Error Detection Enhancement in COTS Superscalar Processors with Event Monitoring Features

Amir Rajabzadeh, Mirzad Mohandespour, Ghassem Miremadi
Sharif university of technology
Department of computer engineering ,Azadi Ave, Tehran, Iran
{rajabzad, mohandes}@ce.sharif.edu, miremadi@sharif.edu

Abstract

Increasing use of commercial off-the-shelf (COTS) superscalar processors in industrial, embedded, and real-time systems necessitates the development of error detection mechanisms for such systems. This paper presents an error detection scheme called Committed Instructions Counting (CIC) to increase error detection in such systems. The scheme uses internal Performance Monitoring features and an external watchdog processor (WDP). The Performance Monitoring features enable counting the number of committed instructions in a program. The scheme is experimentally evaluated on a 32-bit Pentium® processor using software implemented fault injection (SWIFI). A total of 8181 errors were injected into the Pentium® processor. The results show that the error detection coverage varies between to 90.92% and 98.41%, for different workloads.

1. Introduction

Commercial off-the-shelf (COTS) superscalar processors such as Intel Pentium family [14], [16], PowerPC and DEC's Alpha [15] are widely used in industrial [17], embedded [11], [21] and real-time [19], [21] applications. The most notable reasons, which encourage the engineers to use COTS processors in these applications are time-to-market [17], development cost, and maintainability [19] of the systems. In such applications, system failures may endanger human life or may cause loss of valuable information [17]; here reliability [11], safety, security [17], and availability [18], [11] are of important concerns.

The combination of internal error detection mechanisms available in the COTS superscalar processors, has limited error detection coverage and poor error containment provisions [19], [20]. For example, Pentium® processors employ different levels of parity checking to detect errors occurring in the memory arrays of the chip; however, these levels of parity checking can, in the best case, provide a 53% error detection coverage [3]. This is an unacceptable coverage which necessitates the use of additional error detection techniques to enhance the error detection coverage. Many of the previous cost-effective approaches were based on the use of an external watchdog processor (WDP) to monitor processor and memory access behavior [4], [5], [6], [7], [9], [12], [13]. However, these techniques cannot be applied to COTS processors with on-chip caches and pipeline features.

This paper presents an error detection scheme called Committed Instructions Counting (CIC), which includes six different error detection mechanisms to detect different types of control flow errors (CFEs) in processors with on-chip caches and pipeline features. The scheme combines a simple WDP and the Performance Monitoring features of COTS superscalar processors, such as AMD x86-64 [1], Alpha [2], Pentium [3], MIPS R10000 [8] and PowerPC 604 [10] to increase error detection coverage. The main advantage of the scheme is that its WDP does not need to monitor memory accesses externally or being involved with the pipeline complexity; this makes the scheme applicable to processors with on-chip caches and pipeline features.

Using Performance Monitoring, the CIC scheme is able to count the number of committed instructions at run-time. The WDP checks the number of instructions calculated at the compile-time and the number of instructions calculated at run-time. The CIC scheme was evaluated using software implemented fault injection (SWIFI) in a Pentium® processor test system. The experimental evaluation is based on injection of 8181 faults. The results show that the error detection coverage varies between to 90.92% and 98.41%, for different workloads.

Next section discusses Performance Monitoring features. Section 3 presents the design of the CIC scheme. The experimental system is given in section 4. In section 5, the experimental results are presented. Section 6 concludes the paper.

2. Performance Monitoring features

Many of current superscalar processors such as AMD x86-64 [1], Alpha [2], Pentium [3], MIPS R10000 [8], PowerPC 604 [10], include features, called Performance Monitoring features, to measure and monitor various parameters related to the performance of the processors.

The Performance Monitoring features use special internal counters, which can be initialized to count the occurrences of several events in the processors. Examples of such events are cache hits, instructions executed and branches taken. Some processors have also special pins, called event-ticking pins, which can signal out the occurrences of internal events of the processors.

In our study, we have used a Pentium® processor whose Performance Monitoring features include two counters called CTR0, CTR1 and two external event-ticking pins called PM0, PM1 pins [3].

