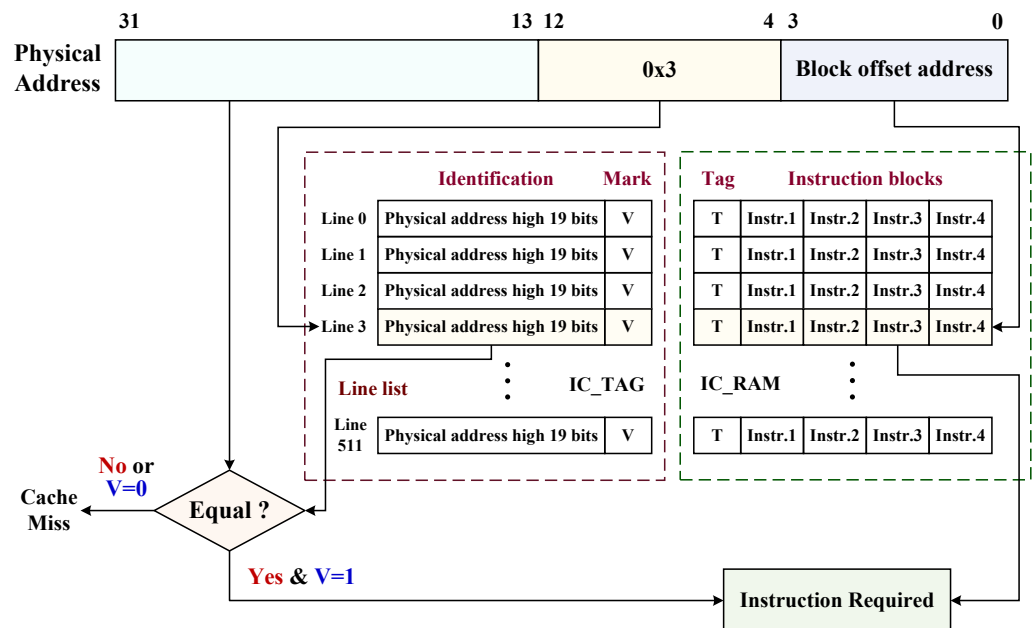


Figure 10. The internal implementation mechanism of the I-Cache with the size of 8 KB.

4.3.3. Worst Situation Depiction

In the above-mentioned optimizations of M-Cache and I-Cache, an important consideration is the clock timing overhead or conflict induced by SPU, which is induced by instruction monitoring and instruction fault recovery. The worst situation of performance overhead occurs when M-Cache and I-Cache both fail to contribute to the integrity verification for the current BB, and the instruction monitor requires searching the BB's IN_{ref} block in the entire monitor memory.

Moreover, the FSRM will further perform the instruction fault recovery once the BB integrity check is invalid. Figure 11 depicts the timing diagram of one BB execution with integrity validation and instruction recovery in the worst situation. The period of T1 represents the total time consumption for searching the BB INF_{ref} block in both M-Cache and monitor memory from a new BB being detected, and its search result can be obtained with a high probability before the recalculated results of the LHash engine.

The period of T2 represents the time consumption of the golden LHash DIG_{lhash} being obtained and waiting for verification. Period T3 indicates that the integrity checker completes the comparison and outputs the validation status within one clock cycle. Since the searching process of INF_{ref} relying on the ADD_{start} of each BB in the M-Cache and monitor memory is simultaneous with instruction executions, it can minimize the performance overhead of the BB integrity checking; thus, the time consumption on searching INF_{ref} in the M-Cache and monitor memory are both acceptable for integrity validation.

In the instruction recovery phase when the BB integrity verification has failed, the activated checkpoint recovery signal closely follows the status of the BB integrity validation. It is later only three clock cycles from the end of the current BB boundary to CPU_Stall signal suspension, it is indicated that the BB recovery will not cause potential clock timing conflicts even in executing the shortest BB (only three instructions).

