Hardware Assisted Pre-emptive Control Flow Checking for Embedded Processors to improve Reliability

Roshan G. Ragel Sri Parameswaran
University of New South Wales and National ICT Australia*
Sydney NSW 2052 Australia.
{roshanr,sridevan}@cse.unsw.edu.au

ABSTRACT

Reliability in embedded processors can be improved by control flow checking and such checking can be conducted using software or hardware. Proposed software-only approaches suffer from significant code size penalties, resulting in poor performance. Proposed hardware-assisted approaches are not scalable and therefore cannot be implemented in real embedded systems. This paper presents a scalable, cost effective and novel fault detection technique, to ensure proper control flow of a program. This technique includes architectural changes to the processor and software modifications. While architectural refinement incorporates additional instructions, the software transformation utilizes these instructions into the program flow. Applications from an embedded systems benchmark suite are used for testing and evaluation. The overheads are compared with the state of the art approach that performs the same error coverage using software-only techniques. Our method has greatly reduced overheads compared to the state of the art. Our approach increased code size by between 3.85-11.2% and reduced performance by just 0.24-1.47% for eight different industry standard applications. The additional hardware (gates) overhead in this approach was just 3.59%. In contrast, the state of the art software-only approach required 50-150% additional code, and reduced performance by 53.5-99.5% when error detection was inserted.

Categories and Subject Descriptors
B.8.1 [Performance and Reliability]: Reliability, Testing, and Fault-Tolerance

General Terms
Design, Performance, Reliability

Keywords
Control Flow Checking, Embedded Processor Reliability, Hardware/Software Technique, Micro-instruction Routines, Preemptive Fault Detection, Reliable Processors

1. INTRODUCTION

Current processor based systems are often required to deal with critical applications, making reliability an important concern in the design of such systems. In this paper, we focus upon control flow errors (CFE), errors that cause divergence from the proper control flow of an application program. Techniques for control flow checking (CFC) are widely used to enhance the reliability of computer systems. The basic concept of CFC is to verify that the runtime flow of a system corresponds to the expected or the specified behavior of the same system.

CFEs, if not detected in time could lead to data corruption, process crashes, error propagation or fail-silence violations. An application or a process is said to be fail-silent if it either works correctly or becomes silent (stops functioning). A violation to fail-silent behavior is called fail-silence violation. Fault injection studies conducted by Ohlsson et al. [26] and Schutte et al. [31] show that the CFEs comprise between 33% and 77% of all transient errors that occur in a computer system. Given that the majority of the system errors are transient and are not reproducible [15], a runtime error detection mechanism is the only feasible mechanism to detect CFEs. It is essential to take the following parameters into consideration while crafting CFC mechanisms: error detection coverage; error detection latency; processor performance; memory overhead; and monitor complexity.

CFEs occur due to various low-level errors or failures. The three error models used in this paper are bit flips in instruction memory, transmission errors during communication, and errors in registers. *Bit flips in instruction memory* will be caused due to burst errors and will corrupt instructions. They will occur in on-chip or off-chip memory. *Transmission errors* may occur when bit vectors are transferred between any two levels of memory hierarchy or between different functional units. *Corruption of register values*, in particular those which determine the destination address or the condition of a branch instruction could cause an illegal branch in the control flow of an application.

The technique propsoed in this paper detects bit flips in instruction memory by duplicating the control flow instruction (CFI) and performing a hardware checking. The duplicate copy is inserted just before the original CFI. At runtime, the duplicate copy of the CFI will perform a comparison against the original fetched instruction and will send an error if there is a mismatch. The error signal is sent before the original CFI is executed, thus making the error detection preemptive. Preemptive CFE detection prevents process crashes (a crash of the entire process incurs a higher recovery overhead due to the overhead of process creation) and error propagation (latency of propagated errors is at least several hundreds of instruction cycles [12] and errors which are not detected early may cause severe problems like check point corruption which complicates error recovery).

CFEs are usually detected by dynamic software monitoring where: additional software code is inserted into application programs; or, hardware-assisted runtime monitoring is enabled, where a hardware block is dedicated to performing security and reliability checks (for example a watchdog processor for CFC in [17]). Software only approaches increase code size enormously by adding many additional instructions to perform monitoring and therefore signif-

*National ICT Australia is funded through the Australian Governments Backing Australias Ability initiative, in part through the Australian Research Council.

CODES+ISSS'06, October 22–25, 2006, Seoul, Korea.

icantly reduce performance. Hardware assisted approaches use additional hardware blocks to perform monitoring and therefore incur considerably high hardware cost and needs self checking mechanisms within the monitors to ensure that the monitoring hardware themselves are reliable. Furthermore, techniques those use additional hardware monitors are not scalable.