3. The CIC scheme

In this section, we present the CIC scheme. To facilitate the discussion, we give the following definitions:
Definition 1: A Branch Free Block (BFB) is a sequence of non-branching instructions in which the execution always enters at the first instruction and leaves via the last instruction [9].
Definition 2: All instructions between two physically consequent BFBs constitute a Partition Block (PB).
Definition 3: A program crash occurs when either the execution illegally continues outside the program or the processor generates an exception as a result of a CFE (e.g. invalid opcode and page fault).

The CIC scheme assumes that the program is partitioned into BFBs and PBs. In this model, we distinguish between seven types of CFEs , as are shown in Figure 1 [9].

Types:

1: from a BFB to other BFB
2: from a PB to itself or another PB
3: from a BFB to itself
4: from a PB to a BFB
5: from a BFB to a PB
6: from a BFB to outside program
7: from a PB to outside program

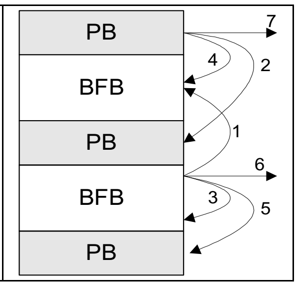

Figure 1 - Program partitions and types of CFEs

When we use CFE in the discussion, we mean the occurrence of an illegal branch, which does not create any subsequent CFEs. The CIC scheme combines six error detection mechanisms implemented in a WDP to detect different types of CFEs. The CIC mechanisms will be explained in the next subsections.

3.1. BFB Internal Instruction Counting (BIIC)

This mechanism internally counts the exact number of instructions executed in a BFB. At the beginning of each BFB, an internal performance counter is set to zero and the pre-calculated number of instructions of the BFB embedded in the program is sent out to the WDP. During execution of the BFB, the content of the internal counter is increased by one for each instruction executed. At the end of the BFB, the content of the counter (run-time calculated) is also sent to the WDP. The WDP checks the two values (i.e., pre-calculated and run-time calculated) and signals the occurrence of an error if any discrepancy exists. The BIIC mechanism detects all CFEs of type 3 and most CFEs of type 1 if no program crash occurs.

3.2. BFB External Instruction Counting (BEIC)

In this mechanism, the number of instructions executed in a BFB is counted externally by the WDP using the processor event-ticking pins. For each BFB, the BEIC mechanism performs the following operations: 1) at the BFB entry point, the pre-calculated number of instructions appearing in the BFB is sent to the WDP, 2) the processor event-ticking pins operate as clock signals to increment a WDP-counter when an instruction is executed (Figure 2), and 3) the WDP signals the existence of an error if the number of instructions executed in run-time exceeds the number of the pre-calculated one. Note that the BEIC mechanism in not as precise as the BIIC mechanism, because the external event ticking pins work with bus frequency, but instruction execution event occurs with core frequency and the bus frequency is less than the core frequency. The BEIC mechanism checks for an upper limit of the number of instructions executed. The mechanism detects all CFEs of type 6. CFEs of type 1, type 3 and type 5 can also be detected if the number of instructions executed violates the upper limit.

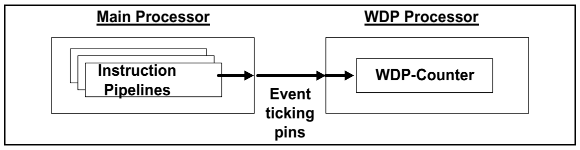

Figure 2 - External mechanisms

3.3. PB External Instruction Counting (PEIC)

The PEIC mechanism checks the maximum number of instructions that are allowed to be executed continually outside the BFBs. In a test run of the program, the maximum number is counted by a Performance Monitoring counter. If the number of instructions executed exceeds the maximum value, the WDP signals the occurrence of an error. This mechanism operates as follows: 1) the maximum number of instructions allowed to be executed outside the BFBs is sent to the WDP immediately after exiting a BFB, 2) the processor event-ticking pins operate as clocks to increment a WDP-counter when an instruction is executed (Figure 2), 3) the WDP indicates the existence of an error if the number of instructions counted in run-time exceeds the maximum value. The PEIC mechanism detects all CFEs of type 7. CFEs of type 2 and type 4 can also be detected if the number of instructions executed within the PBs violates the maximum value.

