A Hardware Approach to Concurrent Error Detection Capability Enhancement in COTS Processors

Amir Rajabzadeh, Seyed Ghassem Miremadi
Razi University, Kermanshah, Iran, Sharif University of Technology, Tehran, Iran
rajabzadeh@razi.ac.ir, miremadi@sharif.edu

Abstract

To enhance the error detection capability in COTS (commercial off-the-shelf) -based design of safety-critical systems, a new hardware-based control flow checking (CFC) technique will be presented.

This technique, Control Flow Checking by Execution Tracing (CFCET), employs the internal execution tracing features available in COTS processors and an external watchdog processor (WDP) to monitor the addresses of taken branches in a program. This is done without any modification of application programs, therefore, the program overhead is zero. The external hardware overhead is about 3.5% using an Altera Flex 10K30 FPGA. For different workload programs, the execution time overhead and the error detection coverage of the technique vary between 33.3 and 140.8% and between 79.7 and 84.6% respectively. The errors are detected with about zero latency.

1. Introduction

Commercial Off-The-Shelf (COTS) processors are widely used in low-cost safety-critical systems design today [1-4], because of the chip's availability in the market, time-to-market [5], development cost [3,4,6], test equipment cost [4], trust in products and maintainability [3] of the systems.

Safety-critical systems require high error detection. However, COTS processors used in these systems may have limited error detection capability [3,6]; hence, this motivates the use of additional error detection techniques to enhance the error detection capability in these systems.

To enhance the error detection capability, Control Flow Checking (CFC) techniques have shown to be cost-effective, and to present effective coverage [7]. CFC techniques can be implemented with software (SW-CFC), hardware (HW-CFC), or a hardware-software approach (HWSW-CFC).

The main drawbacks of the SW-CFC techniques are:

1) Processor crashes and Infinite loops are difficult or impossible to detect,

2) Modifications to application programs are needed.

HWSW-CFC techniques, such as Time-Time-Address checking (TTA) [8], Time Signature Monitoring (TSM) [9], Signature Instruction Stream (SIS & ASIS) [10] Continuous Signature Monitoring (CSM) [11], Implicit signature checking (ISC) [12] and Committed Instructions Counting (CIC) [1], can remove the first drawback.

HW-CFC techniques, such as Watchdog Direct Processing (W-D-P) [13] and On-line Signature Learning and Checking (OSLC) [14], can remove the above mentioned SW-CFC's drawbacks.

Although, the HW-CFC and HWSW-CFC techniques can remove or moderate the drawbacks of SW-CFC techniques, they incur in two main drawbacks:

1) As these techniques monitor all memory accesses to extract signatures [8,11-14], from addresses or instructions, they can not be applied to processors with internal cache.
2) As these techniques use execution time as a signature [8,9], they can not be applied to superscalar processors.

One effective way to remove the two latter drawbacks from the HW-CFC and HWSW-CFC techniques is to employ new features in modern COTS processors, i.e., feature specific control flow checking (FS-CFC) techniques [1]. This paper presents a new hardware-based FS-CFC technique, called Control Flow Checking by Execution Tracing (CFCET), which employs the internal execution tracing features available in several modern COTS processors [15]. The execution tracing can be configured to generate

special bus cycles needed to externally monitor the addresses of taken branches in a program. This is done by run-time comparison of the linear addresses at which branch instructions are stored as well as their target addresses against corresponding addresses calculated at compile-time.

The external hardware overhead is about 3.5% using an Altera Flex 10K30 FPGA and the execution time overhead varies between 33.3 to 140.8% for different workload programs. The overheads have been measured experimentally by executing the workloads on a Pentium system. The error detection coverage of the technique has been evaluated with simulation and the results show that it varies between 79.7 to 84.6% depending on the different workload programs. The main advantages of the CFCET technique are:

1) It can be applied to the COTS processors with pipeline and on-chip caches.
2) No modifications to the workload programs are required.
3) Errors can be detected with about zero latency.

