FACTA UNIVERSITATIS
Series: **Electronics and Energetics** Vol. 29, N^{o} 2, June 2016, pp. 243 - 260
DOI: 10.2298/FUEE1602243M

PARALLEL EXECUTION TRACING: AN ALTERNATIVE SOLUTION TO EXPLOIT UNDER-UTILIZED RESOURCES IN MULTI-CORE ARCHITECTURES FOR CONTROL-FLOW CHECKING

Mohammad Maghsoudloo, Hamid R. Zarandi

Amirkabir University of Technology (Tehran Polytechnic), Iran

Abstract. *In this paper, a software behavior-based technique is presented to detect control-flow errors in multi-core architectures. The analysis of a key point leads to introduction of the proposed technique: employing under-utilized CPU resources in multi-core processors to check the execution flow of the programs concurrently and in parallel with the main executions. To evaluate the proposed technique, a quad-core processor system was used as the simulation environment, and the behavior of SPEC CPU2006 benchmarks were studied as the target to compare with conventional techniques. The experimental results, with regard to both detection coverage and performance overhead, demonstrate that on average, about 94% of the control-flow errors can be detected by the proposed technique, with less performance overhead compared to previous techniques.*

Key words: *On-line error detection, control-flow error, error detection coverage, multi-core processor, CPU utilization.*

1. INTRODUCTION

Advances in CMOS technology have provided reduction in transistor size and voltage levels [1]. As the number of available transistors continues to grow exponentially, adding and using more hardware resources has emerged as a convenient solution to increase the performance of the microprocessors. While these additional resources yield performance enhancements, they lead to increases in the level of power dissipation [2]. In order to improve the performance per cost ratio of processors, computer architects are forced to shift from single-core single-threaded processors to multi-core multi-threaded processors or Chip Multi-Processors (CMPs) [2], [3].

However, current trends in computer architectures have shown that the forthcoming of new processors will involve new challenges related to increasing the vulnerability of them against hardware transient faults [4]. Most soft errors are the result of energetic particle

Received April 6, 2015; received in revised form December 27, 2015
Corresponding author: Mohammad Maghsoudloo
Amirkabir University of Technology (Tehran Polytechnic), 424 Hafez Ave, Iran
(e-mail: m.maghsoudloo@aut.ac.ir)

strikes, induced by high-energy neutrons from cosmic rays, and alpha particles from decaying radioactive impurities in packaging and interconnect materials [5].

Soft errors, that occur in both system memory and combinational logic of the computer system, will need to be addressed during the design phase of the systems, especially for safety-critical applications [2]. To counter the errors, present in different level of memories, designers typically employ redundant information or hardware, such as error correcting codes (ECCs), and bit interleaving to protect memories.

Similarly, combinational logic within the processor should be protected. Therefore, high-availability systems need much more hardware redundancy than that provided by ECC and parity bits. For example, IBM has historically added 20-30% of additional logic within its mainframe processors for fault tolerance [6]. There have been several approaches to add extra hardware redundancy and modification for tolerating hardware faults or detecting soft errors in CMPs, called HardWare Fault Tolerant techniques (HWFT), such as Core Cannibalization Architecture (CCA) [7], Core Salvaging [8], Mixed-Mode Multi-core (MMM) [9], which imposes more power consumption, and complexity challenges [10]. The inclusion of redundant hardware design and modification may negatively impact the design cycles of systems and also area- and power-efficiency of the new and modern processor [11], [12], [13].

On the other hand, SoftWare Fault Tolerant techniques (SWFT) usually provide adequate levels of fault tolerance, hence system reliability [14], such as mSWAT [1], and Detouring [11]. Despite the flexibility of SWFT techniques which can moderate the negative impacts of HWFT ones, they can lead to huge and considerable performance degradation during the execution, because of the nature of their structures which are based on execution replication [10].

In the other category, Control-Flow Checking (CFC) methods, key methods used for monitoring the behavior of a program, have shown to provide effective and cost-efficient error detection coverage [15], [16], [17], [18], [20]. Unfortunately, due to the crucial drawbacks of the hardware-based methods, Hardware-based CFC (HCFC) methods are not considered to be appropriate for current architectures [23]. Moreover, the structure of conventional Software-based CFC (SCFC) techniques has not been adapted with the inherent features of CMPs. Although using the SCFC techniques can reduce the huge performance overhead of the software-based methods, however, these techniques still have potential of imposing high performance overhead (due to inserting some instructions into the programs) on the systems that would undermine the obtained benefits of the modern processors and would waste the efforts taken by designers to improve the execution time of the programs and utilization of the processors. Typically, the performance overhead of these techniques grows to 40% for only detection, and also grows to more than 100% for detection and correction [18] that would be intolerable in current architectures and applications.

So, one effective way for enhancing the reliability of CMPs, is to modify the configuration of software-centric methods, so that the adaptability with the status and load of the CPU become visible in their structure. This paper presents an adaptive and efficient SCFC technique which takes the advantages of multi-core multi-threaded processors, such as parallel execution capabilities, to remove the main drawbacks of the conventional SCFC techniques [21], [22].

Shifting from single-core to multi-core processor means that programmers must write concurrent multi-threaded programs for the optimal use of hardware capacities. Unfortunately, some challenges, such as load balancing, sequential and synchronization dependencies,

cause that parallelism is growing slowly and, the resources in multi-core processors may not be employed properly [19], [28], [29]. So, there are some idle cycles and resources during the execution of each core, which is noticeable when the number of cores increases. Therefore, the availability of idle hardware resources in current processors has motivated us to use under-utilized CPU capacities for employing a technique for parallel and concurrent execution tracing which is compatible with the features of multi-core multi-threaded processors. Using this idea will lead to moderation of the negative effects of the serial control-flow checking, proposed by the previous techniques, that causes undesirable performance and memory overheads.