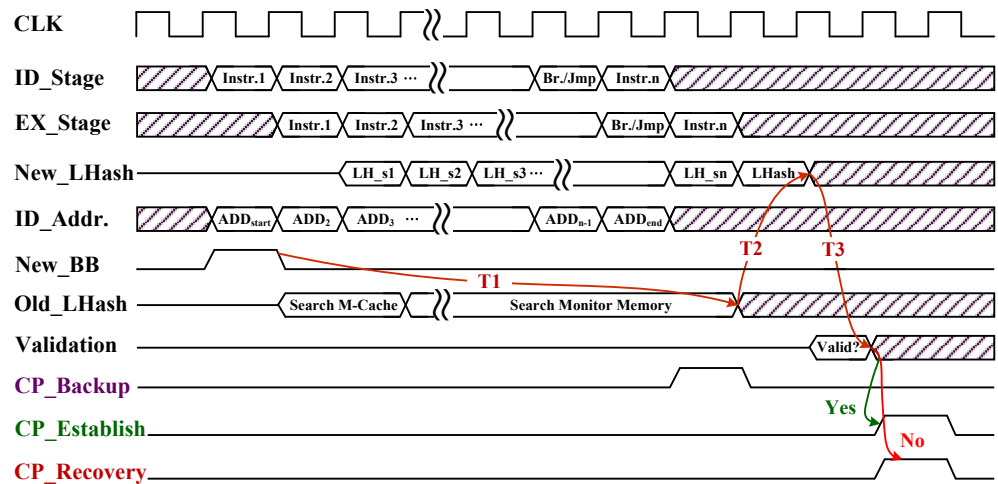


Figure 11. The timing diagram of one BB execution with integrity validation and instruction recovery in the worst situation.

5. Experiments and Results

This section presents the experiments and results of the CPU-SPU collaborative SoC, and expatiates the proposed SPU in terms of the performance overhead and security capability. This section contains the following three parts: the experimental setup for the program execution; the performance overhead evaluation; and the security capability evaluation.

5.1. Experimental Setup

We integrated the hardware-implemented SPU into the scalar OpenRISC processor system of OR1200 for constructing the CPU-SPU collaborative SoC for real-time program execution monitoring and instruction fault self-repairing. The main frequency of the CPU core is set as @100 MHz, and the internal clock signal of integrated SPU satisfies the synchronization with the processor. The hardware configurations of I-Cache and D-Cache support the different sizes of 2, 4, 8, and 16 KB.

We first configured the processor with a typical depth size of 8-KB I-Cache and 8-KB D-Cache, in which the internal structures consisted of the 512 cache line blocks. We developed the embedded system in Verilog hardware description language (HDL), performed the logic synthesis and implementation in Xilinx ISE Design Suite 14.7, and completed the hardware implementation on the Xilinx Virtex-5 FPGA development board. The GNU Cross

Compilation Toolchain or32-elf-gcc matching with the OR1200 instruction set architecture (ISA) was utilized to generate InsCodes.

Moreover, we configured some necessary controllers for some off-chip peripherals, such as the DDR2 SDRAM, parallel flash, serial ports, and Ethernet. In the system initialization stage (Boot Process), the SoC bitstream is programmed from the flash memory onto FPGA at power-up; then, the bootloader (U-Boot) boots a Linux kernel to mount the root file system for execution. We defaulted that the SPU monitors the program execution and recovers the instruction fault during the whole application life cycle.

5.2. Performance Overhead Evaluation

While the integrated SPU monitors the program execution in real-time, this inevitably causes some performance overhead to the embedded system. Although we adopted some architectural optimizations in both M-Cache and I-Cache to reduce the performance overhead of CPU–SPU collaborative SoC, it is still necessary to evaluate the final performance overhead induced by the SPU as an indicator of its applied practicability. In our evaluations of the SPU performance overhead, ten various benchmarking programs were selected from Mibench suite [29] to simulate real embedded application scenarios. These ten benchmarks were developed based on the market-proven industry-standard EEMBC-CoreMark; they are comprehensive and advanced performance benchmarks in academia, industry, and military applications.

First, the selected ten benchmarks are preprocessed under offline analysis and static extraction via running Perl scripts to generate the INF_{ref} blocks of BBs; then, the benchmarks are compiled by the GNU Cross Toolchain or32-elf-gcc and downloaded into FPGA for program execution, respectively. Furthermore, the numbers of total instructions and all the BBs of each benchmark are counted, and the INF_{ref} table required on-chip storage space is also calculated.

Considering that the hit rates of I-Cache and M-Cache could influence the SPU performance overhead, we used the or1ksim [30] simulation software to record the hit rates of the I-Cache and M-Cache, respectively. Hence, the SPU average performance overheads under the different benchmarks can be calculated according to the indicator of CPI on the SoC with and without integrating the SPU.

5.2.1. SPU Performance Overhead

The performance overhead of the CPU–SPU collaborative SoC is shown in Table 2. In the selected benchmarks, OpenECC has the largest numbers of instructions and BBs, and its INF_{ref} table for integrity monitoring occupies the maximum on-chip storage space of 26.30 KB in the monitor memory accordingly. We also found that the average hit rates of I-Cache and M-Cache configured with 8 KB both exceeded 98%, and even the hit rate of the M-Cache reached 99%. The primary reason is that the BBs INF_{ref} blocks were stored into monitor memory by the order of BBs execution.

Their high-hit rates contributed by both temporal locality principle and spatial locality principle can effectively maintain the SPU with a low performance overhead. The benchmark of quicksort had the highest M-Cache hit rate (99.83%); its small number of program BBs determined both a lower number on BB overlaps and a higher proportion in caching INF_{ref} blocks from the monitor memory. The higher proportion in caching INF_{ref} blocks from the M-Cache can avoid frequent access to monitor memory; therefore, it had the lowest performance overhead (0.43%).