3.4. Index

This mechanism assigns an arbitrary unique index to each BFB. The unique index of a BFB is sent to the WDP at the beginning and also at the end of that BFB, respectively. The WDP reports for an error if these two indices are different. The Index mechanism detects all CFEs of type 1 if no program crash occurs.

3.5. Entry exit phase checking (Phase)

This mechanism checks the correct order of entering and exiting the BFBs [9]. During error-free operation, an entry to a BFB should always be followed by an exit from that BFB. Hence, a fault, which causes the execution to erroneously pass through two subsequent entry points or two subsequent exit points, can easily be detected. The Phase mechanism detects all CFEs of types 4 and 5 if no program crash occurs.

3.6. Workload timer (WL-timer)

The WL-timer is a timer located in the WDP that checks the total number of instructions allowed to be executed in the whole workload program. This mechanism is especially suitable to capture program crashes and illegal loops. The WL-timer should regularly be reset by the workload program. If no reset occurs in time, the occurrence of an error is signaled.

4. Experimental system

The architecture of the experimental system consists of five parts as are shown in Figure 3: 1) a Pentium board, 2) a WDP, 3) a fault activator logic, 4) an interface logic, and 5) a Host computer. The Pentium board was equipped with a 100 MHz Intel Pentium® processor and Linux OS. This processor was selected due to three reasons: 1) this processor supports Performance Monitoring features, which are functionally equivalent to the features of other Pentium processor families, 2) the implementation is simple and 3) the results can be obtained and evaluated faster. The Pentium® processor has two instructions (WRMSR, RDMSR) to access the internal Performance Monitoring registers. The WRMSR instruction may be executed only in CPL0 (Current Privileged Level 0) [3], which may only be handled through the OS kernel. Note that the CIC scheme is offered for industrial and embedded systems in which I/O operation is an important part and performed through device drivers in kernel mode. In the same manner workload is inserted in Linux kernel to access Performance Monitoring registers and also to communicate with external WDP.

Each time the system resets, workload program runs once. When the workload starts, a Manager program running in the Host computer waits for a random time to issue fault injection command (a pulse on the Pentium® NMI pin). Notice that a custom board is developed to make processor pins accessible.

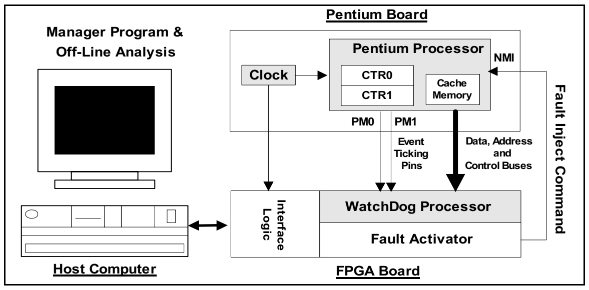

Figure 3 - Experimental system

The WDP, the fault activator logic, and the interface logic were integrated in a board, called the FPGA board equipped with an Altera Flex10k30 FPGA. The WDP decodes the processor address, data and control buses to detect special I/O cycles and also accepts the run-time information. Furthermore, the information on the coverage and latency of the error detection mechanisms are recorded by the WDP. The fault activator has two main tasks: 1) activation of the NMI pin of the Pentium® when a fault is to be injected, and 2) initialization of the registers in the WDP to record the coverage and latency information. The interface logic establishes communication between the Host computer and the WDP as well as the fault activator. The task of the Host computer is to manage and to control the whole experiment. For each fault injection, the Manager program performs several operations in the following order: 1) resetting the Pentium® processor and all registers in the WDP, 2) waiting for program start signal from the Pentium® processor, 3) generating a random time for fault injection, 4) issuing the command for fault activation, 5) reading latency and coverage information from the WDP. The information is stored in a Microsoft Access bank which will be analyzed at the end of the experiment.