To evaluate the proposed technique, eleven well-known programs in the SPEC CPU2006 Benchmark suite [30] were utilized. These benchmarks were run on a quad-core processor system, Intel® core™ i7-740QM with 6.00 GB Memory RAM, with a real operating system (Red Hat Enterprise Linux AS release 10). The results of injecting about 12000 errors reveal that on average, about 94% of the CFEs were detected by the proposed technique. Moreover, based on a comparison metric, which considers the effects of methods in CFE detection coverage and performance overheads, the proposed technique is found to have higher coverage with lower overhead compared to the previous works.

The structure of this paper is as follows: section 2 introduces terminology. Section 3 describes the main problem and motivation. The structure of the proposed technique is explained in Section 4. The simulation environment and the experimental results are shown in Section 5. Finally, section 6 concludes the paper.

2. BACKGROUND

All of the hardware techniques have been traditionally considered to achieve reliability requirements. However, it is not feasible for those where cost is a critical issue. The use of Commercial Off-The-Shelf (COTS) components for safety-critical applications has been suggested to accelerate the development cycle and produce cost effective systems. COTS components require specific approaches to take into account the effect of possible hardware faults [17].

Therefore, SWFT techniques are recently preferred. Re-execution of the program in different level of the code (thread, process, or instruction) is the basis of the most SWFT techniques to detect faults. However, these methods require around 100% performance overhead, which may not be suitable for most of the current applications and systems [11], [12], [13].

Consequently, the need for state-of-the-art high computing performances, coupled with cost containment, provides a strong motivation for investigating feasible alternatives to traditional solutions [17]. Transient or intermittent faults induced in hardware have an impact on software running on it, which is either data error or control flow error. Control-flow errors occur when a processor jumps to an incorrect next instruction, which have been demonstrated to account for more than 70% of all errors [17]. So, due to the incidence and importance of control-flow errors and the advantages of software-based techniques, using the SCFC techniques for detecting control-flow errors has been shown as an effective and low-cost alternative solution to enhance the reliability of the processors. Moreover, previous studies on the effects of Multiple Bit Upset (MBU), the new types of errors, arisen due to shrinking the feature size of transistors, has demonstrated that the probability of occurrence of control-flow errors has been recently increased compared to the past [31].

Table 1 summarizes the features of different methods for enhancing the reliability of the CMPs. As Table 1 demonstrates, if the problem of high performance degradation, imposed by the SCFC techniques, can be moderated, the SCFC techniques will be more compatible with the features of the modern processors, compared to the other types of the proposed methods.

Table 1 A general comparison among ifferent methods for enhancing the reliability of CMPS

Categories	Detecting hardware faults/errors		Need program modification	Need hardware modification	Performance degradation
	In memory	*In combinational logic*			
HWFT	Most	Almost all	No	Yes	Low
ECC	Almost all	None	No	Yes	Low
HWFT+ECC	Almost all	Almost all	No	Yes	High
HCFC	Most	Most	No	Yes	Low
SWFT	Most	Most	Yes	No	Very high
SCFC	Most	Most	Yes	No	High

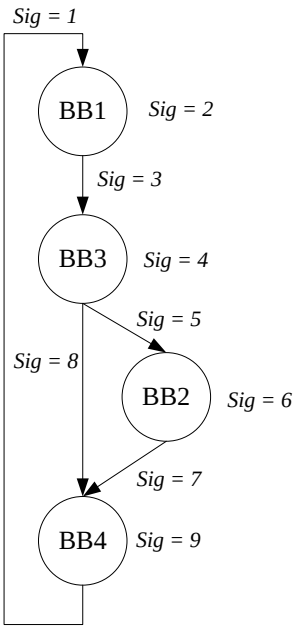
3. PROBLEM AND MOTIVATION

In the general case, almost all of the previous SCFC techniques are based on inserting some instructions into the program code. These instructions are responsible for monitoring the flow of the program execution and detecting any violation of run-time behavior. In fact, the fault model, considered in the paper, is the transient faults which lead to illegal sequence of basic blocks execution and control flow errors.

Extracting the Control Flow Graph (CFG) from relations among basic blocks of a program code is always considered a prerequisite step in both of SCFC and HCFC methods. Any incorrectness and limitation in capturing the control dependencies among nodes of the CFG causes that the flow of a given program will not be precisely followed in checking phase. As Fig. 1(a) shows, after determining control dependencies among basic blocks of the program, each node of the CFG should be labeled by a unique signature, and to store the values of run-time signatures, a global variable or register should be reserved (for example variable *Sig* in Fig. 1(a)).

After assigning signatures to each basic block (BB), types and locations of the checking and updating instructions should be specified. Fig. 1(b) demonstrates the method, used by the conventional techniques, for locating and specifying additional statements in the program code. Most of the previous SCFC techniques added these instructions at the beginning and/or at the end of the BBs in order to achieve high detection coverage. The sequence of the signatures is checked at run-time by conditional branch instructions. These instructions compare the value of the run-time signature with the pre-defined value assigned to each block at the design or compile time in order to detect any misbehavior. The run-time signatures should be updated, so that after checking instructions they confirm the correct execution. The major differences among previous related methods are in the content of updating statements. For example, the Control-Flow Checking by Software Signatures (CFCSS) technique [16] computes the signature of the destination blocks from the signature of the source block by implementing the XOR functions between the signature of the current node and the destination node. In addition, the Yet Another Control-Flow Checking using Assertions (YACCA) technique [17] defines a set of instructions that updates the

signature using AND operation and XOR function with two constant masks. Finally, the Control-Flow Error Detection through Assertions (CEDA) technique [15] inserts fewer instructions (only a XOR operation) than the previous works by calculating signatures differently. Moreover, this technique can uniquely identify the node from which the illegal jump occurred, because for eventual CFE correction, stronger requirements are required. None of the above mentioned techniques can be used here as proposed. Therefore, this technique has been shown to be more efficient and effective compared to the prior methods [15], [18]. Fig. 1(b) shows three typical basic blocks of a program with the added instructions, specified according to CEDA. Due to the advantages of the CEDA, this technique is selected as the most efficient previous SCFC techniques to compare with the proposed method, in the rest of this paper.