This paper presents a hardware software technique to detect CFEs at the granularity of micro-instructions (MI). For the first time, such a technique is being used to deal with this problem and we are able to reduce the overheads to a considerable minimum. MIs are instructions which control data flow, and instruction-execution sequencing, in a processor at a more fundamental level than the level of machine instructions. The SI performed in our scheme is minimal compared to software only approaches, because it is used only as an interface between check points and MI routines. Our checking architecture could be deployed in any embedded processor on which we have the design control to observe its control flow at runtime and trigger a flag when any unexpected control flow pattern is detected.

Parity checking as implemented in many other well known solutions [8], provides good protection against single bit errors when the probability of errors are independent. However, in many circumstances, errors come in groups, which we call bit bursts. Parity checking provides very limited protection against bit bursts. The technique proposed in this paper detects CFE caused by not only independent bit flips, but also bit bursts.

1.1 Motivation

At a given time, a processor executes only a few instructions and large part of the processor is idle. Utilizing these idling hardware components by sharing them with the monitoring hardware, to perform CFE detection reduces the impact of the monitors on hardware cost. Using MI routines within the machine instructions, allows us to share most of the monitoring hardware. Therefore, our technique requires little hardware overhead in comparison to having additional hardware blocks outside the processor. This reduction in overhead is due to maximal sharing of hardware resources of the processor.

1.2 Paper Overview and Organisation

In this paper, we address fault tolerance by focusing on the specific problem of ensuring correct execution of expected control flow of a program. We have evaluated memory, area and clock period overheads associated with the proposed architecture using applications from an embedded systems benchmark suite called MiBench [11]. Hardware synthesis and simulations are performed by commercial design tools. Results demonstrate that our proposed solution has considerable reduction in the overheads compared to solutions proposed by other techniques in the literature.

The remainder of this paper is organized as follows. A survey on related work is presented in Section 2. Section 3 presents the proposed error checking architecture. Section 4 describes a systematic methodology to design the proposed solution for a given architecture. Implementation and evaluation are presented in Section 5. Results are presented in Section 6 and conclusions in Section 7.

2. RELATED WORK

For the last three decades, many different CFC mechanisms have been proposed to verify proper flow of application programs. Some of the first known publications on CFE detection include [29, 37], where the authors outline a general software assisted scheme for CFE detection. CFE detection techniques can be divided into two major categories based on where the error detection scheme is implemented: one, hardware- or architecture-based CFE detection schemes; and two, software-based CFE detection schemes. In hardware assisted CFE detection, the application is divided into blocks, and signatures are associated with those blocks statically. Then at runtime, a similar signature is calculated by a hardware monitor and compared against the one calculated statically. Software CFE detection is performed by having appropriate signatures for similar blocks as per hardware techniques, but the checking is done by software code inserted into the instruction vector at compile-time. Since software code performs the necessary checking, there is no need for a separate hardware monitor.

Hardware assisted error detection schemes [5, 22, 24, 28, 30, 33] use watchdog processors to compute runtime signatures from the instruction control flow, compare them against pre-computed signatures, and thus validate the application behavior. In [17], the authors compare and discuss system level CFE detection using different techniques. In [16] the CFE detection scheme is tested with very simple applications and a non-complex hardware architecture. In [24] the control flow graph is divided into a sequence of branch free nodes, called *path sets* which will be assigned with a unique signature and compared against the runtime signature. This approach needs a complicated parser to generate the path set and then the signature.

Hardware assisted CFE detection schemes could be classified further into two categories depending upon how the static signatures are stored and accessed. The first embeds signatures into the application binary itself [3, 4, 14, 31, 32, 36] and the second uses a separate memory (dedicated memory of the watchdog processor) to store and access the signatures [19, 20]. In [32] the signature is embedded into the application's instruction stream at the assembly level, and branch address hashing is used to reduce memory overhead. In [19], the signature calculated at compile time is stored in a separate memory belonging to a watchdog processor and therefore the original application/program does not have to be modified. There exists a theoretical exploration [35] of a possible hardware assisted CFC without storing a compile time signature. As far as we are aware, no implementation exists for the approach shown in [35].