4.1. Fault injection method

The fault injection method used in the experiments is

SWIFI. To generate CFEs, the bits of the Program Counter (PC) are changed, one bit for each fault. This is done as follows: 1) the fault activator logic activates the NMI pin of the Pentium® processor, 2) the NMI service routine reads the return address from the stack and changes a bit of the return address and then writes it back to the stack, and 3) after returning from the NMI service routine, the execution continues at an unexpected address due to the change of the value of the return address. A change in the most significant bits of the PC may cause an illegal branch outside the workload program, while a change in the least significant bits of the PC may cause an illegal branch within the workload program. To make the coverage results confident, most of the changes have been concentrated in the least significant bits of the PC (bits 0 through 7) to force the CFEs occurring within the workload program. The Manager program issues the fault injection command randomly in time during the execution of the workload program.

4.2. Workloads and overheads

Three workload programs, all written in C language, were used in the experiment: a linked list (List), an inverse matrix (Matrix) and a quicksort (Qsort). A program, called *postprocessor,* was developed which accepts assembly code of a program and generates a version of that program protected with the redundant instructions needed to implement the CIC scheme, see Figure 4.

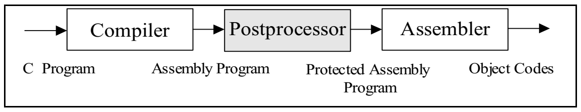

Figure 4 – Generation of a protected program

The structure of the program after inserting the redundant instructions is shown in Figure 5.

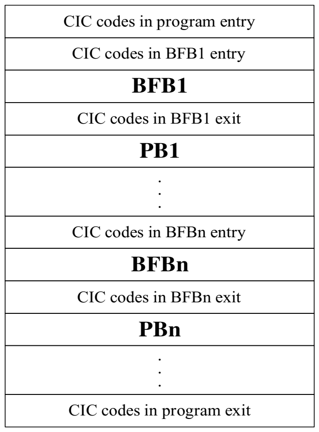

Figure 5 – The CIC scheme inserted in a program

To reduce overheads, only BFBs containing more than four instructions were provided with the redundant instructions. The BFBs containing less than four instructions belong to the PBs of the program. The redundant instructions inserted in the workload programs incur overheads as are summarized in Table 1.

Table 1 – Overhead statistics

	Qsort	Matrix	List
R1	2,651	5,415	11,365
R2	143	126	128
R3	598	530	538
R4	**28%**	**10%**	**5%**
R5	4,999,284	11,551,140	4,170,314
R6	14,480,282	17,610,375	7,646,631
R7	**189%**	**52%**	**83%**

R1: No. of instructions in the original program
R2: No. of protected BFBs
R3: No. of redundant instructions
R4: Overhead of the program size
R5: No. of instructions executed in the original program
R6: No. of instructions executed in the protected program
R7: Overhead of the number of instructions executed

5. Experimental results

This section presents the experimental results on the error detection coverage and the error detection latency. The results are based on 8181 faults, 2727 faults for each workload. The coverage results are presented in Table 2. Note that an error may be detected with more than one mechanism. To determine the effectiveness of the CIC mechanisms, the different types of CFEs are divided into two classes: 1) the CFEs resulted in the program crash, and 2) other CFEs. As shown in the Table 2, the percentages of all errors resulted in program crashes, were 60.67%, 67.50% and 70.50% for Qsort, Matrix and List programs, respectively. All program crashes were detected with the WL-timer. The BEIC and PEIC mechanisms were capable to detect most of program crashes (51.04%+ 48.60%=99.64%) with error detection latency less than WL-timer latency (see Table 4). The BIIC, Index and Phase mechanisms were unable to detect any program crash at all. The results show that only 86.23% of all errors, which did not end to program crashes (other CFEs class), could be detected. This depends mainly on the corresponding coverage obtained for the List program, which was about 69.22%. This may depends in turn on the corresponding coverage of the BEIC and PEIC mechanisms, which were only 0.13% and 1.28%, respectively. This indicates the dependency of the coverage on the different workload programs features such as size and execution time of BFBs and PBs and also workload nature.

