A Performance Counter-based Control Flow Checking Technique for Multi-core Processors

Hussien Al-haj Ahmad, Yasser Sedaghat, Mohammadreza Rezaei
Dependable Distributed Embedded Systems (DDEmS) Laboratory
Computer Engineering Department
Ferdowsi University of Mashhad
Mashhad, Iran
hu.alhajahmad@stu.um.ac.ir, y_sedaghat@um.ac.ir, rezaei.mr89@stu.um.ac.ir

Abstract— **Today, both the rapid improvement of process technology and the arrival of new embedded systems with high-performance requirements, have led to making the current trend in processors manufacturing shift from single-core processors to multi-core processors. This trend has raised several challenges for reliability in safety-critical systems that operate in high-risk environments, making them more vulnerable to soft errors. Hence, using additional methods to satisfy the strict system requirements in terms of safety and reliability is unavoidable. In this paper, an efficient hybrid method to detect control flow errors in multi-core processors has been proposed and evaluated. About 36,000 software faults have been injected into three well-known multi-threaded benchmarks at run-time. The experiment results show that the fault coverage is 100%. The results also show that the execution time overhead varies between 31.25% and 51.02%, and the program size overhead varies between 20.23% and 67.64% with respect to the employed benchmark.**

Keywords—multi-core; hardware performance counter; control flow checking; safety-critical systems

I. INTRODUCTION

Recently, the rapid improvements in processors technology and the need to achieve high performance computations have led to produce processors with more cores, so-called multi-core processors [1, 2]. Using these processors has led to improve the performance significantly. Therefore, the improvement in processors manufacturing has also extended to cover the embedded systems that are widely used in several applications such as avionics and automotive control systems [1, 2]. While multi-core trend has become the common trend in processor technology, it has introduced several challenges for reliability in safety-critical embedded systems [1, 3, 4]. Strictly speaking, the advancements in the process technology in tandem with the production of chips comprising millions of transistors have led to making these processors more susceptible to faults [5-8]. Critical embedded systems which work in harsh environments are more vulnerable against faults. Therefore, the use of a faster processor, e.g. multi-core, will not be sufficient to meet the strict system requirements such as safety, real-timeness, and reliability. As a result, several recent projects such as ARTEMIS ACROSS [5], have focused on employing multi-core technology in embedded systems in order to make these systems work efficiently with high level of reliability.

Transient faults, also known as soft errors, pose a major threat to system reliability [3, 5]. When a soft error, e.g. Single Error Upset (SEU), occurs in a safety-critical system, it may lead to disastrous. As a result, they will need to be addressed in the design phase of the system development process [8]. Studies have been shown that transient faults can be classified, in terms of effects, into data errors and control flow errors (CFEs) [3, 6, 8]. Data errors appear when the value of a variable is changed, erroneously. A CFE appears if a program is executed in an abnormal fashion (deviations from the normal execution flow). The experimental results have shown that about 33%–77% of these faults lead to CFEs [9, 10], depending on the type of processors used [11]. Hence, employing additional methods to satisfy the system reliability requirements and detect any unanticipated behavior caused by CFEs as early as possible is unavoidable [10-12].

In this paper, a hybrid, efficient technique to detect CFEs in the modern multi-core processors is proposed. This method takes the advantage of the hardware performance counter, a common feature in the most modern processors, in order to perform the control flow error detection and keep the system reliable. The method proposed in this paper can achieve 100% of control flow error detection, including unexpected interrupts errors, e.g. infinite loops.

The rest of this paper is organized as follows: Section II summarizes the related work on control flow checking methods and highlights the motivations of presenting control flow error detection method for multi-core architectures. Section III discusses the proposed method and the fault model used in this paper. Section IV discusses the evaluation of the proposed method. Finally, conclusion section concludes the paper.

II. RELATED WORK

A CFE may occur in both computers and digital systems [7]. This error may cause by an error occurred in processor registers, such as occurrence of SEU in a bit of the Program Counter (PC) register during the execution of the program, or in system memory. As a result, the program will violate the correct sequence of its control flow and result in incorrect outcomes [13]. Hence, control flow checking (CFC) methods which provide a cost-efficient error detection, are considered an essential need to keep the system reliable [9, 11, 14, 15]. In order to detect CFEs, numerous methods have been proposed in the literature that fall into three broad categories, namely, hardware-based CFC, software-based CFC, and hybrid CFC methods, combining software-based CFC with hardware-based CFC methods [12].