The indicator CPI tended to increase with the number of benchmark instructions, which indicates that the probability of M-Cache missing increases. When the overlapped BBs occupied a larger proportion in the program partition, that caused INF_{ref} block discontinuity in the monitor memory.

For example, running the benchmarks of OpenECC and basicmath requires searching the INF_{ref} table cached in M-Cache. A small minority of discontinuous INF_{ref} blocks are not cached in the current M-Cache at the time of executing the jump instructions, and the

monitor memory is accessed in response; thus, their CPI values are higher than the other benchmarks. After we complete the calculations of the experimental data from these ten benchmarks, the average performance overhead induced by SPU was 1.92%, ranging from 0.43% (quicksort) to 3.36% (OpenECC).

Table 2. Performance overhead of the CPU–SPU collaborative SoC (8-KB I-Cache, 8-KB D-Cache, and 8-KB M-Cache).

Benchmark	Total Instruction	Total BB	Memory Size (KB)	I-Cache Hit	M-Cache Hit	CPI without SPU	CPI with SPU	Performance Overhead
AES	22,170	3535	13.81	99.26%	99.35%	3.528	3.636	3.06%
basicmath	26,515	4327	16.90	98.13%	98.97%	2.643	2.712	2.61%
bitcount	19,684	3344	13.06	98.29%	99.40%	1.653	1.680	1.63%
blowfish	19,128	3247	12.68	97.95%	99.56%	3.536	3.597	1.72%
CRC	18,941	3231	12.62	99.53%	99.58%	1.723	1.748	1.45%
FFT	13,506	2143	8.37	96.58%	99.71%	2.943	2.979	1.22%
OpenECC	56,313	6734	26.30	98.71%	98.52%	3.184	3.291	3.36%
patricia	23,130	3853	15.05	97.90%	98.14%	1.649	1.685	2.18%
quicksort	6707	1018	3.98	99.72%	99.83%	1.856	1.864	0.43%
SHA1	20,455	3400	13.28	98.65%	98.27%	2.351	2.388	1.57%
Average	–	–	13.61	98.47%	99.13%	2.507	2.558	1.92%

5.2.2. Optimized Effects of M-Cache

In order to further explore the effects of M-Cache in reducing the performance overhead of the SPU, we made experimental statistics to evaluate the indicator of CPI under different depths of M-Cache while keeping the 8-KB I/D-Cache unchanged. Since the M-Cache hit rate was closely related to its size, the depths of M-Cache were configured with different sizes of no M-Cache: 16, 32, 64, 128, and 256. Table 3 presents the performance overhead of the SPU configured with different depths of M-Cache.

These experimental results indicate that the performance indicators of CPIs are constantly decreasing with the raising in depths of the M-Cache, however, the CPI reduction trend begins to slow down when the M-Cache hit rate reaches a saturation, such as depth (128) and depth (256). In fact, depth (128) and depth (256) are both suitable for M-Cache configuration. We selected depth (256) to obtain the lowest performance overhead of 1.92%, which is a suitable depth for SPU to achieve a good tradeoff between storage space and performance overhead.

Table 3. Performance overhead of the SPU configured with different depths of M-Cache (8 KB I-Cache and 8 KB D-Cache).

Benchmark	CPI without SPU	CPI with SPU under the Different Depths of M-Cache					
		No M-Cache	Depth (16)	Depth (32)	Depth (64)	Depth (128)	Depth (256)
AES	3.528	4.986	4.217	3.862	3.736	3.642	3.636
basicmath	2.643	3.935	3.120	2.839	2.755	2.716	2.712
bitcount	1.653	2.362	1.964	1.796	1.713	1.685	1.680
blowfish	3.536	4.968	4.353	3.895	3.742	3.608	3.597
CRC	1.723	2.180	2.019	1.954	1.816	1.754	1.748
FFT	2.943	4.891	4.162	3.583	3.214	2.986	2.979
OpenECC	3.184	4.764	3.975	3.482	3.381	3.302	3.291
patricia	1.649	2.336	1.986	1.816	1.712	1.690	1.685
quicksort	1.856	2.637	2.179	1.970	1.887	1.867	1.864
SHA1	2.351	3.674	2.662	2.539	2.436	2.392	2.388
Average	2.507	3.673	3.064	2.774	2.639	2.564	2.558
Performance overhead	–	46.52%	22.21%	10.63%	5.27%	2.28%	1.92%

5.2.3. Optimized Effects of I-Cache

As described above, the instruction monitor is also closely related to the hit rate of the I-Cache. In order to further explore the effects of the I-Cache hit rate on the performance overhead of SPU, we continued the benchmark evaluation experiments by maintaining the 8-KB D-Cache and M-Cache unchanged, and the size of I-Cache is reconfigured as 2, 4, and 16 KB, respectively. The performance overheads of ten selected benchmarks under different sizes of I-Cache are shown in Figure 12. The performance overhead incurred by the SPU decreased with the enlargement of I-Cache addressing space as well as the I-Cache hit rate improvement.