The next section presents the error models, the idea and implementation of the CFCET technique, the List2VHDL translator, and the ETdriver module. A case study containing overheads and coverage is discussed in section 3. Method discussions and comparisons are given in section 4. Finally, section 5 concludes the paper discussion.

2. The CFCET technique

The CFCET technique is aimed to detect specific errors using the execution tracing features available in most COTS processors. This section describes the Program Jumps Graph (PJG), the error model and the CFCET technique.

2.1. Program jumps graph (PJG)

To define the PJG, some other definitions are necessary.

Definition 1. *A control instruction* is an instruction that modifies the instruction sequence such as *jump*, *call, loop* and *ret* instructions.

Definition 2. *A destination instruction* is an instruction whose address is the destination of a control instruction [13].

Definition 3. *A Program Jumps Graph (PJG)* is a directed graph which represents legal path in a program. A PJG contains a set of vertices (V) and edges (E) written as PJG = {V, E}, where V={*vi* : i = 1, 2, …, n} and E={*ei* : i = 1, 2, …, m}. Each node, *vi,* represents either a control instruction address or a destination instruction address, and each edge, *ei,* represents the branch br*i,j* from a control instruction, *vi,* to a destination instruction, *vj*.

Definition 4. *A Reference PJG* is a PJG that is extracted from a program source.

Definition 5. *A Run-time PJG* is a PJG generated during the program execution.

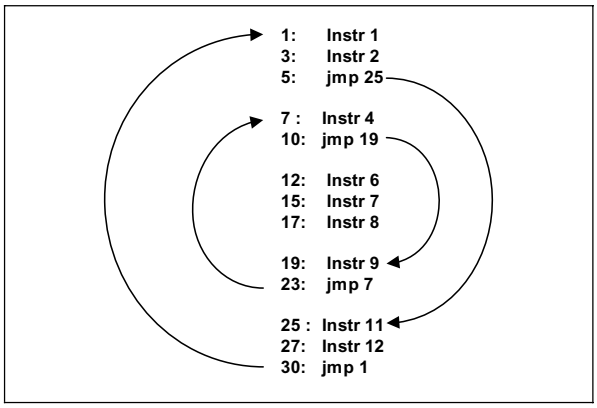

Figure 1.a. A typical program

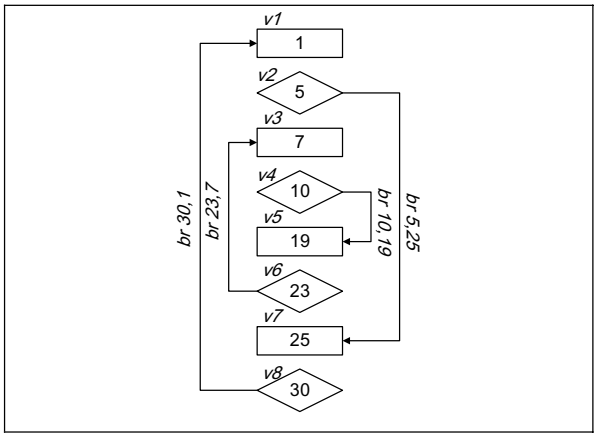

Figure 1.b. The program jumps graph

Figures 1a and 1b show a typical program and its reference PJG, respectively, where V={1, 5, 7, 10, 19, 23, 25, 30} and E={ br*5,25* , br*10,19* , br*23,7* , br*30,1*}.

2.2. Error model

The error model in this work is based on the Control Flow Error (CFE). Using the PJG definitions, we can now define the CFE model.

Definition 6. *A Control Flow Error (CFE)* is any run-time PJG violation from the reference PJG.

Three types of CFE previously presented in [16], is used in this work. These types of CFE can be caused

by transient or permanent faults found in the memory or address circuits [16].

CFE type 1) Branch Insertion Error (BIE): a BIE occurs when one of the non-control instructions in the program is changed to a control instruction as the result of a fault and the control instruction actually causes a taken branch.