(a)

```
.L1:  Br    Sig != 0001, Err
      movl  Sig, Sig Xor 0011
      pushl %ebp
      movl  %esp, %ebp
      pushl %edi
      pushl %esi
      subl  $44, %esp
      movl  $0, -28(%ebp)
      movl  $0, -36(%ebp)
      Br    Sig != 0010, Err
      movl  Sig, Sig Xor 0001
      jmp   .L3
.L2:  Br    Sig != 0101, Err
      movl  Sig, Sig Xor 0011
      movl  $0, -32(%ebp)
      addl  %eax, %eax
      addl  %edx, %eax
      addl  %edi, %eax
      Br    Sig != 0110, Err
      movl  Sig, Sig Xor 0001
      jmp   .L4
.L3:  Br    Sig != 0011, Err
      movl  Sig, Sig Xor 0111
      movl  $0, -28(%ebp)
      addl  %eax, %eax
      addl  %edx, %eax
      addl  %edi, %eax
      cmpl  $2, -22(%ebp)
      addl  $1, -30(%ebp)
      Br    Sig != 0100, Err
      movl  Sig, Sig Xor 0001
      jle   .L2
```

(b)

Fig. 1 The conventional algorithm for (a): assigning signatures, (b): locating the added instructions, and specifying types of the added instructions

3.1. The problem

Unfortunately, due to the execution of checking and updating statements, added at the beginning and at the end of each BB, applying previous SCFC techniques on the current processors will lead to high performance degradation. The conditional branch instructions, used as the usual checking statements by the previous SCFC techniques, take two clock cycles; and the XOR operation, used for updating the values of run-time signature, takes one clock cycle. So, the negative impacts of the checking statements on the execution time are more than the updating ones. On average, about 75% of performance degradation due to the previous SCFC techniques is imposed just because of executing the checking statements. This drawback of the SCFC techniques can undermine the obtained benefits of the modern processors and would waste the efforts taken by designers to improve the execution time of the programs and utilization of the processors.

3.2. The motivation

In order to reduce the negative impacts of the checking instructions, the features of HCFC techniques can be modeled. The HCFC techniques check the behavior of the main processor in parallel with the execution of the main program [24], [25]. In general case, an external checker or watchdog processor is responsible to check the sequence of the signatures of the main program, or the accesses of the main processor to the memory for detecting CFEs during the execution. Due to the reasons mentioned in the previous section, although HCFC techniques can moderate the main drawback (performance overhead) of SCFC techniques, these techniques require some modifications in hardware, which make it impossible to be used in COTS and current processors.

The one effective way to take the advantages of parallel CFC in current architectures (without adding any hardware redundancy) is to use the under-utilized CPU capacities in multi-core processors. Utilization is the percentage of time that a component is actually occupied, compared to the total time that the component is available for use [10], [19], [29]. Unfortunately, developing parallel software for shared-memory multi-cores, using today's programming languages, can be challenging. Therefore, due to the lack of high-level parallelization constructs, multi-core processors are not being fully leveraged [19], [29].

Fig. 2 shows the average percent of CPU usage of the system for three real applications from SPEC CPU2006 benchmarks suite when running on a real quad-core system (the execution times of applications in this figure are not real). In the example, the cores are only executing the designated application. The results show that since parallelism is not thoroughly possible, the CPU is not fully occupied during the execution of these applications (even when the number of running programs grows to 3) which becomes worse when the number of cores in the system increases. Therefore, there are enough resources in the system to run CFC mechanism during the idle cycles of CPUs.

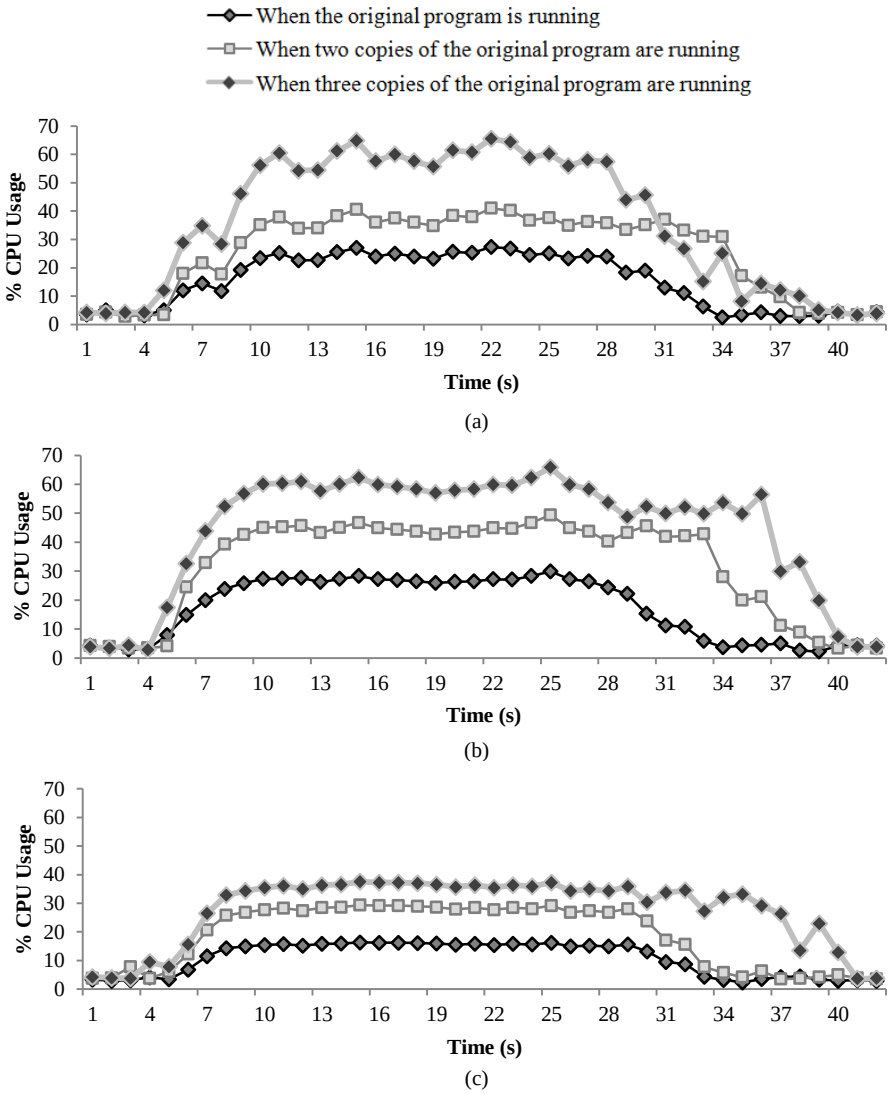


Fig. 2 Percentage of CPU usage for three benchmarks: (a): bzip2, (b): bzip2*, (c): mcf