The general approach adopted by the most of the CFC methods is dividing high-level program source code into basic blocks (branch free interval). A basic block (BB) is formed from continuous instructions that run continuously, in the absence of errors, from the first instruction to the last one.

Branches or call instructions are only allowed at the end of BBs. These blocks associate with each other through directed edges that are used to represent the legal jumps. Control flow graph (CFG) is formed by combining a set of BBs with a set of corresponding legal jumps. Extracting the CFG is an essential step in CFC process. Therefore it should extract accurately in which it can reflect the proper control flow of the corresponding program without any limitations [3].

A. *Hardware-based CFC Methods*

Hardware-based CFC methods usually lie on introducing additional, special purpose hardware modules (like a watchdog processor). A watchdog processor is a processing element used for detecting CFEs by monitoring the main processor during running the program. Although these methods introduce a higher fault coverage [3, 8], they impose higher costs on the system. Therefore, These methods are typically considered an appropriate solution when cost is not crucial or reconfiguring hardware architecture is allowed.

B. *Software-Based CFC Methods*

Software-based CFC methods use CFG in tandem with signatures and additional instructions in order to detect any undesirable violations. Examples of such methods are Enhanced Control Flow Checking using Assertions (ECCA) [16], CFC by Software Signature (CFCSS) [14], Control-flow Error Detection through Assertions (CEDAs) [11] and Assertion for CFC (ACFC) [17]. Enhanced Committed Instructions Counting (ECIC) is an error detection method presented in [18] for embedded and real-time systems using commercial off the shelf (COTS) processor. This method exploits the performance monitoring features [19] of a processor in order to detect errors. ECIC is applied to single-threaded benchmarks and the experimental results were shown that the error detection coverage is close to 98.18%. ECIC, however, has involved limitations. It can be applied only in COTS processors that have performance monitoring features and special pins (Event Ticking Pins). ECIC drawbacks have been eliminated by the method proposed in [20]. In [3], a software behavior-based technique is presented to detect CFEs in multi-core architectures. To examine the correctness of sequence execution of the program, a software-based watchdog thread is developed and scheduled to run in parallel with the main process. However, the performance gained by multi-core processors will be adversely affected if a separate core is assigned to run the watchdog thread.

Software CFC methods are low-cost, i.e. it can be implemented entirely in software without any additional hardware elements. In addition, they have the capability of being implemented in applications when using hardware methods are not possible, especially for COTS processors and modern processors equipped with cache memories [5, 6, 15, 21]. However, they suffer from significant problems such as performance degradation due to the redundant instructions inserted to the program.

C. *Hybrid CFC Methods*

Most of the hybrid CFC technique are based on redesign the processor [22], or employing a special hardware module to observe the execution of the main processor [5, 6]. To improve error detection coverage, the hybrid CFC methods take advantages of both hardware and software CFC methods. Since hardware CFC technique can provide a higher fault coverage, hybrid CFC technique, usually, employ hardware for both accelerating the checking and improving the error detection capability. Combining software-based CFC technique with external hardware module may result in higher average of the control flow error detection, e.g. Hybrid Error-Detection Technique Using Assertions (HETA) [5] and [6, 21, 23].

The analysis of three important issues has led to introduce the proposed technique. First, a general trend in process technology is towards multi-core processors, and the safety-critical systems have shifted to use multi-core processors instead of single-core. Second, lack of CFC methods that run in multi-thread environments for detecting CFEs. Third, high memory and performance overhead produced by previous CFC methods are not allowed in real-time safety-critical embedded systems that have tight memory and performance budget. Hence, introducing a new hybrid method to provide full error detection with appropriate overheads, is mandatory.

In this paper, a hybrid CFC method is presented. This method aims to exploit the hardware facilities, i.e. the counters implemented in modern processors in order to detect CFEs. Next section discusses the proposed method in more details.