A concurrent, on-chip hardware assisted CFC technique is proposed in [13]. The technique uses control signatures to enable CFC. However, the designers of [13] prohibit indirect register branches as they are unpredictable at compile time. As most of the CFIs are indirect register branches, this technique is not general enough so that it could be applied to most of the embedded processors. Another hardware assisted concurrent error-detection method is proposed in [8] for embedded space flight applications. [8] uses parity checks in registers and signatures for CFC (*XOR*ing instructions until a check-point where they are verified). Multiple bit errors (or bursts) are not captured by this technique and signatures will not reveal the exact point of the error in the flow of an application.

Software error detection scheme uses software routines to check proper control flow at runtime. The routines are inserted into the application at assembly level or at a higher level. Some of the techniques are described in [1, 9, 10, 12, 21, 25]. Based on how the checking routines are used, we have different classifications for software based control flow checking. Control Checking with Assertions (CCA) inserts assertions at the entry and exit points of identified branch-free intervals [12, 18, 23]. CCA is implemented as a pre-processor to a compiler, based on the syntactic structure of the language and does not require any CF graph generation and analysis. An enhanced version of CCA, Enhanced Control Checking with Assertions (ECCA) is proposed in [1] which targets real-time distributed systems for the detection of CFEs. ECCA addresses the limitations of CCA, and is implemented at both high and intermediate levels (register transfer language) of a language. Another type of software based CFC is called Block Signature Self Checking (BSSC) [21]. An application program is divided into basic blocks and each of these are assigned with a signature. A set of instructions (assertions) at the end of a basic block reads the

signature from a runtime variable and compares it to an embedded signature following the instructions. A mismatch in the comparison indicates a CFE.

The PreEmptive COntrol Signature (PECOS) checking [2] is the only preemptive software based technique available to date for control flow error detection. Even though we use a hardware software approach, our approach is inspired by PECOS. Preemptive detection means that the error is detected before the erroneous CFI is executed. PECOS uses assertions formed with assembly instructions that can be embedded in the assembly language code of an application. PECOS detects CFEs caused by a direct or an indirect fault in the control flow instruction. Due to the extensive use of assembly instructions as assertions, PECOS has very high overheads in memory and performance. Our hardware software approach discussed in this paper reduces the overhead by trading a few more transistors for performance gain. Furthermore, we propose techniques to capture CFEs caused by program counter corruptions.

Generally, hardware assisted schemes are not scalable as they perform monitoring by observing the memory access patterns using watchdog processors. These schemes will only work for a processor that runs a single application. Our scheme is scalable like a software-only approach due to the software instrumentation we perform, however unlike software-only approaches our method incurs very little code size and performance overheads.

2.1 Contributions

Our contributions in this work are:

1. detection and correction of bit bursts that causes CFE unlike parity based hardware assisted techniques;
2. preemptive and correctable CFE detection as opposed to signature based hardware assisted mechanisms;
3. a scalable mechanism as we use SI as the interface for CFC;
4. a methodology for embedding CFE monitoring at the granularity of MIs;
5. a technique that requires very little code size overhead and performance overheads compared to the software-only approaches; and
6. a method to share most of the monitoring hardware and therefore requires very little additional hardware.

2.2 Limitations

Our scheme will not capture CFEs caused by a corrupted non CFI turning into a CFI. However, the hamming distances between the opcodes of non CFIs and CFIs are usually high in typical Instruction Set Architectures (ISA) [2] and therefore we can safely assume that a non CFI turning into CFI is unlikely. We also assume full control over the whole hardware design process.

3. ERROR CHECKING ARCHITECTURE

In this section, we provide an overview of our hardware architecture for CFC. The hardware modifications that are performed to enable CFC on a pipelined RISC architecture are:

1. enhancements to the controller to treat CFIs to handle instruction memory bit flips;
2. an addition of a shadow register file and the related logic to handle CFEs caused by indirect CFIs; and
3. an inclusion of a shadow PC and the accompanying logic to handle program counter corruption that causes CFEs.

In this section, we also describe how the architecture works at runtime to detect CFEs.

3.1 Instruction Memory Bit Flips Detection

Figure 1 depicts the conceptual flow diagram of the proposed instruction memory bit flips detection and correction mechanism. For ease of illustration, we only depict the hardware units related to the checking architecture with respect to the whole architecture of an embedded processor. *IMem* in Figure 1 represents the instruction memory segment of the processor. Each CFI of a given application (*CFIo*) is preceded by a duplicate copy of the same instruction (*CFId*). This instrumentation is performed by a software component at compile time. The pipelines stages shown in the upper part of Figure 1 belongs to *CFId* and the lower part belongs to *CFIo*. Each fetch writes the binary of the instruction fetched into the instruction register (IR) of the processor. Whether a fetched CFI is either original or duplicate is decided by the processor by checking a special single bit flag that flips back and forth for each CFI. A duplicate CFI, *CFId* in its execution stage (EXE), compares its own binary against the one fetched next to it and signals an error when there is a mismatch.