Moreover, the space enlargement of I-Cache helps to reduce the number of times BB integrity verification. When executing the selected ten benchmarks with a 16-KB I-Cache, the performance overhead of the CPU–SPU collaborative SoC had significant reductions compared to I-Cache configured with 2-KB, where the highest performance overhead was 6.87% (OpenECC at 2-KB), and the lowest performance overhead was at 0.38% (quicksort at 16 KB). The mechanism of this trend is that, when the I-Cache hit rate increases, the number of times the CPU fetches instructions from the external memory decreases.

Considering the applied practicality of the SPU in transplantation with reasonable resource consumption and low hardware complexity, we finally selected the 8-KB I-Cache to reach a good balance between the performance overhead and resource consumption; in addition, the size remains the same with the D-Cache and M-Cache depths, and it is easily configured in other instruction set architecture platforms.

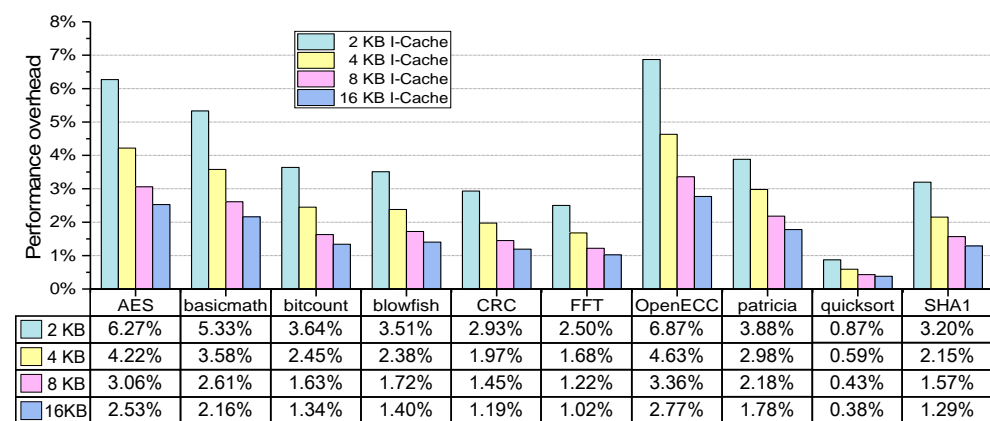


Figure 12. Performance overheads of ten selected benchmarks under the different sizes of I-Cache.

5.3. Security Capability Evaluation

An important consideration of the CPU–SPU collaborative SoC is its security capability. In order to validate the effectiveness of the proposed SPU on program execution monitoring and fault self-repairing, we utilized the OpenRISC debugging system of OR1K to observe the program execution, instruction modification, exception output, and interrupt location. This debugging system consists of the Joint Test Action Group (JTAG)-TAP module and the Advanced debug interface (ADI), which acts as an interface to directly communicate with the processor core and Wishbone system bus so that we can start and break the execution pipelines of programs and read or write processor internal registers by accessing the CPU.

We performed instruction tampering evaluations aimed at both nontransfer instruction tampering and transfer instruction tampering from the different locations of processor core, I-Cache, flash memory, and source program code. Conveniently, we also selected the partial instructions from the instruction stream of benchmark OpenECC (as shown in Figure 2) as an example, in which, the nontransfer instruction *l.nop 0x0* was artificially tampered with as *l.nop 0x1* and the branch transfer instruction *l.bf 1f730* was modified to the different branch address of *l.bf 1f734*. Table 4 presents the security capability evaluations of the CPU–SPU collaborative SoC under different instruction tampering attacks.

The exception binary results from serial port printing were analyzed and displayed in the upper computer, and their corresponding error_log files were generated for examination. Ultimately, the instruction fault recovery was completed within three clock cycles after the delay slot instruction execution.

Table 4. Security capability evaluations of the CPU–SPU collaborative SoC under different instruction tampering attacks.