CFE type 2) Branch Target Modification Error (BTME): a BTME occurs when the target address of one control instruction is modified as the result of a fault and the control instruction actually causes a taken branch.

CFE type 3) Branch Deletion Error (BDE): a BDE occurs when a fault causes a control instruction of a program changes to a non-control instruction.

2.3. The basic idea of the CFCET technique

The CFCET technique checks the correctness of the PJG at run time. When the workload program is executing, the CFCET technique compares the reference PJG with the run-time PJG to detect an illegal edge (CFE) caused by transient faults in the memory or address circuits. The reference PJG has been extract from the program codes and has been stored in the associative memory of the WDP. The run-time PJG information, i.e., addresses of the control and destination instructions, is available in the buses during a special bus cycles i.e., Branch Trace Message (BTM) cycles. In normal operation, the run-time PJG is a sub-graph of the reference PJG. Any violation from the reference PJG is announced as a CFE. To utilize the BTMs cycles, the CFCET technique uses a special internal facility in COTS processors, called the execution tracing features.

Execution tracing features*:* Many of the current COTS processors such as Pentium [15] and PowerPC include features called execution tracing, to trace the program executed in the processor. In our study, we have used a Pentium® processor whose execution tracing features are mainly based on an internal register called the TR12 register. The TR12 register can optionally be configured to generate a special bus cycle, called the Branch Trace Message (BTM). During a BTM cycle, useful information of control instructions is sent out of the processor when a branch is taken. The information includes the linear addresses at which the control instructions are stored as well as their target addresses. A BTM cycle occurs first if a branch is taken. There are two forms of BTMs: normal and fast. The normal BTM produces two cycles; the first cycle indicates that the target address of the control instruction is available on the external buses, and the second cycle indicates that the address of the instruction is available on the external buses. The fast BTM produces only the latter cycle. The CFCET technique uses the normal BTMs.

The CFCET implementation*:* to implement the CFCET technique, the following actions are required.

1) Extracting the PJG's program, i.e., linear addresses of jumps, loops, subroutine calls and return instructions and their target addresses, and developing a WDP which stores the PJG. To do this a program called *List2VHDL* has been developed.
2) Enabling the execution tracing unit of the main processor to generate the desire BTMs cycles. To do this a module called *ETdriver* has been developed.

The watchdog processor: the CFCET technique requires an external WDP connected to the main processor. The WDP contains a BTM cycles decoder, an associative memory and several data registers.

The BTM cycles decoder is used to recognize special BTM cycles and also pick out relevant information to be stored in the data register from the busses.

The WDP also contains an associative memory with *m* entries, corresponding to *m* edges in the reference PJG. The associative memory actually has *m* comparators, which compare the stored control/destination Instructions' addresses (reference PJG) with the run-time control/destination Instructions' addresses (run-time PJG). The WDP organization is shown in Figure 2.

To automatically generate the VHDL codes needed for the WDP synthesis, the *List2VHDL* program may be used.

List2VHDL: to employ the CFCET technique, a WDP must be developed. To do this, a program called *List2VHDL* has been developed which could accept a *list file* and generate a VHDL program, see Figure 3. The *list file* may be generated by the *objdump* program (in Linux OS). This VHDL program contains a BTM cycles decoder, several registers and the reference PJG which is stored into an associative memory. This VHDL description could synthesize to an FPGA as a WDP.

ETdriver: A module for the execution tracing control: the CFCET technique requires that the main processor generates the BTMs cycles. Therefore, a program must enable and configure the execution tracing unit of the processor in a way that the processor sends out the linear addresses of taken branches, i.e., instruction addresses and their target addresses of control instructions. In our study, we have used a Pentium® 233 MHz processor whose Execution Tracing features are mainly based on an internal registers, i.e., TR12. To write in the TR12 register, the processor has an instruction, i.e., WRMSR. The

WRMSR instruction can not be executed in all CPLs (Current Privileged Level) and may only be executed in CPL0 (kernel mode). Therefore, a driver module, called ETdriver, under the Linux OS has been developed. The driver module configures the execution tracing in the normal BTMs generation, i.e., each BTM generates two cycles, one for the target address of the control instruction and another for the address of the control instruction.