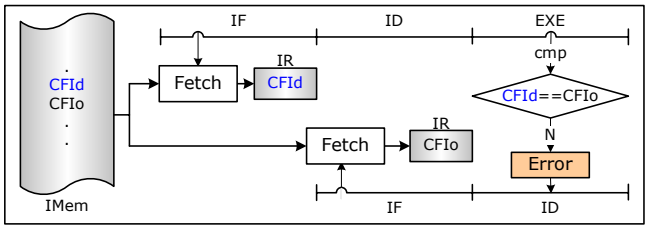

Figure 1: Control Flow Checking Architecture

MIs of the CFI are formed such that the CFI will perform the tasks as described above in the same order. The outcome from the comparison between *CFIo* and *CFId* could be used to either detect the error and stop the application or correct the CFI by assuming an error free duplicate CFI. Our method's preemptive error detection property is demonstrated in this approach, since a fault in a CFI is detected before the erroneous instruction itself is executed.

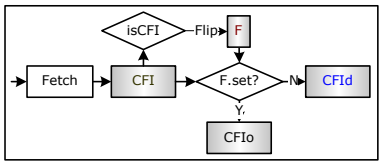

Figure 2: Separating Original and Duplicate CFIs

Figure 2 depicts how duplicate and original CFIs are distinguished by our processor when an instrumented application is executed. A single bit *flag register (F)* is added (or the current flag register could be extended) to the architecture of the processor which will be reset at the beginning. When CFIs including duplicates are encountered the flag will be flipped between zero and one. If the flag is set to one when a CFI is fetched, then the fetched CFI is deemed to be an original (*CFIo*) and otherwise it is deemed as a duplicate (*CFId*).

The same technique described in this section will also detect CFEs caused by errornous transmissions between different memory hierarchies and the transient CFEs in the instruction memory data bus.

3.2 Shadow Register File

For CFIs with register indirect addressing, it is essential to verify the contents of the registers apart from the binaries of the CFIs themselves. The rudimentary solution is to have a shadow register file and make each register writeback to write to both the real and the shadow register files. When a register is used in a CFI, the duplicate CFI will not only perform a comparison between the binaries of the instructions, but also perform comparisons between the real and shadow registers used by the CFIs. Performing writebacks to shadow registers for each instruction will involve huge

amount of unwanted switching activity. This could be reduced by performing shadow register writebacks at only necessary points in an application program. This could be achieved by using the *use-def chains* [1] (register definitions) of a particular application, which is already present at compile time in all the optimizing compilers.

3.3 Shadow Program Counter

CFEs may also occur due to bit flips or a burst in the program counter (PC). We propose a shadow PC to overcome this. A shadow PC is included in the hardware and will be loaded and incremented synchronously with the real PC. When a PC read operation is performed a copy of the PC value will also be read from the shadow PC and a comparison is performed between them. A mismatch will result in program abortion, or continuation with the assumption that the shadow PC is not corrupted. This scheme will give a better error coverage than a processor without a shadow PC.

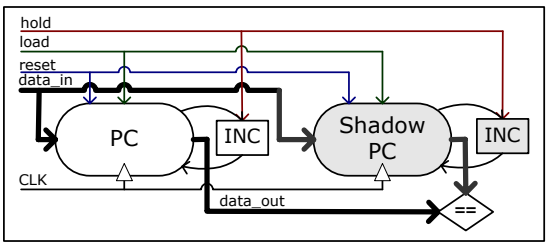

Figure 3: Shadow Program Counter

Figure 3 depicts the proposed architecture for a shadow PC. The input signals to the real PC are extended to the shadow PC and output from the real PC is compared against the output from the shadow PC to detect bit flips or a burst in the real PC.

4. DESIGN FLOW

In this section, an overview of the proposed design flow for the checking architecture is provided. First, the design of a software interface that allows the applications to interact with the architectural enhancement is described, and then the design of the architectural enhancement itself is discussed.

4.1 Software Design

Figure 4(a) describes the implementation details of the interface between an application program and the fault checking hardware. It is worth noting that the duplicate CFIs inserted at the right places in an application serves as the interface between software and fault checking hardware. In the software instrumentation process, the source code of an application is compiled by the front end of a compiler and the assembly code for the target ISA is produced. Then a software parser is used to instrument the assembly code. CFIs are located and duplicate copies of the CFIs are inserted into the application.