Table 2-Fault detection coverage

		Coverage (%)						
Work-loads	Error Classes	BIIC	BEIC	PEIC	Index	Phase	WL-timer	**Total**
Qsort	CFEs resulted in program crashes (60.67%)	0 (0)	54.23 (32.90)	45.65 (27.67)	0 (0)	0 (0)	100 (60.67)	100 (60.67)
	Other CFEs (39.33%)	44.66 (17.57)	33.1 (13.02)	6.8 (2.70)	2.17 (0.85)	36.94 (14.53)	0 (0)	91.71 (36.07)
	All CFEs (100%)	**17.57**	**45.92**	**30.37**	**0.85**	**14.53**	**60.67**	**96.74**
Matrix	CFEs resulted in program crashes (67.50%)	0 (0)	52.32 (35.32)	47.34 (31.95)	0 (0)	0 (0)	100 (67.50)	100 (67.50)
	Other CFEs (32.50%)	18.57 (6.04)	10.15 (3.3)	10.42 (3.39)	0.7 (0.23)	75.82 (24.64)	0 (0)	95.11 (30.91)
	All CFEs (100%)	**6.04**	**38.62**	**35.34**	**0.23**	**24.64**	**67.50**	**98.41**
List	CFEs resulted in program crashes (70.50%)	0 (0)	47.08 (33.19)	52.36 (36.91)	0 (0)	0 (0)	100 (70.50)	100 (70.50)
	Other CFEs (29.50%)	27.59 (8.14)	0.13 (0.04)	1.28 (0.38)	4.85 (1.43)	38.01 (11.21)	0 (0)	69.22 (20.42)
	All CFEs (100%)	**8.14**	**33.23**	**37.29**	**1.43**	**11.21**	**70.50**	**90.92**
Total	CFEs resulted in program crashes (66.22%)	0 (0)	51.04 (33.80)	48.60 (32.18)	0 (0)	0 (0)	100 (66.22)	100 (66.22)
	Other CFEs (33.78%)	31.32 (10.58)	14.48 (5.45)	6 .17 (2.16)	2.49 (0.84)	49.70 (16.79)	0 (0)	86.23 (29.13)
	All CFEs (100%)	**10.58**	**39.26**	**34.33**	**.84**	**16.79**	**66.22**	**95.36**

* The coverage results within parentheses give the coverage of all CFEs.

Table 3- Proportion of errors detected by only one mechanism for all workloads

	Coverage (%)						
Error Classes	BIIC	BEIC	PEIC	Index	Phase	WL-timer	**Total**
CFEs resulted in program crashes	-	-	-	-	-	0.34	0.34
Other CFEs	22.26	2.80	1.02	0.89	53.56	-	80.53
All CFEs	**4.34**	**0.54**	**0.20**	**0.17**	**10.44**	**0.27**	**15.69**

One interesting issue is the proportion of all errors, which was detected by only one mechanism. This result is given in Table 3. As the result shows, the Phase mechanism is a powerful mechanism which could alone detect 10.44% of all errors.

Table 4 shows the mean and the median latencies of each mechanism for different workloads. The latency values are calculated with respect to the CPU external clock frequency which was 66.6 MHz. The mean latencies varied between 80 and 28681 cycles for different workloads and mechanisms excluding the WL-timer. The corresponding values for median latencies varied between 53 and 583 cycles. The mean and median latencies for the WL-timer were in average 1064766 and 1121194, respectively. It is notable that these latencies belong to the errors, which could be detected by the WL-timer alone (0.36%).

Table 4 - Mean and median latency (external clock pulse)

	BIIC		BEIC		PEIC		Index		Phase		WL-timer	
	Mean	Median	Mean	Median	Mean	Median	Mean	Median	Mean	Median	Mean	Median
Qsort	219	218	28584	495	8024	472	103	89	149	77	1,991,486	2,104,837
Matrix	88	90	500	463	507	492	307	307	80	53	1102432	1,103,638
List	210	355	699	532	635	583	95	90	136	81	100,379	155,107
Total	**172**	**221**	**1352**	**497**	**3055**	**515**	**168**	**162**	**122**	**70**	**1064766**	**1121194**

6. Summary and conclusion

COTS superscalar processors are widely used in industrial, embedded, and real-time applications. In these applications, features like reliability, safety, security, and availability are important. In spite of several internal error detectors available in COTS superscalar processors, most contemporary COTS processors have very limited error detection coverage. This paper presents a scheme called CIC to enhance error detection coverage in COTS superscalar processors. The scheme combines a simple external hardware and the Performance Monitoring features of the processors. The CIC scheme includes six different error detection mechanisms to detect different CFEs are participated.