The source code of *init_module* and *cleanup_module* of the ETdriver are shown in Figure 4.

The *init_module*/*cleanup_module* is executed when the ETdriver is inserted/removed. To insert/remove the ETdriver, the linux uses insmod/rmmod commands. When the ETdriver is inserted, *init_module* is executed and the main processor begins to generate the BTMs cycles. Generation the BTM cycles will be continued until the ETdriver is removed.

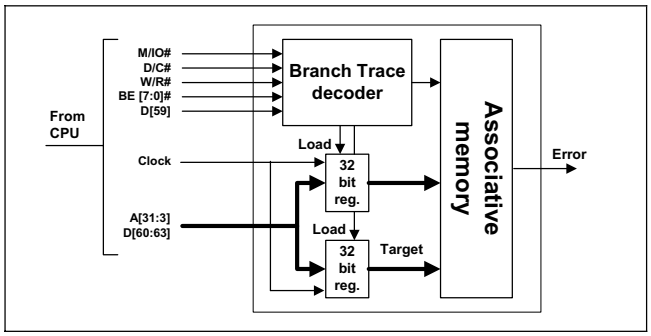

Figure 2. CFCET WDP organization

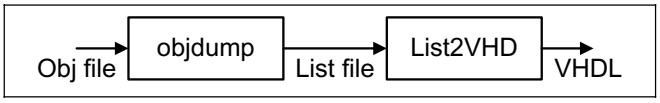

Figure 3. List2VHDL translator

```
Int init_module (void)
{
__asm (“pushal”);
__asm (“pushfl”);
//Write in the TR12
__asm(“movl $0x0e,%ecx”);
__asm(“movl $0x00,%edx”);
__asm(“movl $0x02,%eax”);
//Normal Cycle (2 cycles)
__asm("wrmsr");
__asm(“popfl”);
__asm(“popal”);
return 1;
}
```

```
void cleanup_module (void)
{
__asm (“pushal”);
__asm (“pushfl”);
//Write in the TR12
__asm(“movl $0x0e,%ecx”);
__asm(“movl $0x00,%edx”);
__asm(“movl $0x00,%eax”);
//clean execution tracing
__asm("wrmsr");
__asm(“popfl”);
__asm(“popal”);

}
```

Figure 4. The ETdriver module

3. A case study

To demonstrate the effectiveness of the CFCET technique, three different workload programs were run in a Pentium® 233 Mhz system. These workloads are:

1) Matrix: a matrix manipulation program, which finds the inverse of an 8×8 matrix.
2) List: a linked list program, which deletes 10 elements of 16-element structures from a linked list and adds them to another linked list.
3) Qsort: a quick sort program, which sorts a 100-element array of integers.

The hardware overhead and the execution time overhead were extracted, experimentally. The error detection coverage was obtained base on simulation.

Hardware overhead: the external WDP used in the CFCET technique incurs external hardware overhead. The hardware overhead depends on the number of control instructions in the workload program. Three programs, the Matrix, the List and the Qsort, which have all written in the C language and then changed to assembly codes, have been used as workloads. Using the List2VHDL translator, the VHDL description files corresponding to the workloads have been generated. The WDP has been synthesized to an Altera Flex 10K30 FPGA. Table 1 shows the amount of the FPGA resources used for the WDP implementation. As shown in Table 1, the WDP consumes about 1.2% of the D flip flops (DFFs) and 3.5%, 3.5% and 3.1% of the logic cells (LCs) for the Matrix, the List and the Qsort workload programs respectively. Consumed resources depend on the number of jumps, loops, subroutine calls and return instructions in the program. These values are 62, 60 and 52 instructions for the Matrix, the List and the Qsort workload programs respectively (see Table 2).