III. THE PROPOSED METHOD

The proposed method uses the counters in order to detect CFEs. The hardware performance counter, basically, is composed of a small number of special purpose configured registers [19]. Since these registers are built inside a processor (on-chip structures), they are able to collect information about the running programs without affecting the performance. In multi-core processors, these counters are independent of each other in the sense that each core has its own set of counters [24]. The proposed method employs the counters beside redundant software instructions in order to count and check the executed instructions. Also, an interface which provides access to the performance counters is needed. Moreover, this interface should not only offer an ability to get access to the performance counters per-thread but also guarantees that each available counter has been customized for a particular thread.

A. *Fault Model*

CFEs can grouped into three general categories [8, 11]: 1) illegal intra-basic block jumps: they denote incorrect jumps within a block, 2) illegal inter-basic block jumps: they refer to incorrect jumps to other block, and 3) illegal jumps from a basic block to outside the program space. In the proposed method, the program is divided into basic blocks and partition blocks (PBs). PBs contain extra software instructions that are added at compile time to detect any violation affects the program control flow. The PB resides between two consecutive BBs. As shown in Fig. 1 (a), the three illegal jumps mentioned above will lead to raise nine possible types of CFEs: Type 1: from a BB to the beginning of another BB, Type 2: from a BB to any point in another BB, Type 3: from a BB to any point in a PB, Type 4: from the end of a PB to the beginning of another

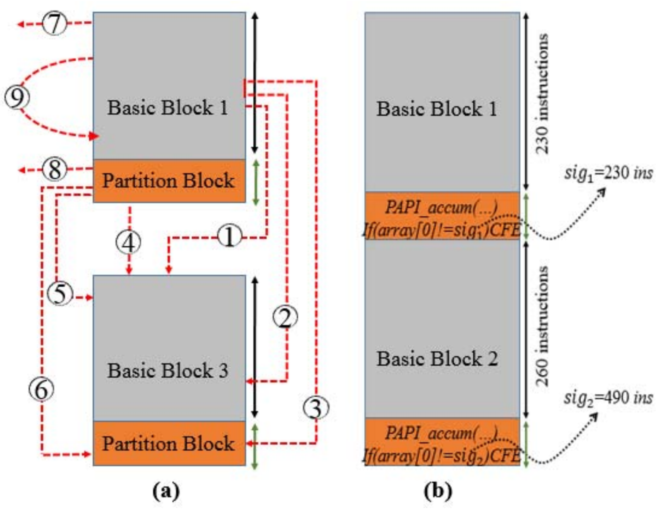

Fig 1. (a) Fault Model, (b) CFE-tolerant basick block structure

BB (legal branch but invalid), Type 5: from a PB to any point in another BB, Type 6: from a PB to any point in another PB, Type 7: from a BB to outside the program memory space, Type 8: from a PB to outside the program memory space, Type 9: from a BB to any point in the same BB.

Type 7 and 8 are almost detected by the operating systems as a segmentation fault [3, 9, 15]. Error type 4 is often occurs due to data errors. As illustrated early, transient faults in processors may result in CFEs and data errors. This paper treats CFEs and not data errors. Most of CFC methods are not able to detect control flow error represented by Type 7, 8 and the illegal jumps to the same basic block, e.g. Type 9. The CFC methods, like software-based CFC methods, in order to detect such errors, insert extra instructions inside basic blocks. While using this scheme enables to detect intra-block CFEs, it may lead to comparative more performance overhead. The method proposed in this paper addresses 9-types CFEs shown in Fig 1. (a) as will be illustrated in the next section.

B. *Mechanism for the proposed method*

The hardware performance provides counters for counting micro-architecture events like clock cycles, branch misses and executed instructions [19]. As counters is an on-chip hardware built inside a processor, there is very limited overhead in counting such events. Some CFC methods proposed in the literature, count the executed instructions inside a basic block using inserted software instructions to detect any CFEs. This scheme tends to be ineffective due to performance overhead. Counting instruction by hardware performance counters in order to detect CFEs will reduce the negative impact of extra instructions on performance.