The CIC scheme is a cost-effective, high coverage and low latency approach to enhance error detection in COTS superscalar processors with internal Performance Monitoring features. The scheme does not involve with the processor pipeline complexity, on-chip caches, and memory write buffering features.

7. References

[1] Advanced Micro Devices, Inc., *AMD x86-64 Architecture Programmer's Manual, Volume 2: System Programming*, September 2002.

[2] Compaq Computer Corp., *Alpha Architecture Handbook*, 1998.

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

[4] H. Madeira, 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)*, Mondello, Italy, Sept. 92, pp. 121-132.

[5] H. Madeira and J.G. Silva, "On-line signature learning and checking: experimental evaluation", *proceedings of IEEE CompEuro 91: Advanced Computer Technology, Reliable Systems and Applications*, Italy, May 1991, pp. 642-646.

[6] H. Madeira, J. Camoes and J.G. Silva, "Signature verification: a new concept for building simple and effective watchdog processors", *Proceedings of the IEEE Mediterranean Electrotechnical Conference*, May 1991, pp. 1188-1191.

[7] A. Mahmood and E.J. McCluskey, "Concurrent error detection using watchdog processors - a survey", *IEEE Transactions on Computers*, Feb. 1988, pp. 160 -174.

[8] MIPS Technologies Inc., *MIPS R10000 Microprocessor User's Manual*, Oct. 1996.

[9] G. Miremadi, J. Ohlsson, M. Rimen, and J. Karlsson, "Use of Time, Location and Instruction Signatures for Control Flow Checking", *Proceedings of the DCCA-6 International Conference*, Urbana Champaign, IEEE Computer Society Press, ISBN 3-211-82649, 1998.

[10] Motorola Inc., *PowerPC ™ 604 RISC Microprocessor Technical Summary*, 1994.

[11] N. Oh, P.P. Shirvani and E.J. McCluskey, "Error Detection by Duplicated Instructions In Super-scalar Processors", *IEEE Trans. on Reliability*, Mar. 2002, pp. 63-75.

[12] M. Namjoo and E.J. McCluskey, "Watchdog processors and capability checking", *FTCS-12,* Santa Monica, CA, June 1982, pp. 245-248.

[13] J. Ohlsson and M. Rimen, "Implicit signature checking", *FTCS-25*, 1995, pp. 218 -227.

[14] Kontron Embedded Computers AG, *ETX-P3M User's Guide*, 2003, URL:http://www.kontron.com.

[15] PCI Industrial Computer Manufactures Group, CompactPci Industrial Boards, http://www.picmg.org.

[16] AdvanTech Industrial Computers, http://www.advantech.com.

[17] P. Croll 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.

[18] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto and L. Tagliaferri, "Control-flow checking via regular expressions", *Proceedings of 10th Asian Test Symposium*, Nov. 2001, pp. 229-303.

[19] P. Chevochot and I. Puaut, "Experimental evaluation of the fail-silent behavior of a distributed real-time run-time support built from COTS components", *Proceedings of The International Conference on Dependable Systems and Network*, July 2001, pp. 304 -313.

[20] A. Avizienis, "A fault tolerance infrastructure for dependable computing with high-performance COTS components", *Proceedings International Conference on Dependable Systems and Networks*, June 2000, pp.492 -500.

[21] C.D. Gill, R.K. Cytron and D.C. Schmidt," Multiparadigm scheduling for distributed real-time embedded computing", *Proceedings of the IEEE , Volume 91, Issue: 1* , Jan. 2003, pp. 183 -197.