Execution time overhead: the CFCET program size and execution time overheads are presented in Table 2. As shown in Table 2, the program size overheads are zero for all workload programs because the CFCET technique does not require a modification the program codes. This is a great advantage of the CFCET technique in comparison to other techniques.

The execution time overhead in absence of faults has been experimentally evaluated. Notice that a BTM is generated due to a taken branch. Therefore, if the number of taken branches increases, then the execution time increases. The number of taken branches shown in Table 2 were counted using Performance Monitoring [15]. As shown in Table 2, the number of taken branches for Matrix, List and Qsort workload programs are 5789, 345, 21642, which incur execution

time overheads of 33.3%, 73.3% and 140.8% respectively.

The execution time of workload programs shown in Table 2 are based on internal clock pulses, which is 233 MHz. These execution times are measured by the Time Stamp Counter (TSC) in the processor [15]. The TSC is a 64-bit time counter, which is provided on the chip. This counter increment on every clock cycle from processor reset. The execution time of workload programs without and with the execution tracing in Table 2, were measured using the TSC.

Table 1. Available and consumed FPGA resources for the external WDP implementation

	Matrix	List	Qsort
Number of control instructions in the program	62	60	52
Total available LCs in The FPGA	4992		
Total available DFFs in The FPGA	5392		
Total of consumed LCs For the WDP (%)	175 (3.5%)	174 (3.5%)	153 (3.1%)
Total of consumed DFFs For the WDP (%)	65 (1.2%)	65 (1.2%)	65 (1.2%)

Table 2. Overheads statistic

	Matrix	List	Qsort
Overhead of the program size	0%	0%	0%
Number of control instructions in the program	62	60	52
Number of taken branches at run-time	5,789	345	21,642
Execution time without execution tracing (per internal clock pulses)	115,804	4,735	182,808
Execution time with execution tracing (per internal clock pulses)	154,320	8,200	440,229
Execution time Overhead	33.3%	73,3%	140.8%

Simulation-based coverage evaluation: the error detection coverage of the CFCET technique was obtained based on simulation. For simulating the error detection coverage of the CFCET, we assume that probability distribution of the error occurring in each memory byte will be uniform. A program contains two groups of instructions; i.e., *control* instructions and *non-control* instructions. In this subsection, *control* instructions contain two groups; i.e., *branch* instructions and *ret* instructions. According to the error model; i.e., CFEs, which have been described in subsection 2.2, each *non-control* instructions in the program can only be affected by a Branch Insertion Error (BIE). Each *branch* instruction in the program can be affected by Branch Target Modification Error (BTME), and Branch Deletion Error (BDE) and each *ret* instruction in the program can only be affected by Branch Deletion Error (BDE). In these cases, the CFCET behavior is clearly determined.

CASE 1. BIE occurrence: a BIE occurring in a node causes a BTM to be generated on the buses. The WDP recognizes BTM cycles and picks out relevant information from the buses. Since these instruction and target addresses are not in the associated memory, the WDP announces a CFE. Therefore, the coverage is 100% with 0 cycle latency.

CASE 2. BTME occurrence: a BTME occurring in a node causes a BTM with modified address to be generated on the buses. The WDP recognizes BTM cycles and picks out relevant information from the buses. Since the target address of this instruction address is different from the saved one in the associated memory, the WDP announces a CFE. Therefore the coverage is 100% with 0 latency.

CASE 3. BDE occurrence: A BDE occurring in a node is not detected by the WDP, because it is similar to a not taken branch. Notice that, in this case the PJG violation has not occurred.

Notice that, although the faults in a *non-control* instruction opcode or operands may change it to another non-control instruction, e.g., changing ADD to SUB or ADD ax, 10h to ADD ax, 30H, or the faults in a branch instruction opcode may change it to another branch instruction; e.g. jne L1 to je L1, they do not belong to the errors models which were introduced. These cases are summarized in Table 3.