Whenever the selected event occurs during the execution of the program, the processor increments the respective event counter by one. The event selected in the proposed method is the number of executed instructions. In order to detect control flow errors in multi-core processor environments, the source code of a program is divided into basic blocks and a control flow graph is extracted for each thread from the source code (per-thread CFG). Each per-thread basic block is assigned a signature, "*sig"*, at compile time. The "*sig"* is a variable that holds a value, which in the absence of errors, should be equal to the counter value. Obtaining an accurate number of executed instruction at compile time is hard. So, the value of "*sig"* variable is obtained in two phases. First, the "*sig"* is assigned an arbitrary value. Second, the program is compiled and executed. Using the GNU Project Debugger (GDB), the number of executed instructions is obtained accurately (by adding a breakpoint at a specific place in the code and retrieving the counter value). After that, the old value of "*sig"* variable is updated to the value obtained at run-time. Then, the program is compiled again. It is important to note that per-thread CFG is extracted from the source code and the value assigned to the "*sig"* variable was obtained after executing the program. Function call instructions can appear at the end of BB and the body of function will be handled as a separate CFG. Thus, the functions are a part of BBs of the main CFG.

Additional instructions, also, are inserted and located at the end of each per-thread basic block. The structure of the CFE-tolerant basic block (after inserting the redundant instructions to detect CFEs), is shown in Fig. 1 (b). These instructions consist of two types (in loop basic block, there are three types of the added instructions): retrieving the value of the counter and "*test"* instruction. The task of "*test"* instruction is to confirm that the current running basic block is executed correctly (all its instructions are executed sequentially without any violations).

PAPI (Performance Application Programming Interface) provides a platform, operating system and machine, independent access to the hardware performance counters [24]. It provides a set of functions for retrieving the counter value for specific events. "*PAPI_accum"* function is used to retrieve and accumulate the counters value. The counters are zeroed and returned to counting after completing the operation. When the control of the program reaches the PB, which contains "*test"* instructions, the per-thread "*array"* (a variable defined to hold the counter value) will store the counter content. In the absence of errors, the value stored in the "*array*" should be equal the "*sig*". Otherwise, an error will occur in the program and it will stop.

Basic block that is a header of a loop requires one additional instruction (in addition to the two prior instructions). When the control flow returns back to the header of loop BB, the performance counter, due to the use of "*PAPI_accum*", will be accumulated making the "*test"* instruction interprets the counter value as an error. The third instruction named "*set"* is introduced to solve this problem. Having checked the "*array*" under "*test"* instruction, "*set"* instruction sets the value of the "*array*" to non-zero value in order to avoid any false CFE detection. Overhead induced by "*set"* instruction is negligible since it is limited to BBs which are the header of loops.

If error detection latency (EDL) —the interval between fault activation and error detection [11]— is not critical, then the instructions for checking CFE can be delayed. Meaning that instead of inserting them at the end of each basic block, these instructions can be added to critical basic blocks (basic blocks where it is important to check for any violation in the control flow of the program). This can result in reducing both the memory and performance overhead.

C. *False Accumulation*

It is straightforward that any faulty behavior induced by skipping some instructions can be detected when the program control reaches "*test*" instructions. However, an illegal jump from a basic block *BBi* to a specific place in another basic block *BBj*, in the same thread, can lead to false accumulation error if and only if the following conditions are met:

- Number of executed instructions in *BBi* + number of instructions that will be executed in *BBj = sigj*.
- *BBj ∉ suc(BBi)*,

Where *suc(BBi)* is a set of successors of *BBi*, and *sigj* is the expected value of counter at end of *BBj*. Hence, the false accumulation can occur, resulting in undetected CFE. However, the probability of false accumulation is close to zero.

D. *Optimization Issues*

The performance is a major issue for embedded systems [25]. High overhead in both memory and time execution of a program raising from additional instruction inserted inside the loop region, make the employed CFC technique inappropriate. For this reason, these instructions are inserted outside the loop region when it is possible to move them without any effect on the final result. The desirable objective of this scheme is to prevent the performance degradation resulting from repeated execution of "*test*" and *"PAPI_accum"* instructions during each loop iteration.