4. The Proposed Technique for Parallel SCFC

The idea, for exploiting under-utilized resources in multi-cores, is to develop a watchdog thread, which is responsible for checking the sequence of signatures, in parallel with updating phases of signatures by the main threads. As Fig. 3 shows, although redundancy at thread-level allows the operating system to freely schedule the competing threads across all available resources [1], the structure of the watchdog thread should be organized with regard to some important facts:

1. Too many new dependencies should not appear between the watchdog and the main threads. Increasing the synchronization and communication dependencies among threads of a process will directly lead to the increased number of interruptions, and also reduction in the level of parallelism in the multi-threaded programs. So, as much as possible, the watchdog thread should not interrupt the execution of the main ones.
2. The watchdog thread can check the value of the signatures, just between two consecutive updating phases of each main thread. Otherwise, the number of false positive CFEs (number of wrongly CFE detection) will be increased. Supposed that, the watchdog thread compares the value of the signature with the expected value, after two updating phases of the main thread. Then, the mismatch is detected as a CFE, while in reality it is not.

According to these two facts, there is a conflict between them: how could the watchdog thread guarantee that the values of the signatures will be checked exactly between two consecutive updating phases of the main ones, while the execution of the main threads cannot be interrupted? This conflict between facts would be removed, if the values of the signatures are not overwritten till the certain time, so the watchdog has enough time to check the sequence of the signatures in each thread. This solution can be implemented by replacing a register with an array for storing the values of the signature. It should be noted that the virtual cores in Fig. 3 show a physical core that is assigned to a virtual machine. By default, virtual machines are allocated one core each. If the physical host has multiple cores at its disposal, however, then a core scheduler assigns execution contexts and the virtual core essentially becomes a series of time slots on logical processors. Therefore, the virtual cores are the software/hardware co-elements that is used by the system and does not show the virtualization in the operation of watchdog thread.

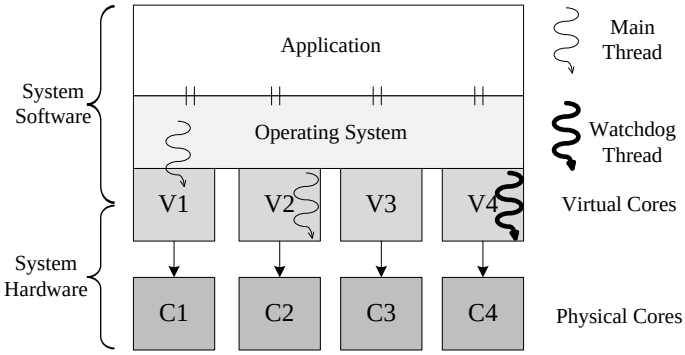


Fig. 3 The presence of watchdog thread in the architectural model of a quad-core processor

Fig. 4 illustrates this idea by implementing two updating phases of a thread. The first bit of each element of the array is reserved, as a tag bit, to show the state of each element (*U*: *Un-checked*, *C*: *Che cked*). After initialization of the elements (step *1* in Fig. 4), the main thread update the value of the signature, stored in the first element, and the state of this element is changed to *Un-checked* (step *2*). In the second updating phase, the second element of the array is updated, and the value of signature in the first element is still

maintained. This process continues until the last element of the array is updated. During the updating phase and from the first element, the watchdog can check the elements of the array which is tagged with *Un-checked*. If any mismatch is observed, occurrence of the CFE is reported, otherwise, the state of the element is changed to *Checked*. When the last element of array is also updated (step *4*), the main thread is suspended until the watchdog checks all elements, and the main thread is let to overwrite the elements of the array from the first one (steps *6, 7, and 8*).

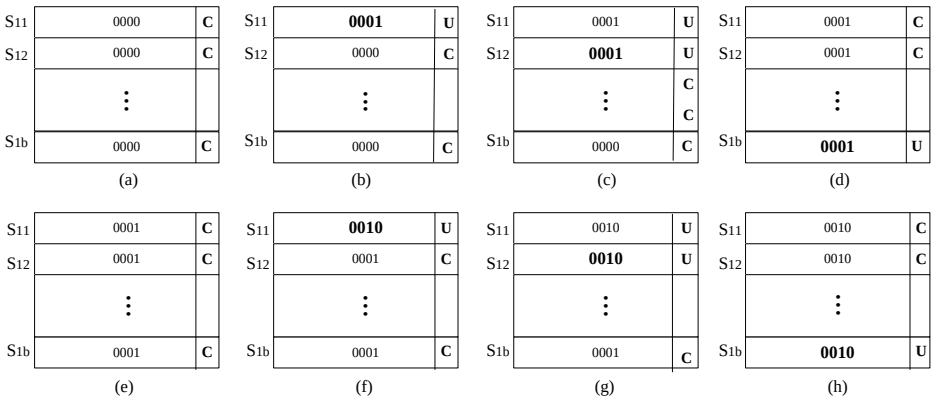


Fig. 4 The steps of updating signatures stored in the signature array of each thread: (a):step1, ... , (h):step 8 (C:Checked, U:Un-Checked)

Ignoring dependencies among running threads in phase of designing the CFGs causes that interactions among threads to be considered as CFEs. Using one shared global variable or register for storing the value of the signatures is the main reason of wrong CFE detection, since the running threads can access and alter the value of the signature register, simultaneously and without any limitation. For instance, in a dual-threaded program, if one thread updates the value of the signature between two consecutive checking phases of the other one, unexpected value of the signature is wrongly detected as a CFE by the second thread. These weaknesses in detection is known as false positive CFEs, when a technique detects some behavior in execution as CFEs, while in reality they are not. In other words, this mistake means that a positive inference about a CFE occurrence is actually false.