Table 3. Faults in the program and the CFCET behaviors

Memory regions	Possible changes	A	B	C	D%
Non-control opcodes	Non-control→branch	√	BIE	Y	100
	Non-control→ret	√	BIE	Y	100
	Non-control→ non-control				
Non-control operands					
branch opcodes	branch→branch				
	branch→ret	√	BTME	Y	100
	branch→non-control	√	BDE	N	0
branch operands	branch→modified target branch	√	BTME	Y	100
Ret opcodes	Ret→branch	√	BTME	Y	100
	ret →ret				
	ret → non-control	√	BDE	N	0

A: In the error model domain B: Effective error model
C: Can detect by the CFCET D: Error Detection Coverage

For extracting the average of error detection coverage of the CFCET technique, the single event upset (SEU) faults were injected in the *non-control* instructions opcodes, *branch* instructions opcodes and operands and *ret* instructions opcodes in the workload programs. This has been done as follows for each workload programs:

1) Compiling the program, written in C language, to an object file using *gcc* compiler.
2) Disassembling the object file to a List file using *objdump* program.
3) Extracting the instructions opcode and their operand values of the program and storing them in an access data bank, say *program bank*.
4) Categorizing the instruction's opcode of the *program bank* to three groups, i.e., branch, non-control and ret group instructions.
5) Creating another access data bank, say *reference bank*, and filling it with the total of the Pentium instructions set (ISA) opcode values.
6) Categorizing the instructions opcode of the *reference bank* to four groups, i.e., *branch*, *non-control*, *ret* and *unused* group instructions.
7) Defining two variables, i.e., NOIE (Number of Injected Errors) and NODE (Number of detected Errors), and set them to zero.
8) Injecting an SEU (Single Event Upset- a bit flip) fault in to an instruction opcode in the *program bank.*
9) Mapping the faulty instruction to the *reference bank* to determine what the new group instruction is. According to the Table 3, if changed instruction does not belong to the error models (hachured rows), go to article 8, otherwise incrementing the NOIE.
10) According to the Table 3, if the error can be detected by the CFCET, incrementing the NODE.
11) Repeating the articles of 8, 9 and 10 for all instructions' opcode in the *program bank*. This is done using an SEU fault injection per each bit in the instruction opcode, i.e., one fault for each bit of instruction opcode in each iteration. Actually, a list of all possible instructions, from the Pentium ISA, for which the opcode values have Hamming distance 1 from the opcode values of the workload programs' instructions have been obtained.
12) Injecting an SEU fault in to a *branch instruction* operand in the *program bank* and incrementing the NOIF.
13) Incrementing the NODE. According to the Table 3, all of this type of errors (branch→modified target branch) can be detected by the CFCET.
14) Repeating the articles of 12 and 13 for all *branch instruction* in the *program bank*. This is done using an SEU fault injection per each bit in the *branch instruction* operand, i.e., one fault for each bit of *branch instruction* operand.

The error detection coverage of the CFCET technique is NODE/NOIE. These results are presented in Table 4.

Table 4. Error detection coverage of CFCET

	Matrix (%)	List (%)	Qsort (%)	Matrix + List + Qsort (%)
Error detection coverage	83.3	84.6	79.7	82.5

4. Method discussions and comparison

In this section, the other aspects of the CFCET are presented.

The CFCET and memory management system: most of the memory management system uses the paging and the segmentation systems.

The CFCET and paging system: the address before the paging system is called *linear address*, linear address means the address offset in a current segment. The address after the paging system is called *physical address*, which is delivered to memory chips. The CFCET technique uses the internal execution tracing features, which provide the ability to monitor linear addresses of taken branches in a program. Therefore, the paging system has not any conflict to the CFCET technique implementation.

The CFCET and segmentation system: An address before the segmentation system is called *virtual address*; virtual address is a relative address from the beginning of the process. This addressing mode, i.e., segmentation, is the default and cannot be disabled in the Intel architecture.

Today, the segmentation system is ignored for most of new operating systems. For example in Linux OS, four segments have been defined; user code segment, user data segment, kernel code segment and kernel data segment, each of them has a defined capacity ranging from 0 to 4GB. Therefore, the segmentation system is ignored and the virtual addresses are equaled to the linear addresses. Therefore, all processes' linear addresses begin from zero and these types of segmentation systems cannot make limitations for the CFCET.