The SI described above is in the instrumentation process for CFIs with constant offsets. For CFIs with register indirect addressing, the register source will be duplicated by means of shadow registers at the time of register definition and used by the duplicate CFIs for comparison. As described in Section 3.2 this could be achieved by two different means and they are: [i] by enabling *shadow register writeback* whenever a register writeback is performed; or [ii] by generating and using special instructions to perform register writeback only in places of the application where a register writeback related to CFI is performed. The former will incur additional unwanted switching activity and the later will require building special

[1]Data structures which model the relationship between the definitions of variables, and their uses in a sequence of assignments

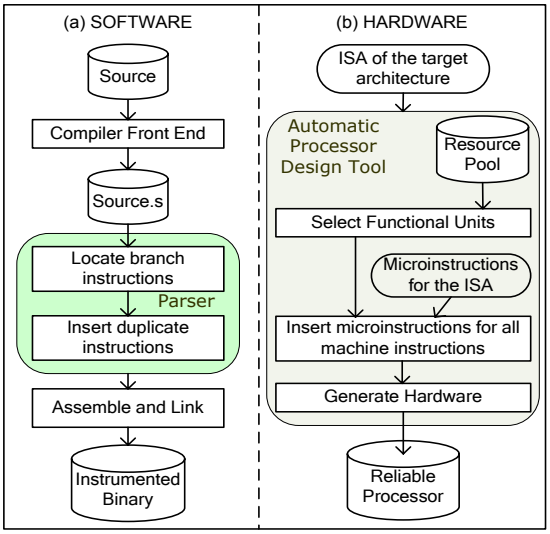

Figure 4: Design flow for the control flow checking architecture

instructions and instrumenting them into the application with the help of *use-def chains*.

Finally, an instrumented version of the assembly program is assembled and linked using the back-end of the same compiler, to generate the binary for the target architecture.

4.2 Architectural Design

Without losing the generality of our technique, we use an automatic processor design tool to implement the reliable processor model in hardware. This automatic design tool is used to design Application Specific Instruction-set Processors (ASIPs) [6, 7], custom designed for applications or application domains. Automatic processor design tools serves as a perfect starting place to build processor models.

Figure 4(b) describes the hardware design process of the model for the reliable processor. In the tool the functional units required to implement the processor will be chosen from a resource pool. Using the information of the ISAs, MI routines are formed, and are included into the processor model design. These MI routines will form the logic of the processor that will do the CFC along with regular operations at runtime. The final task in the architectural design process is to generate the hardware model in a hardware description language for simulation [behavioral] and synthesis [gate level] (indicated by *Reliable Processor* in Figure 4).

5. IMPLEMENTATION AND EVALUATION

Even though the techniques described in this paper for CFC can be deployed in any type of embedded processor architecture, we have taken the PISA (portable instruction set architecture) instruction set as implemented in SimpleScalar™ tool set for our experimental implementation. The PISA instruction set is a simple MIPS like instruction set. CFC in the processor is enabled by altering the rapid processor design process described in [27] for hardware synthesis (allowing a processor described in VHDL which is synthesizable).

To evaluate our approach, applications from MiBench benchmark suite were taken and compiled with the GNU/GCC® cross-compiler for the PISA instruction set. As mentioned previously, an automatic processor design tool, called ASIP Meister [34] is used to generate the VHDL description of the target processor as described in section 4.2. The output of the ASIP Meister are the VHDL models of the processor for simulation and synthesis.

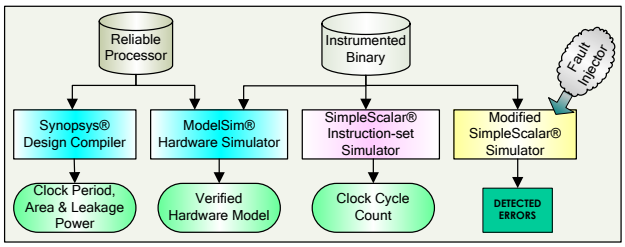

Figure 5: Testing and Evaluation

As shown in Figure 5, the instrumented binary produced from the software design process and the processor simulation model generated by the hardware design tool are used in ModelSim® hardware simulator to verify the correctness of our design. Applications with different CFIs are simulated in Modelsim® and their behaviors are observed and verified by looking at the waveforms of related signals.

The gate level VHDL model from ASIP Meister is used with Synopsys Design Compiler® to obtain the clock period, area and leakage power overheads.