However, reserving separated arrays for storing the run-time signatures of each thread, introduced as one of the perquisite steps of the proposed technique for parallel CFC. Threads can alter the values of the signatures, stored in their own specific arrays, and their accesses to the arrays of the other threads are limited to check the values of them. So, the proposed technique for parallel CFC guarantees that the described scenario for checking and updating the signatures in multi-threaded programs will not be happen during the run-time.

```
1   i: a number assigned to each thread (id)
2   j: a number assigned to each element of arrays (index)
3   Sij: the value of element j in the array of thread i as the signature
4   Lij: the tag bit of element j in the array of thread i
5   Eij: the value of the expected value for element j in the array of thread i
6   Ti: the thread, identified by the value of variable i as its id
7   b: the number of elements in a array
8   n: the number of threads in the program
9   begin
10  |   x = 0,  i = 1
11  |   for (j = 1,  j <= b,  j++)
12  |   begin
13  |   |       while (Lij == 1)
14  |   |       begin
15  |   |   |       i++
16  |   |   |       if (i >  n) then
17  |   |   |   |       i = 1
18  |   |   |       end
19  |   |       end
20  |   |       if (Sij == Eij) then
21  |   |   |       x++,  Lij = 1
22  |   |       end
23  |   |       else
24  |   |   |       "CFE Occurrence"
25  |   |       end
26  |   |       i++
27  |   |       if (i > n) then
28  |   |   |       if (x == n) then
29  |   |   |   |       i = 1
30  |   |   |       end
31  |   |   |       else
32  |   |   |   |       i = 1,  goto 13
33  |   |   |       end
34  |   |       end
35  |   |       if (j == b)
36  |   |   |       Continue (Ti)
37  |   |       end
38  |   end
39  |   goto 10
40  end
```

Fig. 5 The algorithm of watchdog thread used in the proposed technique

The configuration of the watchdog thread should be organized at the design time considering the information provided by the algorithm in Fig. 5. The following steps describe the structure of the watchdog thread:

- To select the execution flow of the first thread for checking the sequence of signatures, the value of *i* is initialized to 1 (*line 10*). To ensure that the element *j* in the array of each thread has been checked (before increasing the value of the *j*), variable *x* is reserved to count the number of threads during each round of the algorithm.
- To determine the state of selected element, whether it is *Checked* or *Un-checked*, the value of L_{ij} is compared with 0 (*line 13*). 0 represents the *Un-Checked*, and 1 indicates the *Checked* state. If this element of the selected array has been checked in the previous round, the value of *i* is increased by 1 (*line 15*), and the next thread from the list of threads is selected for checking the state of element *j* in its array.
- The value of the signature is compared with the expected value to detect possible CFEs during execution (*line 20*).
- To count the number of threads, which have been checked in this round, the value of variable *x* in increased by 1, and the state of element *j* is changed to *Checked* (*line 21*).
- The value of *i* is increased by 1, to select the next thread from the list of threads (*line 26*).
- If the value of *i* is still less than the number of threads in the program, this round is repeated until the element *j* of the last thread in the list of threads is also checked (*line 27*).
- One step should be intended to ensure that the elements *j* of all threads has been checked (*line 28*). After the first round of each checking phase, if the value of *x* is less than the number of threads in the program, the tag bit of elements *j* for all thread will be re-checked. In the cases, where the main threads did not have enough time to update the signature stored in element *j*, state of their elements remains *Un-Checked*, and the process of checking the signature should be performed for them in the next rounds (*line 32*).
- At *line 38*, the watchdog has already checked the element *j* of each thread. So, the value of *j* is increased by 1 to check the next element in the array of main threads (*line 11*).
- At *line 35*, if the value of *j* is equal to the number of elements in array of each thread and all elements of each array have been already checked, then the watchdog let the main threads to continue the execution and overwrite the array from the first node (*line 36*), and the value of *i* and *j* are re-initialized (*lines 10* and *11*).

Fig. 6 shows the structure of BBs in the proposed technique. Suppose that the size of arrays is five. Therefore, after five updating phases of the main threads, they will be suspended until the watchdog let them continue their executions and overwrite the arrays. This interruption in the executions of the main threads is inevitable, because the main threads should be ensured that overwriting the elements of the array will not be detected as a CFE in the cases where the watchdog has not had enough time to check the previous value of those elements. The experimental results will show that these interruptions have negligible side effects on the execution time of the program. Moreover, the code of watchdog thread algorithm can be also divided to basic blocks. So, the proposed techniques can be easily applied on the code of watchdog thread.