However, this scheme seems to be not suitable for event-controlled loops or loops that have in their body conditional statements (*if-then* statements). In this case, the proposed method is forced to move the additional instructions inside the loop region. However, this will increase performance overhead. To reduce performance overhead, loop unrolling technique [25], as a traditional compiler optimization method, is involved in the proposed method. This technique aims to reduce instructions that control the loop, such as "end of loop" tests on each iteration, by rewriting loop body. Thereby increasing speed of a program. Thanks to the loop unrolling mechanism, inappropriate performance overhead can be decreased to an acceptable value. The authors of [25] investigated the impact of employing loop unrolling on control flow reliability. The results of this investigation showed that a significant fault coverage with acceptable overhead in memory and performance can be achieved. The proposed method can benefit from the loop unrolling technique with a huge positive impact on the execution time of the program. The impact of the loop unrolling technique on both memory and performance overheads is investigated in the next section.

IV. EVALUATION OF THE PROPOSED TECHNIQUE

The proposed technique has been tested in two phases, at the first phase the proposed technique have been implemented and then assessed in the presence of injected faults. The second phase challenged the methods efficiency considering memory and performance overheads.

To assess the proposed method, a quad-core processor system with shared memory, running Ubuntu Linux OS release 14.0.1 is used. This processor was selected due to it supports hardware performance counter and multi-threaded programs. Also, three well-known multi-threaded benchmarks are employed. These benchmarks are Matrix Multiplication (MM), Insertion Sort (IS) and Quick Sort (QS)

A. *Fault Injection Experimental Results*

Several studies have clarified that a transient error in the PC register can lead to CFEs. Thus, the fault model used in the experiments is the bit flip fault applied to the PC register. In addition, a sophisticated Software Implemented Fault Injection (SWIFI) method based on GDB is used to perform fault injection at run-time without changing the number of instructions executed. About 36,000 faults are injected into the benchmarks. These injected faults will affect the program and can result in different cases:

- CR (Correct Result): the injected fault does not change the control flow of the program, i.e. the final result is produced, correctly.
- OS (Operating System): the injected faults lead to deviation the running of the program and produce exceptions like a segmentation fault which, mostly, detected by the OS.
- IR (Incorrect Result): the injected fault changes the final result and leads to a wrong output.
- TO (Time Out): the injected fault modifies the program execution time (such as entering in infinite loops).
- ED (Error Detection): the injected fault is detected by the test instructions that are inserted into each BB for control flow checking.

The occurrences of unexpected interrupts or illegal infinite loops (endless loops), is a major limitation of most previous CFC methods. In order to detect infinite loops, the proposed method exploits the PAPI feature named "*overflow*". "*PAPI_overflow*" is a low-level function provided by PAPI which enables to call user-defined handlers when an overflow, i.e. exceeding a predefined threshold, occurs. Based on the foregoing, a hardware performance counter to count the clock cycles needed by the program being run is established, and "*PAPI_overflow*" is used. Whenever overflow occurs, the handler will be called and infinite loop error, or unexpected interrupts, is reported.

Fault injection results are presented in TABLE I. It shows the number of injected faults, the occurrence of IR, TO, CR, and ED. As is clear from this table, the number of faults that caused an error in the used benchmarks are fully detected. Thus, the fault coverage of the proposed method is 100%. It should be mentioned that not all of the injected faults lead to faulty results because some of them are detected by the operating system.

B. *METHOD EFFICIENCY*

Higher fault coverage and overhead are not sufficient to compare two different CFC methods because they are not able to reflect the effectiveness of the proposed method, accurately. In other words, there is a trade-off between three major parameters: fault coverage, performance overhead and memory overhead. Therefore the method adopted must provide an appropriate balance between them as much as possible.

TABLE I. FAULT INJECTION RESULTS

Benchmark	Fault Injected[a]	IR	TO	Faulty Results (IR+TO)	CR	ED
MM	12,000	772	7080	7852	0	7852
QS	12,000	429	3771	4200	612	4200
IS	12,000	1632	1521	3153	1719	3153

[a] Not all the injected faults will lead to faulty results, some of them may result in program crash.