We evaluated the clock cycle overhead of the proposed architecture using a cycle accurate instruction set simulator, SimpleScalar™ 3.0/PISA tool set. The simulator is modified to include CFC as proposed in Section 3. The simulator is built around the existing cycle accurate simulator *sim-outorder* in the tool set and used to calculate the clock cycle overheads. As depicted in Figure 5, the same simulator is modified to perform fault injection analysis and the results are tabulated in the next Section. The micro-architectural parameters of SimpleScalar™ were configured to model a typical embedded processor as designed by ASIP meister. The parameters used for the simulated processor are shown in Table 1.

Parameter	Value	Parameter	Value
Issue	in-order	Issue width	1
Fetch queue size	4	Commit width	1
L1 I-cache	16kB	L1 D-cache	16kB
L1 I-cache latency	1 cycle	L1 D-cache latency	1 cycle
Initial memory latency	18 cycles	Memory latency	2 cycles

Table 1: Architectural Parameters for Simulation

The clock cycle counts from the modified SimpleScalar™ simulator and the clock period from Synopsys Design Compiler® are used to calculate the total execution times of all the applications.

6. EXPERIMENTAL RESULTS

In this section, we present memory, area and power overheads incurred by the proposed CFC solution, as well as the impact of the technique on performance (total execution time). Later we give results from the fault injection analysis performed on our model processor when our solution is implemented. For the purpose of experiments we have used the PISA instruction set as described in Section 5 and applications from the MiBench benchmark suite which represents typical workload for embedded processors.

6.1 Hardware Overhead

Table 2 tabulates the area, clock period and leakage power overheads due to the changes in the hardware. The overheads here represent the extra logic in our design. Taiwan Semiconductor Manufacturing Company's (TSMC) 90nm core library with typical conditions enabled is used for the hardware synthesis. The second column in Table 2 represents the parameters for the processor model without the CFC enabled (the base processor) and the third column represents the parameters of the processor model when the hardware for CFC is enabled. The percentage of overheads in area is 3.59%, clock period is 0.24% and leakage power is 3.71%.

Parameters	Without CFC	With CFC	% overhead
Area (cells)	228489	236700	3.59
Clock Period (ns)	16.85	16.89	0.24
Leakage Power (μW)	485	503	3.71

Table 2: Hardware Overhead

6.2 Performance Overhead

Table 3 reports the performance overhead incurred by our scheme for different applications (first column) from MiBench benchmark suite. In Table 3 columns 2-4 tabulate the clock cycle comparisons (in millions) and percentage of overheads and columns 5-7 tabulate execution time comparison (in seconds) and percentage of overheads. The columns that are sub-titled *NoCFC* represent simulations of the processor model without the CFC enabled and *CFC* represents simulations of the processor model with the CFC enabled.

Benchmarks	Clock Cycle/10^6			Execution Time/s		
	NoCFC	CFC	%	NoCFC	CFC	%
adpcm.decode	121.6	122.8	0.99	2.05	2.07	1.23
adpcm.encode	89.96	90.12	0.18	1.52	1.52	0.42
blowf.encrypt	79.21	79.49	0.35	1.34	1.34	0.59
blowf.decrypt	80.44	81.05	0.76	1.37	1.37	1.00
crc32.checksum	57.62	57.62	0.00	0.97	0.97	0.24
jpeg.compress	16.41	16.56	0.91	0.28	0.28	1.15
jpeg.decompress	10.79	10.89	0.93	0.18	0.18	1.17
jpeg.transcoding	8.96	9.07	1.23	0.15	0.15	1.47

Table 3: Performance Comparison

From Table 3, the clock cycle overheads range from 0.00% to 1.23% with an average of 0.67% and the execution time overheads range from 0.24% to 1.47% with an average of 0.91%. In contrast, the performance (execution time) overhead in software only PECOS technique reported was in the range of 53.5-99.5%.

6.3 Codesize Overhead

Table 4 reports the code size overhead and the overhead in the number of instructions executed resulting from our SI for eight different applications (first column). In Table 4 columns 2-4 tabulate the comparison between the number of executed instructions (in millions) and percentage of overheads and columns 5-7 tabulate code size (the number of lines) comparisons and percentage of overheads. The columns with title *NoCFC* represents simulations of the processor model without the CFC enabled and *CFC* represents simulations of the processor model with the CFC enabled.