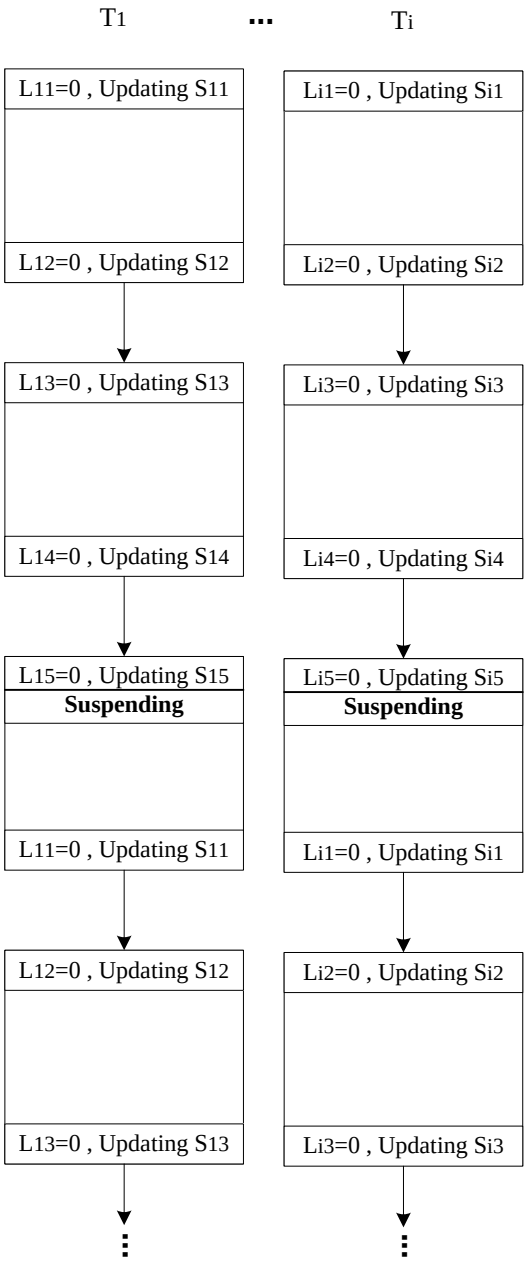


Fig. 6 The structure of the BBs in the proposed technique

5. EXPERIMENTAL RESULTS

In order to evaluate our design decisions, a quad-core processor system, Intel® core™ i7-740QM with 6.00 GB Memory RAM, is used as the simulation environment with a real operating system (Red Hat Enterprise Linux AS release 10). Also, the behaviors of eleven well-known programs of SPEC CPU2006 [30] have been studied on the simulation environment.

To implement the real behaviors of soft errors which lead to program sequence changes, a software function was written as a saboteur thread that runs simultaneously with the main program. This thread has accesses to the registers, such as instruction pointer, stack pointer, and also memory spaces with which program's threads have interactions. During simulation, the saboteur thread manipulates the content of registers and variables in a random fashion (like the effects of the bit-flips and stuck-at fault models).

Also, to observe the effects of the injected errors and to see what is going on inside a program after error injection, the GNU Project Debugger (GDB) [32] has been used. The GDB can perform several operations to help you catch misbehaviors of the execution in operation, such as examine what has happened when your program has stopped, make your program stop on specified conditions, and so on. In this work, the GDB has been utilized to identify the destinations of the illegal branches, occurred due to the error injections.

Fig. 7 shows the performance overhead of previous SCFC techniques due to checking and updating instructions. The information provided by the figure confirms the fact that the checking instructions have a greater impact on the performance degradation. On average, about 26.48% out of 35.31% performance overhead of the previous SCFC techniques is imposed just due to the execution time of checking statements.

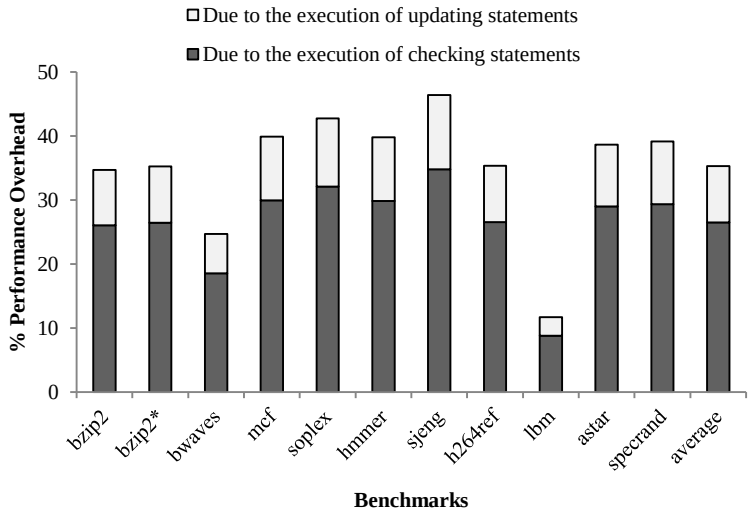


Fig. 7 The percentage of performance overhead due to the previous techniques

Fig. 8 compares the performance overhead of the related techniques. As illustrates by the figure, while array size increases, the number of interruptions during the execution and subsequently the performance overhead, is considerably reduced. For example, when the array size increases to 5, 10, and 15 the performance degradation is reduced to 24%, 20% and 19%, respectively. On the other hand, when the array size is equal to 1, as similar as previous SCF C techniques, the checking phase should take place exactly before each updating phase. Furthermore, in this case, adding a new phase to wait for watchdog thread causes increased performance overhead compared to the other cases. Therefore, increasing the size of the arrays plays an impressive role in moderating the negative effects of the conventional SCFC techniques on the execution time of the programs.

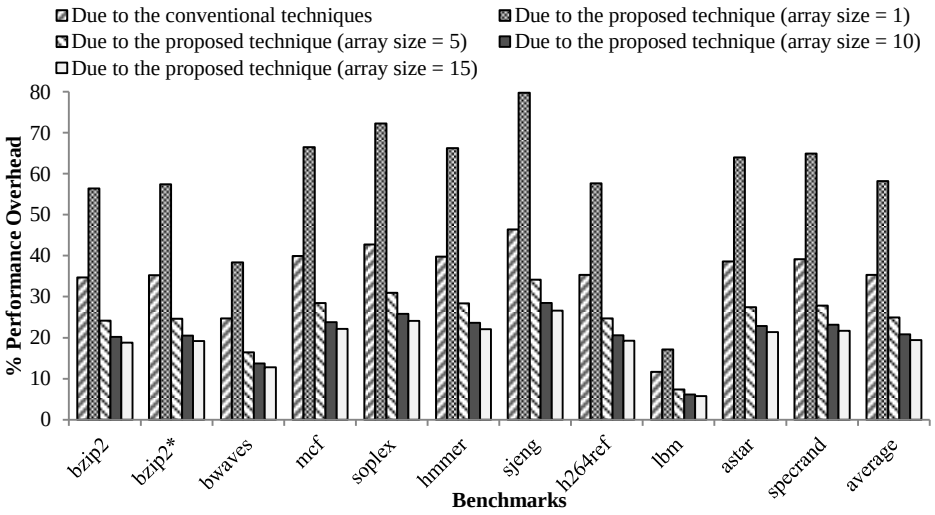


Fig. 8 Comparison of the performance overhead of the related techniques

Fig. 9 represents the average percentage of CPU usage for each benchmark during their executions in the presence of the watchdog thread compared to the average CPU usage for the original programs. With regard to the figure, the average percentage of CPU usage for the original programs is less than 22%. So, there are more underutilized CPU resources in multi-core processors during the execution which can be exploited for parallel SCFC. Also, adding the watchdog thread to the set of the running threads in the programs increases the CPU utilization up to 19%, relatively.