Attack Methods	Nontransfer Instruction Tampering				Transfer Instruction Tampering			
	Processor	I-Cache	Memory	Program	Processor	I-Cache	Memory	Program
Locations								
Approaches	Instruction tampering: l.nop 0x0 ⇒ l.nop 0x1				Instruction tampering: l.bf 1f730 ⇒ l.bf 1f734			
InsCodes	Binary code tampering: 15000000 ⇒ 15000001				Binary code tampering: 13ff fffd ⇒ 14000001			
Exception	LHash Error (“01”)				LHash Error (“01”) & BB Absence (“10”)			
Fault Recovery	3 clock cycles after delay slot instruction execution				3 clock cycles after delay slot instruction execution			

While the CPU executed the programs of selected benchmarks on the FPGA development platform and printed the control scripts of exception statuses into the upper computer, we can track the embedded system for a malicious attack as shown in Figure 13. The integrity signature validations for the above binary instruction codes at the granularity of BBs can recognize any instruction tampering behaviors in transfer and nontransfer instructions. When the adversary tampered with the nontransfer instruction, the SPU asserted a BB LHash verification error and reported the corresponding BB integrity recalculated value and its correct LHash digest. There was only exception error (“01”) for nontransfer instruction tampering when the BB integrity checking failed.

In addition, there were two invalid statuses for the transfer instruction tampering: the SPU first asserted the exception error (“01”) when the current BB integrity checking failed, and then another BB absence (“10”) was reported when the next BB start address (target address of branch/jump) searching missed. We can benefit from the error_logs of both the nontransfer and transfer instruction attacks regarding the BB LHash recognition.

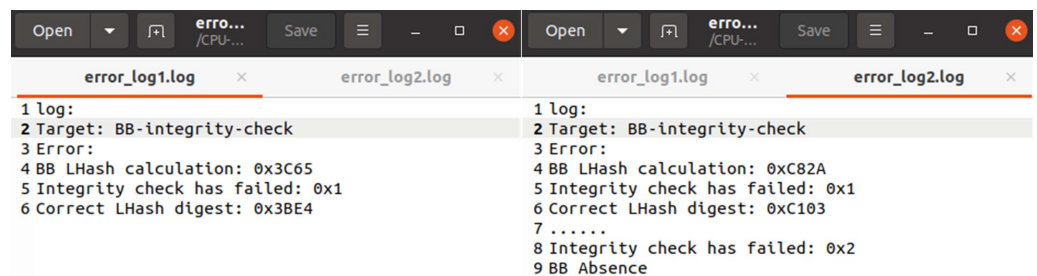


Figure 13. The output error_log files for reporting the instruction tampering behaviors in BBs.

From the theory of probability, the anticollision capability of the LHash engine integrity signature for each BB in the hardware-implemented SPU can be represented as follows.

$$P(m, n) = \frac{1}{C(m, n) \times 2^n} \tag{1}$$

where $P(m, n)$ denotes the success probability for adversaries to correctly guess the integrity signature of each BB, in which m represents the initial digest size of LHash algorithm, and n represents the length of RNG-selected LHash bits from the m digest size. In our CPU–SPU design, the success probability for an adversary to reversely derive BB integrity signature is $\frac{1}{C(96, 16) \times 2^{16}}$, which makes it impossible for the adversary to actualize instruction tampering attacks without being detected during the period of each BB execution. Hence, the LHash integrity signature has a good robustness in resisting instruction tampering attacks.

5.4. Comparison of Security and Practicality

In the previous hardware-enhanced architectures, security and practicality are two most important metrics to evaluate the superiority of protection techniques. The extended SPU was implemented without any modification on corresponding compiler and processor, and thus it is easily transplanted to other embedded processors with different ISAs.

Moreover, the fully hardware-implemented SPU has a high processing performance in monitoring program execution and calculating the LHash digest; therefore, it can keep a low performance overhead to keep up with the CPU's execution pipeline. Therefore, we evaluated the SPU practicality with ISA extension, compiler modification, and performance overhead. The SPU security capability was evaluated based on the protection abilities of attack detection and fault self-repairing. We divided the system security capability into the following three levels.

- Level-I: Only monitor instruction tampering behaviors by a coarse-grained partition.
- Level-II: Only monitor instruction tampering behaviors by a fine-grained partition.
- Level-III: Both monitor instruction tampering behaviors and instruction fault recovery.

The different hardware security mechanisms reported in the literature in terms of security capability and comprehensive practicality are compared in Table 5. Our integrated SPU achieves both instruction tampering detection and instruction fault self-repairing at a low performance overhead of (1.92%), which is a lower value compared with the state-of-the-art security techniques while providing a comprehensive protection feature. In addition, its fine-grained BB integrity monitoring is not necessary to extend ISAs and modify compilers; these enable the SPU to be transplanted into other ISA hardware platforms.

The fine-grained BB integrity checking can provide a sensitive violation detection for instruction tampering behaviors, and it does not incur significant performance overhead or speed degradation in each BB integrity verification. Therefore, we evaluate the security capability of SPU into the level-III, and it has a better practicality.

Table 5. Comparison of different security mechanisms in terms of security capability and comprehensive practicality.