Benchmarks	Executed Inst./10^6			Code Size		
	NoCFC	CFC	%	NoCFC	CFC	%
adpcm.decode	76.55	92.98	13.88	623	676	8.50
adpcm.encode	58.91	72.60	15.37	618	670	8.40
blowf.encrypt	58.43	64.44	4.26	6463	6712	3.85
blowf.decrypt	59.04	65.05	5.97	6463	6712	3.85
crc32.checksum	42.51	50.72	12.82	527	549	4.23
jpeg.compress	11.62	12.88	7.41	58650	65160	11.1
jpeg.decompress	7.45	7.96	3.99	56577	62914	11.2
jpeg.transcoding	5.91	7.41	16.88	52605	58444	11.1

Table 4: Codesize Comparison

From Table 4, overheads in the number of instructions executed ranges from 3.99% to 16.9% with an average of 10.7% and code

size overhead ranges from 3.85% to 11.2% with an average of 7.78%. The code size overhead of software only PECOS technique was in the range of 50-150%.

6.4 Fault Injection Analysis

Table 5 tabulates the results of the fault injection analysis performed on our control flow detection architecture. Random memory errors are generated by performing bit bursts in the instruction memory. Random memory errors represent a wide range of transient errors in hardware and some errors in software. For each application, 10,000 faults are inserted one at a time and the runtime program behavior is observed to identify whether it is captured by our technique. With the current design, an average of 85.3% of the injected CFEs are detected by our system. The rest of the erroneous CFIs fall into the library functions of the application which are presently not instrumented. We will be able to achieve 100% CFE coverage, if we are able to instrument the library functions those are used in the applications.

Benchmarks	No. of Faults		CFEs	Captured by our scheme
	Total	Activated		
adpcm.decode	10000	5723	769	653
adpcm.encode	10000	4635	607	526
blowf.encrypt	10000	6242	911	824
blowf.decrypt	10000	6283	949	819
crc32.checksum	10000	6264	969	923
jpeg.compress	10000	2947	417	334
jpeg.decompress	10000	2901	428	342
jpeg.transcoding	10000	4176	710	558

Table 5: Fault Injection Analysis

This fault injection analysis is only capable of testing our design for instruction memory bit flips and bursts detection. Further analysis could be performed to test the capability of the shadow register file and the shadow PC by injecting faults into the register file and the PC. However, all the injected errors in the register file will be captured by our scheme given that the CFIs which use the errornous registers are checked for errors. Therefore, the error coverage of the shadow register file is equal to the error coverage of the CFIs. Furthermore, all the injected errors in a PC will be captured by our scheme with 100% error coverage.

7. CONCLUSIONS

In this paper, we have presented a hardware software technique to detect CFEs caused by bit flips and bursts at runtime before an erroneous CFI is executed. We have formulated a formal methodology to accommodate this technique within an automatic embedded processor design flow. Our evaluation studies reveal that the solution we have proposed is capable of handling CFEs with as little as 3.59% of area and 0.91% of performance overheads. These overheads are minimal compared to the software solution that deals with the same problem. We conclude that by asserting CFC as a design requirement of an embedded processor, it is practicable to reduce the overhead of CFC as minimal as possible.

Fault injection analysis demonstrates that our solution is capable of capturing 85.3% of the injected CFEs in the instruction memory. The limitation of the error coverage is due to the non instrumented code coming from the runtime libraries. Furthermore, our scheme will also detect CFEs caused by bit flips and bursts in register file and PC with 100% coverage as long as the erroneous CFI is instrumented. Our error detection scheme is preemptive and is capable of correcting CFEs (assuming the error is in the original CFI/register-file/PC) without any additional overheads. We believe that the technique described in this paper could be used with any embedded processors for detecting CFEs efficiently.

8. REFERENCES

[1] Z. Alkhalifa et al. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Transaction on Parallel and Distributed Systems*, pages 627–641, 1999.

[2] S. Bagchi et al. Hierarchical error detection in a software implemented fault tolerance (sift) environment. *IEEE Transactions on Knowledge and Data Engineering*, 12:203–224, March/April 2000.

[3] X. Delord and G. Saucier. Control flow checking in pipelined RISC microprocessors: the Motorola MC88100 case study. In *EUROMICRO '90*, pages 162–169, June 1990.

[4] X. Delord and G. Saucier. Formalizing Signature Analysis for Control Flow Checking of Pipelined RISC Microprocessors. In *Test conference*, pages 936–945, October 1991.

[5] B. Eschermann. On combining off-line BIST and on-line control flow checking. In *FTCS-22*, pages 298–305, July 1992.

[6] J. A. Fisher. Customized instruction-sets for embedded processors. In *DAC'99*, pages 253–257, 1999.

[7] J. A. Fisher, P. Faraboschi, and G. Desoli. Custom-fit processors: letting applications define architectures. In *MICRO29*, pages 324–335, 1996.