As similar as some previous works [3, 8, 11], a metric named "*Evaluation Factor*" is introduced in this paper. *"Evaluation Factor"* takes the formula described below:

$$Evaluation\ Factor = \frac{Fault\ Coverage}{Memory\ Overhead \times Performance\ Overhead} \quad (1)$$

According to (1), the *"Evaluation Factor"* has a direct relationship with "*Fault Coverage*" and a reverse relationship with "*Memory Overhead*" and "*Performance Overhead*". "*Fault Coverage*" expresses the ability of the proposed method to detect CFEs. While "*Memory Overhead*" and "*Performance Overhead*" expresses the overhead incurred by additional instructions. "*Fault Coverage*" is calculated using (2), where ED denotes the number of detected CFEs and "*Faulty Results*" denotes the number of incorrect results plus the number of timeout errors induced by fault injection process.

$$Fault\ Coverage = \frac{ED}{Faulty\ Results} \times 100 \quad (2)$$

In order to compute "*Performance Overhead*", the proposed technique uses the difference between time execution before and after inserting redundant instructions. Equation 3 explains:

$$Performance\ Overhead = \frac{EtTp - EtnonTp}{EtnonTp} \times 100 \quad (3)$$

Where *EtTp* denotes the execution time of the CFE-tolerant program, *EtnonTp* denotes the execution time of the target program (program without CFC instructions). As for the "*Memory Overhead*" resulted from inserting CFC instructions to the target program, the difference between the size of the CFE-tolerant program and the target program is used. As (4) describes:

$$Memory\ Overhead = \frac{MTp - MnonTp}{MnonTp} \times 100 \quad (4)$$

Where *MTp* denotes the program size of the CFE-tolerant program, and *MnonTp* denotes the size of the target program.

In the next subsection, the paper will discuss the performance and memory overheads incurred by applying the proposed technique. In addition, it will show how the proposed technique can benefit from the loop unrolling technique in order to reduce the performance overhead.

C. Performance Evaluation

To quantify the performance and memory overheads, the employed benchmarks are executed without any extra code. The execution results with regard to execution time (*ET*) and size of program code (*SC*) are recorded. Then, the CFC proposed technique is applied to these benchmarks in two different schemes (With-CFC, and With-CFC+LU).

TABLE II summarizes the percentage of the performance overhead (*PO*), and the memory overhead (*MO)* incurred by applying the proposed method (both without and with loop unrolling technique) to three different benchmarks. As is evident from TABLE II, the results show that the proposed method incurs performance overhead varies between 31.25% and 51.02%, and memory overhead varies between 20.23% and 67.64% for different benchmarks, without using loop unrolling technique.

Employing loop unrolling technique will result in lower performance overhead, but this is at the expense of memory overhead. However, this overhead can be controlled by unrolling the most time-consuming loop "*n"* times. The "*n"* variable so-called loop unrolling factor denotes the number of times for unrolling the loop that consumes a lot of time until a desired tradeoff between performance and memory overheads is realized. Therefore, a new equation that enables a user to achieve the tradeoff is introduced. Loop unrolling leads to memory overhead due to repeating the loop region "*n"* times. Therefore, the extra size overhead "*Extra_Size Overhead*" results from loop unrolling is:

$$Extra\_Size\ Overhead = \sum_{i=1}^{m} \frac{SUL_i - SOL_i}{SOL_i} \quad (5)$$

Where "*SUl$_i$*" denotes the size of loop after unrolling it. "*SOl$_i$*" denotes the original size of loop "*l$_i$*". Variable "*m"* denotes the number of loops in the program. "*SUl$_i$*" can be written as:

$$SUL_i = n_{l_i} \times SOL_i \quad (6)$$

"nli" denotes the loop unrolling factor of the loop "*l$_i$*". Therefore, with respect to (5) and (6):

$$Extra\_Size\ Overhead = \sum_{i=1}^{m} (n_{l_i} - 1) \times SOL_i \quad (7)$$

Moreover, the new program size "*New_Program Size*" will be calculated as follow:

$$New\_Program\ Size = Original\ Program\ Siz + Extra\_Size\ Overhead \quad (8)$$