Security Mechanism	Security Capability				Comprehensive Practicality		
	Level	Instruction Tampering	Fault Self-Repairing	Coarse/Fine Grain	ISA Extension	Compiler Modification	Performance Overhead
HAM [31]	I	Yes	No	Coarse	No	No	Medi (5.59%)
CFI-LEA [15]	I	Yes	No	Coarse	Yes	Yes	Low (3.19%)
CCFI [32]	I	Yes	No	Coarse	Yes	Yes	High (52.0%)
CEDA [33]	I	Yes	No	Coarse	Yes	Yes	High (10.5%)
AE-SSS [34]	II	Yes	No	Fine	No	No	Medi (7.70%)
HCIC [35]	II	Yes	No	Fine	No	No	Low (0.95%)
CLR-REV [36]	II	Yes	No	Fine	Yes	Yes	Low (1.87%)
CPU-ASP [37]	II	Yes	No	Fine	No	No	Low (2.52%)
Our SPU	III	Yes	Yes	Fine	No	No	Low (1.92%)

6. Hardware Implementation Evaluation

In addition to causing performance overhead at the processor, the integrated SPU also inevitably increases the original SoC with hardware complexity, resource overhead, and power consumption. The CPU-SPU collaborative SoC was synthesized, implemented, and evaluated on a Xilinx Virtex-5 FPGA development board. In addition, Synopsys back-end design tools of Design Compiler (DC) and IC Compiler (ICC) were utilized to synthesize the SoC into technology-mapped gate-level netlists, and complete automatic placement and routing based on SMIC 130-nm CMOS standard technology library.

Table 6 presents the CPU-SPU collaborative SoC hardware implementation on FPGA and ASIC. According to the resource utilization of FPGA, the hardware-implemented

SPU occupies about 9.83% on the total slices of SoC and consumes a certain amount of on-chip storage resources for integrity checking and checkpoint-rollback operations. On the whole, our integrated SPU is a relatively smaller hardware module compared to the overall CPU–SPU collaborative SoC.

We implemented the LHash engine as a hardware-friendly algorithm, where the Feistel-PG internal permutation structure only requires 817 gate elements (GE), its hardware consumption is less than another lightweight hash implementation of PHOTON [38], where the internal permutation requires 1120 GE. From the ASIC implementation of SoC, the extended SPU occupies 36.8% of the chip area, which is larger than the proportion of 9.83% on FPGA after automatic routing because of more RAM IP library placements, and its dynamic power maintains a low power consumption. Therefore, the proposed SPU reaches a good balance between security capability and hardware overhead.

Table 6. The CPU–SPU collaborative SoC hardware implementation on FPGA and ASIC.

Platform	Resource Utilization	SoC	SPU
FPGA	Slice Registers	2572	958
	Slice LUTs	16,754	2074
	Occupied Slices	6835	672
	BlockRAM/FIFO	57	52
ASIC	Chip Area	3.42 mm ²	1.26 mm ²
	Power Consumption	56.6 mW	7.4 mW

7. Conclusions

Embedded systems applied in safety-critical equipment require a high quality of security to guarantee program execution security, especially for remote-controlled hardware platforms. This paper presents a CPU–SPU collaborative SoC that integrates the proposed SPU to monitor the program execution and instruction fault recovery in real time. The hardware-implemented SPU architecture employs an instruction monitor to verify the BB integrity signature for detecting malicious instruction tampering behaviors caused by hardware Trojans and artificial modification.

The instruction fault self-repairing module (FSRM) was integrated into the SPU to provide a low-cost instruction recovery for BB re-execution. The CPU–SPU collaborative SoC was implemented and validated on the Virtex-5 FPGA development board. The evaluation results after executing different benchmarks indicate that the SPU can provide both high-efficiency instruction execution monitoring and fast instruction fault self-repairing while maintaining a low performance overhead. Its average performance overhead lowered to the 1.92% at typical 8-KB I/D caches.

Moreover, the security capability evaluation and practicality comparison of the SPU confirmed its superiority in detecting instruction tampering behaviors and transplanting different ISA platforms. According to the hardware implementation of SPU, its hardware complexity is acceptable for embedded systems. Ultimately, the SPU satisfies a good balance between security capability and resource consumption.