As similar as some previous works, to give a general comparison among all the methods, which also takes the error detection coverage and the performance overhead into account; a metric called *Method Efficiency* [15], [18], [26], [27] is defined to estimate the efficiency of the methods:

$$MethodEfficiency = \left[\frac{\left(\frac{1}{\%UndetectedErrors}\right)}{PerformanceOverhead}\right] \times 100.$$

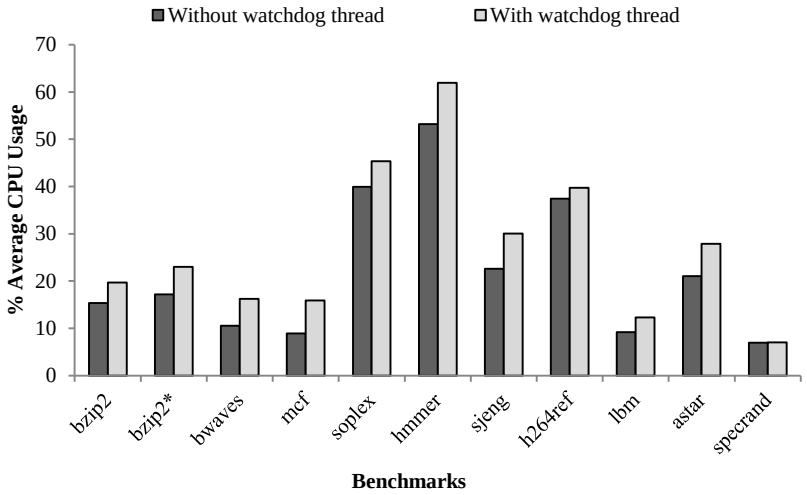


Fig. 9 Average percentage of the CPU usage for the programs during their executions

Table 2 compares the value of four important parameters obtained after applying the related techniques on the programs. The conventional techniques with two sets of the added instructions at the end and at the beginning of each BB is shown as *The Type1 Conventional Techniques*, and the conventional techniques with one set of the added instructions at the end and/or at the beginning of each BB is shown as *The Type2 Conventional Techniques*.

Table 2 General comparison among related techniques in terms of four impressive factors

Category	Technique	Error detection coverage (%)				Performance overhead (%)				Method efficiency				Error detection latency (cycle)			
		bzip2	*mcf*	*sjeng*	*astar*	*bzip2*	*mcf*	*sjeng*	*astar*	*bzip2*	*mcf*	*sjeng*	*astar*	*bzip2*	*mcf*	*sjeng*	*astar*
The conventional techniques	Type1 (like [15])	93.0	98.9	89.9	95.7	34.7	39.9	46.4	38.6	0.41	2.28	0.21	0.60	7.1	6.3	6.4	6.8
	Type2 (like [16])	82.1	86.5	80.4	84.1	17.9	21.3	25.1	20.2	0.31	0.35	0.20	0.31	9.0	8.2	7.8	8.8
The proposed techniques	With L=1	93.0	98.9	89.9	95.7	56.4	66.5	79.7	64.0	0.25	1.37	0.12	0.36	11.9	10.3	9.4	11.2
	With L=5	93.0	98.9	89.9	95.7	24.2	28.5	34.2	27.4	0.59	3.19	0.29	0.85	19.1	16.2	16.2	18.1
	With L=10	93.0	98.9	89.9	95.7	20.2	23.7	28.5	22.8	0.70	3.84	0.35	1.01	35.2	33.0	32.8	34.9
	With L=15	93.0	98.9	89.9	95.7	18.8	22.2	26.6	21.3	0.76	4.09	0.37	1.09	45.0	42.9	41.0	44.6

According to the table, the efficiency of the proposed techniques is more than the efficiency of the conventional techniques, especially when the length of the arrays grows to 5, 10 and 15. It brings to this concept that the proposed technique can effectively moderate the negative impacts of the previous techniques on the execution time of the programs. The only

drawback of the proposed technique is the increase of the error detection latency which can be controlled by adjusting the length of the array with respect to the type of applications and systems. On average, the error detection latency of the previous techniques is about 7 cycles, while this value is increased to 18, 35, and 44 cycles for the proposed techniques, with the arrays size of 5, 10 and 15, respectively. It is concluded that the size of array should be selected with respect to the type of applications and systems. For example, for real-time systems that error detection latency is important, the arrays with shorter lengths are better than the other cases. The efficiency of the proposed techniques is more than the efficiency of the conventional techniques, especially when the length of the arrays grows to 5, 10 and 15. Therefore, for high performance computing, where performance is the main goal of design process, the arrays with longer lengths are better than the other cases.

6. CONCLUSIONS

In this paper, an efficient software behavior-based technique was presented in order to detect control-flow errors in multi-threaded architectures. The goal is to enhance the applicability of the related techniques in order to be employed in multi-core processors. The key innovation is to make some changes in the structure of software-based control-flow checking techniques to exploit under-utilized resources in multi-core processors for parallel control-flow checking. Error injection experiments have shown that the proposed technique, when applied on the programs, can detect the CFEs in over 94%. The latency needed for detecting the CFEs is considerably less than the related techniques which have been recently published. A metric for estimating and comparing the efficiency of the methods was defined, and it was shown that the proposed technique is more efficient compared to conventional methods in order to be used in multi-core architectures.

REFERENCES

[1] S. Kumar, S. Hari, M. Li, P. Ramachandran, B. Choi and S. V. Adve, "mSWAT: Low-Cost Hardware Fault Detection and Diagnosis for Multicore Systems," In Proc of the 42th Annual International Symposium on Microarchitecture, 2009.

[2] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan and D. I. August, "SWIFT: Software Implemented Fault Tolerance," In Proc. of the 3rd International Symposium on Code Generation and Optimization, 2005, pp. 243-254.