The CFCET can also be developed for the operating systems in which the segmentation system is not ignored, because when a new segment is loaded, a BTM sends out the address of the new segment. The WDP can save the new segment's address and add it with the next BTM address.

The CFCET and multitasking environments: although this paper presented the CFCET in a single task environment by disabling interrupts during the workload program execution, however, it may be developed in multitasking environments in the several ways:

1) The address information of all tasks as well as the context-switching interrupt address is stored in the WDP. When an interrupt is invoked, a BTM sends out the relevant address, i.e., the address at which the interrupt occurred, and the interrupt address. The WDP detects a context-switch being performed. At the end of the interrupt another BTM is generated and the return address is sent out. According to the BTM return address, the WDP can detect which process has been switched to.
2) Linux is an open source OS. The Linux context-switching interrupt can be modified to send relevant information to the WDP.

The CFCET versus other control flow checking techniques: several hardware based control flow checking (HW-CFC and HWSW-CFC) techniques have been proposed by researchers, such as: Time-Time-Address checking (TTA) [8], Time Signature Monitoring (TSM) [9], Signature Instruction Stream SIS [10] Continuous Signature Monitoring (CSM) [11], Watchdog Direct Processing (W-D-P) [13] and On-line signature learning and checking (OSLC) [14], Implicit signature checking (ISC) [12] and Committed Instructions Counting (CIC) [1]. In this subsection, the comparison of the CFCET with abovementioned technique will be presented.

As are shown in Table 5, the program size and execution time overhead of the CFCET are compared with other techniques. Program size of the CFCET, W-D-P and OSLC techniques is zero because they do not change the program codes. The program size of the other techniques varies between 4% and 39%. The CFCET incurs 33.3% to 140.8% execution time overhead for different workload programs. While this parameter for the other techniques vary between 0% and 210%. Although, the CFCET does not modify the workload program likes the W-D-P and OSLC techniques, it incurs the execution time overhead. Because, the modern processors use an internal frequency for their cores which is several times greater than the external frequency of their external buses. Therefore, external activities such as BTMs are time consuming and incur performance loss. The error detection latency of the CFCET techniques is about zero. Therefore, the propagation of the error is reduced. Meanwhile, the CFCET detects an error before that error actually occurs, because an illegal jump is detected when it is taken, before the execution of its target instruction, this is very important in safety-critical and fail-silent applications. The other techniques, detect an error after its occurrence. As in Table 5, The CFCET and CIC were designed for the modern processors with on-chip caches and pipelines. However, the other techniques may not be applied to modern processors.

Table 5. Comparison of the CFCET technique with other HW-CFC and HWSW-CFC techniques

CFC techniques	Program size overhead (%)	Execution time overhead (%)	Error detection coverage (%)	Error detection latency	Need program modification	Applicability in modern CPUs
TTA [8]	35~39	28~30	87.5~92.6	62~101 cycles	Yes	No
TSM [9]	10.1~16.4	10~20	74.3~92.7	-	Yes	No
SIS [10]	6~15	-	82	3800 μs	Yes	No
CSM [11]	4~11	-	99.99	1 cycle	Yes	No
W-D-P [13]	0	-	-	0	**No**	No
OSLC [14]	0	0	98.6	25.6 μsec	**No**	No
ISC [12]	10 ~ 25	6.2 ~ 32.7	$1\text{-}2^{-v}$*	2~5 Instr.	Yes	No
CIC [1]	5~10	210	91~98.41	80 cycles	Yes	**Yes**
CFCET	0	33.3~140.8	79.7~84.6	0	**No**	**Yes**

* For v-bit signature

5. Summary and conclusions

COTS processors are widely used in industrial, embedded, and real-time applications. In these applications, features like reliability, safety, security, and availability are important. The low error detection coverage in the COTS processors motivates the use of other error detection techniques to improve the error detection coverage. This paper presents a new hardware based error detection technique called Control Flow Checking by Execution Tracing (CFCET) to increase error detection in COTS processors. The technique uses the internal execution tracing features which facilitate to monitor the

addresses of taken branches in a program using an external watchdog processor (WDP). This technique does not need to modify the application programs. The error detection coverage of the technique has been evaluated by simulation and the results show that depending on the different workload programs used, the technique can detect between 79.7 and 84.6% of errors. The error detection latency of this technique is about zero.