Where "*Original Program Size"* denotes the size of the program without applying loop unrolling optimization. Therefore the "*Memory Overhead"* induced by the loop unrolling will be calculated as follow:

$$Memory\ Overhead = \frac{\sum_{i=1}^{m} (n_{l_i} - 1) \times SOL_i}{Original\ Program\ Size} \times 100 \quad (9)$$

In addition, the "*Performance Overhead*" induced by the loop unrolling will be as follow:

$$Performance\ Overhead = \frac{ETNP - ETOP}{ETOP} \times 100 \quad (10)$$

Where *"ETNP*" refers to the execution time of the new program produced after applying the loop unrolling. The *"ETOP*" refers to the execution time of the original program. CFC methods such as [3], [8], [11] and [20] incur "*Performance Overhead*" (in average) ~10%, ~41%, ~30%, ~50% respectively. Based on the results presented in TABLE-II, the "*Performance Overhead*" using Loop-unrolling is reduced to 27.44% in average and the "*Memory Overhead*" is about 57.1%. Based on (1) and the presented results in both TABLE I and TABLE II, the "*Evaluation Factor"* of the

TABLE II. OVERHEAD CHARACTERISTICS

	Multithread Benchmarks			
	Parameters	MM	IS	QS
Without-CFC	***ET***	3914	129	160
	SC	5.98	3.40	6.02
With-CFC	***ET***	5911	191	210
	SC	7.19	5.70	9.31
	PO	51.02%	48.06%	31.25%
	MO	20.23%	67.64%	54.65%
With-CFC+LU	***ET***	5104	171	191
	SC	7.97	6.21	9.55
	PO	30.40%	32.55%	19.37%
	MO	33.27%	80.34%	58.16%

[a] Without-CFC is the target program without extra instructions
[b] With-CFC is the modified program after inserted extra instructions.
[c] With-CFC+LU is "With-CFC" in tandem with loop unrolling
[d] ET denotes the program execution time (in milliseconds)
[e] SC denotes the size of the program code (in kilobytes)
[f] PO denotes the performance overhead
[g] MO denotes the Memory overhead

proposed method using loop unrolling technique is about 50 (with 100% error detection).

V. CONCLUSION

This paper presents a hybrid CFC method for multi-threaded programs running on multi-core processors. The proposed method takes advantage of the hardware performance counter common feature in modern processors to perform CFE detection and keep system reliable. The distinctive advantages of this method over previous CFC methods are the ability to detect unexpected interrupts errors. Experimental results showed that using the proposed technique, 100% of CFEs could be detected. It is important to note that the proposed method is more portable than other hybrid CFC methods. It does not require any modifications to the processor or using extra hardware monitoring module to observe the execution of the main processor, i.e. the extra hardware overhead is zero. This method requires only that the target multi-core processor has a hardware performance counter feature. However, most modern processors contain hardware performance counters.

REFERENCES

[1] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. de Dinechin, "The shift to multicores in real-time and safety-critical systems," International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), IEEE, pp. 220-229, 2015.

[2] F. Reichenbach and A. Wold, "Multi-core Technology--Next Evolution Step in Safety Critical Systems for Industrial Applications?," 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD),IEEE, pp. 339-346, 2010.

[3] M. Maghsoudloo, H. R. Zarandi, and N. Khoshavi, "An efficient adaptive software-implemented technique to detect control-flow errors in multi-core architectures," Journal of Microelectronics Reliability, Elsevier, vol. 52, Issue. 11, pp. 2812-2828, 2012.

[4] C. El Salloum, M. Elshuber, O. Höftberger, H. Isakovic, and A. Wasicek, "The ACROSS MPSoC–A new generation of multi-core processors designed for safety–critical embedded systems," Microprocessors and Microsystems, vol. 37, Issue. 8, pp. 1020-1032, 2013.

[5] J. R. Azambuja, M. Altieri, J. Becker, and F. L. Kastensmidt, "HETA: Hybrid error-detection technique using assertions," IEEE Transactions on Nuclear Science, vol. 60, pp. 2805-2812, 2013.