[3] W. Shi, H. Hsin, S. L. Laura Falk and M. Ghosh, "An Integrated Framework for Dependable and Revivable Architecture Using Multicore Processors," In Proc. of the 33th Annual International Symposium on Computer Architecture, 2006.

[4] N. Aggarwal, P. Ranganathan, N. Jouppi and J. Smith, "Configurable Isolation: Building High Availability Systems with Commodity Multi-Core Processors," In Proc. of the 34th Annual International Symposium on Computer Architecture, 2007, pp. 340-347.

[5] M. Manoochehri, M. Annavaram and M. Dubois, "CPPC: Correctable Parity Protected Cache," In Proc. of the 44th Annual International Symposium on Microarchitecture, 2011.

[6] T. Slegel, R. Averill III, M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb, "IBM's S/390 G5 Microprocessor design," *IEEE Micro*, vol. 19, pp. 12–23, 1999.

[7] B. F. Romanescu and D. J. Sprin, "Core Cannibalization Architecture: Improving Lifetime Chip Performance for Multicore Processors in the Presence of Hard Faults," In Proc. of the 17th International Conference on Parallel Architecture and Compilation Techniques, 2008, pp. 43-50.
[8] M. D. Powell, A. Biswas, S. Gupta and S. S. Mukherjee, "Architectural Core Salvaging in a Multi-Core Processor for Hard-Error Tolerance," In Proc. of the 36th Annual International Symposium on Computer Architecture, 2009, pp. 93-104.
[9] P. M.Wells, K. Chakraborty and G. S. Sohi, "Mixed-Mode Multicore Reliability," In Proc. of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, 2009, pp. 169-180.
[10] H. Aliee, H. R. Zarandi and A. Tajary, "Dynamically Scheduled Process-Level Redundancy to Tolerate Faults in Multi-cores," In Proc. of the 9th IEEE International Conference on High Performance Computing & Simulation, 2011.
[11] A. Meixner and D. J. Sorin, "Detouring: Translating Software to Circumvent Hard Faults in Simple Cores," In Proc. of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks, 2008 pp. 80-89.
[12] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt and D. A. Connors, "Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance," In Proc. of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks, 2007.
[13] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi and D. A. Connors, "PLR: A Software Approach to Transient Fault Tolerance for Multi-Core Architectures," *IEEE Transactions on Dependable and Secure Computing*, 2008.
[14] M. Fazeli, R. Farivar and G. Miremadi, "Error Detection Enhancement in PowerPC Architecture-based Embedded Processors," *Journal of Electronic Testing: Theory and Applications*, vol. 24, pp. 21-33, 2008.
[15] R. Vemu and J. A. Abraham, "CEDA: Control-flow Error Detection through Assertions," In Proc. of the 12th IEEE International On-Line Testing Symposium, 2006, pp. 151–158.
[16] N. Oh, P. Shirvani and E. McCluskey, "Control-flow Checking by Software Signatures," *IEEE Transactions on Reliability*, vol. 51, pp. 111–122, 2002.
[17] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda and M. Violante, "Improved Software-Based Processor Control-Flow Errors Detection Technique," In Proc. of the 50th Reliability and Maintainability Syposium, 2005, pp. 583-589.
[18] R. Vemu, S. Gurumurthy and J. A. Abraham, "ACCE: Automatic Correction of Control-flow Errors," In Proc. of the IEEE International Test Conference, 2007, pp. 1-10.
[19] G. Blake, R. G. Dreslinski, T. Mudge, K. Flautner, "Evolution of Thread-Level Parallelism in DesktopApplications," In Proc. of the 43th Annual International Symposium on Microarchitecture, ISCA, 2010.
[20] M. Rimén, J. Ohlsson and J. Karlsson, "Experimental evaluation of control flow errors," In Proc. of the Pacific Rim International Symposium on Fault Tolerant Systems, 1995, pp. 238–243.
[21] M.A. Schuette, J.P. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams," *IEEE Transactions on Computers*, vol. 36, pp. 264-276, 1987.
[22] A. Rajabzadeh and G. Miremadi, "Transient detection in COTS processors using software approach," *Journal of Microelectronics Reliability*, vol. 46, pp. 124-133, 2006.
[23] A. Rajabzadeh, G. Miremadi and M. Mohandespour, "Error detection enhancement in COTS superscalar processors with performance monitoring features," *Journal of Electronic Testing: Theory and Applications*, vol. 20, pp. 553-567, 2004.
[24] A. Rajabzadeh and G. Miremadi, "CFCET: A Hardware-Based Control Flow Checking Technique in COTS Processors Using Execution Tracing," *Elsevier Journal of Microelectronics Reliability*, vol. 46, pp. 959-972, 2006.
[25] U. Schiffel, A. Schmitt, M. Süßkraut and C. Fetzer, "ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software," In Proc. of the 29th International Conference on Computer Safety, Reliability and Security, 2010, pp. 169-182.
[26] H. R. Zarandi, M. Maghsoudloo and N. Khoshavi,"Two Efficient Software Techniques to Detect and Correct Control-flow Errors," In Proc. of the 16th IEEE Pacific Rim International Symposium on Dependable Computing,2010, pp. 141-148.
[27] M. Maghsoudloo, H.R. Zarandi, S. Pour-Mozaffari and N. Khoshavi, "Soft Error Detection Technique in Multi-threaded Architectures Using Control-Flow Monitoring," In Proc. of the 14th Euromicro Conference on Digital System Design, Architecture, Methods and Tools, 2011.
[28] E. D. Berger, T. Yang, T. Liu and G. Novark, "Grace: Safe Multithreaded Programming for C/C++," In Proc. of the 24th SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, 2009, pp.81-96.
[29] David Patterson, "The Trouble with Multi-Core," *Journal of IEEE Spectrum*, vol. 47, pp. 28-32, 2010.

[30] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks, http://www.specbench.org/cpu2006, 2006.
[31] Sh. Parsaeeian, and A. Rajabzadeh, "Multi-Bit Upset Faults Correction in Embedded Systems", *Journal of Iran Computer Society*, 2011.
[32] GDB: The GNU Project Debugger, http://www.gnu.org/s/gdb.