[8] J. Gaisler. Concurrent error-detection and modular fault-tolerance in a 32-bit processing core for embedded space flight applications. In *FTCS'94*, pages 128–130, 1994.

[9] O. Goloubeva et al. Soft-error detection using control flow assertions. In *DFT'03*, pages 581–588, November 2003.

[10] O. Goloubeva et al. Improved software-based processor control-flow errors detection technique. In *Reliability and maintainability symposium*, pages 583–589, January 2005.

[11] M. R. Guthaus et al. Mibench: A free, commercially representative Embedded Benchmark Suite. *IEEE 4th Annual Workshop on Workload Characterization*, pages 83–94, December 2001.

[12] G. Kanawati et al. Evaluation of integrated system-level checks for on-line error detection. In *Computer performance and dependability symposium*, pages 292–301, September 1996.

[13] R. Leveugle, T. Michel, and G. Saucier. Design of Microprocessors with Built-in On-line Test. In *FTCS-20*, pages 450–456, June 1990.

[14] D. J. Lu. Watchdog processors and structural integrity checking. *IEEE Trans. Computers*, 31(7):681–685, 1982.

[15] M. R. Lyu, editor. *Software Fault Tolerance*. John Wiley and Sons Ltd, 1995.

[16] H. Madeira and J. Silva. On-line signature learning and checking: experimental evaluation. In *COMPEURO'91*, pages 642–643, July 1991.

[17] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors - a survey. *IEEE Trans. Computers*, 37(2):160–174, 1988.

[18] L. McFearin and V. Nair. Control-flow checking using assertions. In *DCCA'5*, pages 103–112. IEEE Computer Society Press, September 1995.

[19] T. Michel, R. Leveugle, and G. Saucier. A new approach to control flow checking without program modification. In *FTCS21*, pages 334–341, 1991.

[20] T. Michel et al. An application specific microprocessor with two-level built-in control flow checking capabilities. In *EURO ASIC'92*, pages 310–313, 1992.

[21] G. Miremadi et al. Two software techniques for on-line error detection. In *FTCS22*, pages 328–335, July 1992.

[22] G. Miremadi et al. Use of time and address signatures for control flow checking. In *DCCS'5*, pages 201–221, September 1995.

[23] V. S. S. Nair et al. Design and evaluation of automated high-level checks for signal processing applications. In *spie advanced algorithms and architectures for signal processing conference*, pages 292–301, August 1996.

[24] M. Namjoo. Techniques for concurrent testing of vlsi processor operation. *Test Conference*, pages 461–468, 1982.

[25] J. Ohlsson and M. Rimen. Implicit signature checking. In *FTCS'25*, pages 218–227, 1995.

[26] J. Ohlsson, M. Rimen, and U. Gunneflo. A study of the effects of transient fault injection into a 32-bit risc with built-in watchdog. In *FTCS'22*, pages 316–325, 1992.

[27] J. Peddersen et al. Rapid Embedded Hardware/Software System Generation. In *In VLSID'05*, pages 111–116, 2005.

[28] B. Ramamurthy and S. Upadhyaya. Watchdog processor-assisted fast recovery in distributed systems. In *Fifth Dependable computing for critical applications*, pages 125–134, 1995.

[29] T. Rao. Error coding for arithmetic processors. 1974.

[30] N. Saxena and E. McCluskey. Control-flow checking using watchdog assists and extended-precision checksums. *IEEE Transactions on Computers*, pages 554–558, 1990.

[31] M. Schuette and J. Shen. Processor control flow monitoring using signatured instruction streams. *IEEE Trans. Computers*, pages 264–276, March 1987.

[32] M. A. Schuette et al. Experimental evaluation of two concurrent error detection schemes. In *FTCS'16*, pages 138–143, July 1986.

[33] J. Sosnowski. Detection of control flow errors using signature and checking instructions. *IEEE International Test Conference*, pages 81–88, 1988.

[34] The PEAS Team. ASIP Meister, Available at http://www.eda-meister.org/asip-meister/, 2002.

[35] S. Upadhyaya and B. Ramamurthy. Concurrent process monitoring with no reference signatures. *IEEE Transactions on Computers*, 43:475–480, April 1994.

[36] K. Wilken and J. Shen. Continuous signature monitoring: low-cost concurrent detection of processor control errors. *Computer-Aided Design of Integrated Circuits and Systems*, pages 629–641, June 1990.

[37] S. S. Yau and F. Chen. An approach to concurrent control flow checking. *IEEE Trans. Software Eng.*, 6(2):126–137, 1980.