[6] L. Parra, A. Lindoso, M. Portela-Garcia, L. Entrena, B. Du, M. S. Reorda, and L. Sterpone, "A new hybrid nonintrusive error-detection technique using dual control-flow monitoring," IEEE Transactions on Nuclear Science, vol. 61, Issue. 6, pp. 3236-3243, 2014.

[7] M. Fazeli, R. Farivar, and S. G. Miremadi, "Error detection enhancement in powerpc architecture-based embedded processors," Journal of Electronic Testing, vol. 24, Issue. 1-3, pp. 21-33, 2008.

[8] S. A. Asghari, H. Taheri, H. Pedram, and O. Kaynak, "Software-based control flow checking against transient faults in industrial environments," IEEE Transactions on Industrial Informatics, vol. 10, Issue. 1, pp. 481-490, 2014.

[9] R. Vemu, S. Gurumurthy, and J. A. Abraham, "ACCE: Automatic correction of control-flow errors," IEEE International Test Conference,IEEE, pp. 1-10, 2007.

[10] Y. Sedaghat, S. G. Miremadi, and M. Fazeli, "A software-based error detection technique using encoded signatures," 21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems,IEEE, pp. 389-400, 2006.

[11] R. Vemu and J. Abraham, "Ceda: Control-flow error detection using assertions," IEEE Transactions on Computers, vol. 60, Issue. 9, pp. 1233-1245, 2011.

[12] R. G. Ragel and S. Parameswaran, "A hybrid hardware--software technique to improve reliability in embedded processors," ACM Transactions on Embedded Computing Systems (TECS), vol. 10, Issue. 3, p. 36, 2011.

[13] A. Chaudhari, J. Park, and J. Abraham, "A framework for low overhead hardware based run-time control flow error detection and recovery," VLSI Test Symposium (VTS), IEEE 31st, IEEE, pp. 1-6, 2013.

[14] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," IEEE transactions on Reliability, vol. 51, Issue. 1, pp. 111-122, 2002.

[15] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Improved software-based processor control-flow errors detection technique," Annual Reliability and Maintainability Symposium, IEEE, pp. 583-589, 2005.

[16] Z. Alkhalifa, V. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," IEEE Transactions on Parallel and Distributed Systems, vol. 10, Issue. 6, pp. 627-641, 1999.

[17] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," 9th IEEE On-Line Testing Symposium (IOLTS), IEEE, pp. 137-143, 2003.

[18] A. Rajabzadeh and S. G. Miremadi, "Transient detection in COTS processors using software approach," Microelectronics Reliability, vol. 46, Issue. 1, pp. 124-133, 2006.

[19] P. THIS, "The basics of performance-monitoring hardware," IEEE Micro, vol. 22, Issue. 4, pp. 64 - 71, 2002.

[20] A. Rajabzadeh and S. G. Miremadi, "Feature Specific Control Flow Checking in COTS-Based Embedded Systems," Third International Conference on Dependability (DEPEND),IEEE, pp. 58-63, 2010.

[21] L. Parra, A. Lindoso, M. Portela, L. Entrena, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martínez-Álvarez, "Efficient mitigation of data and control flow errors in microprocessors," IEEE Transactions on Nuclear Science, vol. 61, Issue. 4, pp. 1590-1596, 2014.

[22] S. Cuenca-Asensi, A. Martinez-Alvarez, F. Restrepo-Calle, F. R. Palomo, H. Guzman-Miranda, and M. A. Aguirre, "A novel co-design approach for soft errors mitigation in embedded systems," IEEE Transactions on Nuclear Science, vol. 58, Issue. 3, pp. 1059-1065, 2011.

[23] P. Bernardi, L. Sterpone, M. Violante, and M. Portela-Garcia, "Hybrid fault detection technique: A case study on virtex-II Pro's PowerPC 405," IEEE Transactions on Nuclear Science, vol. 53, Issue. 6, pp. 3550-3557, 2006.

[24] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," Proceedings of the department of defense HPCMP users group conference, pp. 7-10, 1999.

[25] G. Nazarian, L. Carro, and G. N. Gaydadjiev, "Towards Code Safety with High Performance," International Conference on Architecture of Computing Systems,Springer, pp. 209-220, 2014.