6. References

[1] Rajabzadeh A., M. Mohandespour and G. Miremadi, "Error Detection Enhancement in COTS Superscalar Processors with Event Monitoring Features", *Proceedings of the 10th IEEE/IFIP International Pacific Rim Symposium on Dependable Computing (PRDC-2004)*, March 2004, pp. 49-54.

[2] Rajabzadeh A. and G. Miremadi, "Transient Detection in COTS Processors Using Software Approach", *to appear in the Elsevier Journal of Microelectronics Reliability*, 2005.

[3] Chevochot P. and Puaut I., "Experimental evaluation of the fail-silent behavior of a distributed real-time run-time support built from COTS components", *Proc. of the International Conference on Dependable Systems and Networks (DSN-2001)*, July 2001, pp. 304–313.

[4] Pizzica S. V., "Meeting military system test requirements with the usage of COTS products", Proc. of the 17th AIAA/IEEE/SAE Digital Avionics Systems Conference (DASC), vol. 1, Nov. 1998, pp. B45/1~B45/7.

[5] Croll P. and P. Nixon, "Developing Safety-Critical Software within a CASE Environment", *IEE Colloquium on Computer Aided Software Engineering Tools for Real-Time Control*, Apr. 1991, pp. 8.

[6] Madeira H., Some R. R., Moreira F., Costa D. and Rennels D., "Experimental evaluation of a COTS system for space applications", *Proc. of the International Conference on Dependable Systems and Networks (DSN-02)*, June 2002, pp. 325–330.

[7] Mahmood A. and E.J. McCluskey, "Concurrent Error Detection Using Watchdog Processors - A Survey", *IEEE Transactions on Computers*, Feb. 1988, pp. 160 -174.

[8] Miremadi G., J. Ohlsson, M. Rimen, and J. Karlsson, "Use of Time, Location and Instruction Signatures for Control Flow Checking", *Proc. of the DCCA-6 International Conference, IEEE Computer Society Press*, 1998. pp. 201-221.

[9] Madeira H., M. Rela, P. Furtado and J. G. Silva, "Time Behaviour Monitoring as an Error Detection Mechanism", *3rd IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-3)*, Sept. 1992, pp. 121-132.

[10] Schuette M. A. and J. P. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams", *IEEE Trans. on Computers*, Vol. C-36, No. 3, March 1987, pp. 264-276.

[11] Wilken K. and J. P. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* , Vol. 9, , June 1990, pp. 629-641.

[12] Ohlsson J. and M. Rimen, "Implicit Signature Checking", *Twenty-Fifth International Symposium on Fault-Tolerant Computing, FTCS-25*, 1995, pp. 218 -227.

[13] Michel T,, R. Leveugle and G. Saucier, "A New Approach to Control Flow Checking without Program Modification", *21st Int. Symposium on Fault-Tolerant Computing*, 1991, pp. 334-341.

[14] Madeira H. and J. G. Silva, "On-Line Signature Learning and Checking: Experimental Evaluation", *Proc. Of Advanced Computer Technology, Reliable Systems and Applications (CompEuro '91)*, May 1991, pp. 642 -646.

[15] Intel Corp., *Pentium ® Processor Family Developer's Manual*, 1997.

[16] Oh N., P. P. Shirvani and E. J. McCluskey, "Control-Flow Checking by Software Signatures", *IEEE Transactions on Reliability*, vol. 51, no. 2, March 2002, pp. 111-122.