Author Contributions: Conceptualization, Z.Z. and X.W.; methodology, Z.Z. and X.W.; software, Z.Z. and Q.H.; validation, Z.Z. and D.X.; formal analysis, J.W.; investigation, Z.Z. and Q.H.; resources, J.L.; data curation, Z.Z., J.M. and J.Z.; writing—original draft preparation, Z.Z.; writing—review and editing, Z.Z.; supervision, X.W.; project administration, X.W.; and funding acquisition, X.W. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China (Grants No. 60973106 and No. 81571142), the Key Project of National Natural Science Foundation of China (Grant No. 61232009), and the Open Foundation of Space-Trusted Computing and Electronic Information Technology Laboratory under grant OBCandETL-2019-02.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Waszecki, P.; Mundhenk, P.; Steinhorst, S.; Lukaszewicz, M.; Karri R.; Chakraborty, S. Automotive electrical and electronic architecture security via distributed in-vehicle traffic monitoring. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2017**, *36*, 1790–1803. [[CrossRef](#)]
2. Dessiatnikoff, A.; Deswarte, Y.; Alata É.; Nicomette, V. Potential attacks on onboard aerospace systems. *IEEE Secur. Priv.* **2012**, *10*, 71–74. [[CrossRef](#)]
3. Ryon, L.; Rice, G. A safety-focused security risk assessment of commercial aircraft avionics. In Proceedings of the IEEE/AIAA 37th Digital Avionics Systems Conference (DASC), London, UK, 23–27 September 2018; pp. 1–8.
4. Chernov, A.V.; Butakova, M.A.; Karpenko, E.V. Security incident detection technique for multilevel intelligent control systems on railway transport in Russia. In Proceedings of the 23rd Telecommunications Forum Telfor (TELFOR), Belgrade, Serbia, 24–26 November 2015; pp. 1–4.
5. Basak, A.; Bhunia, S.; Tkacik, T.; Ray, S. Security assurance for system-on-chip designs with untrusted IPs. *IEEE Trans. Inf. Forensics Secur.* **2017**, *12*, 1515–1528. [[CrossRef](#)]
6. De, A.; Khan, M.N.I.; Nagarajan, K.; Ghosh, S. HarTBleed: Using hardware Trojans for data leakage exploits. *IEEE Trans. Very Large Scale Integr. Syst.* **2020**, *28*, 968–979. [[CrossRef](#)]
7. Ghosh, A.; McGraw, G. Lost decade or golden era: Computer security since 9/11. *IEEE Secur. Priv.* **2012**, *10*, 6–10. [[CrossRef](#)]
8. Chen, G.; Jin, H.; Zou, D.; Zhou, B.B.; Liang, Z.; Zheng, W.; Shi, X. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2013**, *10*, 368–379. [[CrossRef](#)]
9. Alam, M.; Roy, D.B.; Bhattacharya, S.; Govindan, V.; Chakraborty, R.S.; Mukhopadhyay, D. SmashClean: A hardware level mitigation to stack smashing attacks in OpenRISC. In Proceedings of the ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE), Kanpur, India, 18–20 November 2016; pp. 1–4.
10. De, A.; Basu, A.; Ghosh, S.; Jaeger, T. Hardware assisted buffer protection mechanisms for embedded RISC-V. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 4453–4465. [[CrossRef](#)]
11. Kim, L.; Villasenor, J.D. Dynamic function verification for system on chip security against hardware-based attacks. *IEEE Trans. Rel.* **2015**, *64*, 1229–1242. [[CrossRef](#)]
12. Bhunia, S.; Hsiao, M.S.; Banga, M.; Narasimhan, S. Hardware Trojan attacks: Threat analysis and countermeasures. *Proc. IEEE* **2014**, *102*, 1229–1247. [[CrossRef](#)]
13. Ghandali, S.; Moos, T.; Moradi, A.; Paar, C. Side-channel hardware Trojan for provably-secure SCA-protected implementations. *IEEE Trans. Very Large Scale Integr. Syst.* **2020**, *28*, 1435–1448. [[CrossRef](#)]
14. Salamat, B.; Jackson, T.; Wagner, G.; Wimmer, C.; Franz, M. Runtime defense against code injection attacks using replicated execution. *IEEE Trans. Dependable Secur. Comput.* **2011**, *8*, 588–601. [[CrossRef](#)]
15. Qiu, P.; Lyu, Y.; Zhang, J.; Wang, D.; Qu, G. Control flow integrity based on lightweight encryption architecture. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2018**, *37*, 1358–1369. [[CrossRef](#)]
16. Shen, Z.; Dharsee, K.; Criswell, J. Fast execute-only memory for embedded systems. In Proceedings of the IEEE Secure Development (SecDev), Atlanta, GA, USA, 28–30 September 2020; pp. 7–14.
17. Snow, K.Z.; Monrose, F.; Davi, L.; Dmitrienko, A.; Liebchen C.; Sadeghi, A. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In Proceedings of the IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 19–22 May 2013; pp. 574–588.
18. Matsuo, I.B.M.; Zhao, L.; Lee, W. A dual modular redundancy scheme for CPU–FPGA platform-based systems. *IEEE Trans. Ind. Appl.* **2018**, *54*, 5621–5629. [[CrossRef](#)]
19. Kretzschmar, U.; Gomez-Cornejo, J.; Astarloa, A.; Bidarte, U.; Del Ser, J. Synchronization of faulty processors in coarse-grained TMR protected partially reconfigurable FPGA designs. *Reliab. Eng. Syst. Saf.* **2016**, *151*, 1–9. [[CrossRef](#)]
20. Wang, X.; Mal-Sarkar, T.; Krishna, A.; Narasimhan S.; Bhunia, S. Software exploitable hardware Trojans in embedded processor. In Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Austin, TX, USA, 3–5 October 2012; pp. 55–58.
21. Yang, K.; Hicks, M.; Dong, Q.; Austin, T.; Sylvester, D. A2: Analog malicious hardware. In Proceedings of the IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2016; pp. 18–37.
22. Khan, M.N.I.; De, A.; Ghosh, S. Cache-Out: Leaking cache memory using hardware Trojan. *IEEE Trans. Very Large Scale Integr. Syst.* **2020**, *28*, 1461–1470. [[CrossRef](#)]
23. Skorobogatov, S. Hardware security implications of reliability, remanence, and recovery in embedded memory. *Hardw. Syst. Secur.* **2018**, *2*, 314–321. [[CrossRef](#)]
24. Imtiaz Khan, M.N.; Nagarajan, K.; Ghosh, S. Hardware Trojans in emerging non-volatile memories. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 25–29 March 2019; pp. 396–401.
25. Younan, Y.; Joosen, W.; Piessens, F. *Code Injection in C and CPP: A Survey of Vulnerabilities and Countermeasures*; Katholieke Universiteit Leuven: Leuven, Belgium, 2004.
26. Wang, X.; Zhang, Z.; Hao, Q.; Xu, D.; Wang, J.; Jia, H.; Zhou, Z. Hardware-assisted security monitoring unit for real-time ensuring secure instruction execution and data processing in embedded systems. *Micromachines* **2021**, *12*, 1450. [[CrossRef](#)]
27. Wu, W.; Wu, S.; Zhang, L.; Zou, J.; Dong, L. Lhash: A lightweight hash function. In Proceedings of the International Conference on Information Security and Cryptology, Seoul, Korea, 27–29 November 2013; Springer: Cham, Switzerland, 2013; pp. 291–308.

28. Luan, G.; Bai, Y.; Wang, C.; Zeng, J.; Chen, Q. An efficient checkpoint and recovery mechanism for real-time embedded systems. In Proceedings of the 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications, Melbourne, VIC, Australia, 11–13 December 2018; pp. 824–831.
29. Guthaus, M.R.; Ringenberg, J.S.; Ernst, D.; Austin, T.M.; Mudge, T.; Brown, R.B. MiBench: A free, commercially representative embedded benchmark suite. In Proceedings of the 4th Annual IEEE International Workshop Workload Characterization, WWC-4, Austin, TX, USA, 2 December 2001; pp. 3–14.
30. Bakiri, M.; Titri, S.; Izeboudjen, N.; Abid, F.; Louiz, F.; Lazib, D. Embedded system with Linux Kernel based on OpenRISC 1200-V3. In Proceedings of the International Conference on Sciences of Electronics, Sousse, Tunisia, 21–24 March 2012; pp. 177–182.
31. Arora, D.; Ravi, S.; Raghunathan, A.; Jha, N.K. Hardware-assisted run-time monitoring for secure program execution on embedded processors. *IEEE Trans. Very Large Scale Integr. Syst.* **2006**, *14*, 1295–1308. [[CrossRef](#)]
32. Mashtizadeh, A.J.; Bittau, A.; Boneh, D.; Mazières, D. CCFI: Cryptographically enforced control flow integrity. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 941–951.
33. Vemu, R.; Abraham, J. CEDA: Control-flow error detection using assertions. *IEEE Trans. Comput.* **2011**, *60*, 1233–1245. [[CrossRef](#)]
34. Lin, H.; Fei, Y.; Guan, X.; Shi, Z.J. Architectural enhancement and system software support for program code integrity monitoring in application-specific instruction-set processors. *IEEE Trans. Very Large Scale Integr. VLSI Syst.* **2010**, *18*, 1519–1532. [[CrossRef](#)]
35. Zhang, J.; Qi, B.; Qin Z.; Qu, G. HCIC: Hardware-assisted control-flow integrity checking. *IEEE Internet Things J.* **2019**, *6*, 458–471. [[CrossRef](#)]
36. Aktas, E.; Afram, F.; Ghose K. Continuous, low overhead, run-time validation of program executions. In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, 13–17 December 2014; pp. 229–241.
37. Meng, D.; Hou, R.; Shi, G.; Tu, B.; Yu A.; Zhu Z.; Jia X.; Wen Y.; Yang Y. Built-in security computer: Deploying security-first architecture using active security processor. *IEEE Trans. Comput.* **2020**, *69*, 1571–1583. [[CrossRef](#)]
38. Guo, J.; Peyrin, T.; Poschmann, A. The PHOTON family of lightweight hash functions. *Adv. Cryptol.* **2011**, *6841*, 222–239.