Self-Organization And Self-Management In Control-Flow Error Mitigation

Saber Fazel

Abstract: This paper presents a software-based technique to mitigate Control-flow Errors (CFEs) in multithreaded programs. In this paper, we concentrate on self-organization and self-management mitigation of control-flow error using additional instructions insertion into specific portions of multithreaded program at design time regarding to control-flow and data-flow dependency graphs. In order to evaluate the proposed technique, three multithreaded benchmarks quick sort, matrix multiplication and linked list utilized to run on a multi-core processor, and a total of 5000 transient faults has been injected into several executable points of each program. The results show that this technique detects and corrects between 91.9% and 93.8% of the injected faults with acceptable performance and memory overheads.

Index Terms: control-flow checking, control-flow error recovery, mutithreaded program, multi-core processor, on-line testing, interprocess communication, backward error recovery.

---

1 Introduction

THECMOS technology improvement has brought the exponential growth in the number of transistor per chip and decreasing feature size. The transistor size decreasing satisfies the industrial high demand for further power reduction but it still does not meet ITRS technology roadmap. To further reduce the power consumption of integrated circuit, advanced research areas such as Carbon Nano Tube based circuit design [3], spintronic-based logic design [4], Quantum-dot cellular automata [2] and power gating techniques [6][7][8] are proposed to address power-budget issue. The transistor size and voltage levels reduction has increased sensitivity of microprocessors to transient faults [5]. Transient and permanent faults can cause uncompensatable damage in computer systems, especially in safety-critical systems. Two main groups of reliable systems proposed in past [26] are: 1) Hardware-based approaches, relying on adding customize hardware, and 2) Software-based approaches, relying on exploiting devised software to achieve fault tolerance. One of typical hardware-based solutions is to use of an external hardware like watchdog (checker) processor to monitor activities carried out by microprocessors. As soon as any misbehavior is observed, suitable fault containment procedures are activated [1][10][20][22][24][25][26]. Adding redundant hardware for fault tolerance would decrease the benefits of modern processors due to imposing extra area and power overheads to the entire system architecture. Software-based approaches improve the dependability of processors through modifying the program code, while the underlying hardware remains unchanged. Numerous software-based error detection and error recovery techniques have been devised to track the CFEs in multi-core processor and restore the operation of the entire system to a safe condition [5][15][17][31][35][38][39][43][44]. Some software-based error detection techniques can be applied automatically to the source code of a program, automatically. These techniques that check program execution flow are known as control-flow checking (CFC) techniques. Lots of CFC techniques have been proposed in the past which are mainly based on signature monitoring principle [9][12][13][18][23][36]. Some of them are processor-independent and can be applied to any kind of processors and microcontrollers. In these approaches firstly, program code is partitioned into basic blocks and secondly, extra instructions are added to each basic block. Basic block includes a maximal set of non-branching instructions, except in the last instruction [10]. A unique signature is assigned to each basic block to provide an authentication checking process at the beginning of each basic block. Then, the flow of program is examined based on signature updating and checking at run-time. If any mismatch has observed, an error is detected and reported [9]. It has been shown that between 33% and 77% of transient faults result in control-flow errors, such as possible errors in program counter (PC), address circuits, steering and control logic, or any kind of control memories [34]. A Control-flow Error (CFE) is said to have occurred if the processor executes an incorrect sequence of instructions [9][40][37][25][16][11][21][27]. Several CFC methods have been designed for detecting each type of CFEs in a program code. However, a few published works have concentrated on CFEs correction [14] and [28][30][33]. A very common approach to mitigate CFE is to return the program flow to the beginning of the source basic block which illegal branch jumps from that location. However, correcting the CFE is not sufficient and the program may fail since there may be some data errors generated by the CFEs [14]. Thus, both CFE and data errors should be recovered to guarantee that the program will be executed safely after CFE mitigation. In multi-core processors, the synchronization and communication dependencies between threads of multithreaded program are required to be considered to guarantee the consistency between threads operation while the CFE mitigation is employed. The previous proposed CFE recovery techniques suffer from high performance and memory overhead which makes them inapplicable in real-time embedded systems. In this paper, we propose a software-based CFE mitigation technique offering following benefits:

- new dependency graph extraction model
- novel CFE detection method considering multi-thread in the program
- in-situ CFE_handler function for agile program flow and

- *Author name is currently pursuing masters degree program in electric power engineering in University, Country, PH-01123456789.*
  E-mail: author_name@mail.com
- *Co-Author name is currently pursuing masters degree program in electric power engineering in University, Country, PH-01123456789.*
  E-mail: author_name@mail.com

data error correction
- consistent and optimized partial checkpointing at predefined location

The proposed technique called Self-organization and Self-management CFE Mitigator (SSCM), significantly reduces the performance and memory overheads in comparison with well-known CFE recovery techniques in this area [14][28][41]. The GCC, a GNU compiler is utilized for simulating the operation of suite benchmarks under fault injection experiments [42]. A total of 5000 transient faults are injected into the program codes. The experimental results show that between 91.9% and 93.8% of CFEs are detected and corrected by SSCM method. The structure of this paper is as follows: Section 2 introduces dependency graph in multithreaded program. Detection and correction phases which are used in SSCM are described in Section 3. Simulation environment and experimental results are presented by Section 4. Finally Section 5 concludes the paper.

2 DEPENDENCY GRAPH IN MULTITHREADED PROGRAM

To represent multithreaded program, we present a dependency graph composed of connecting graphs of all single threads in the program.

2.1 Dependency Graph of a Single Thread

A thread is an entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled [15][17][29]. However, a thread itself is not a program because it cannot run independently. Thread can only be a part of whole program and be executed within a program [20]. Herein, the dependency graph of a single thread is used to represent a single thread in a multithreaded program. This graph consists of a number of Control-flow Graphs (CFGs) and Data-flow Graphs (DFGs). CFG is a graph composed of a set of nodes V and a set of edge E, CFG={V,E}, where V={N1, N2, …, Ni, …, Nn} and E={e1, e2, …, ei, …,en}. Each node Ni represents a basic block and each edge ei represents the branch bri,j from Ni to Nj. CFGs are depicted at compile time and represented control conditions and right transmission between basic blocks. DFGs represent data dependencies between basic blocks. The DFG model implicitly assumes the existence of variables, whose values store the information required and generated by the operations. Each variable has a lifetime that is the interval from its birth to its death, where the former is the time at which the value is generated as an output of an operation and the latter is the latest time at which the variable is referenced as an input to operation. Fig. 1 shows DFG and CFG generated from control dependency between basic blocks and data dependency among variables in these basic blocks. In this figure, solid and dashed arcs are control and data dependencies, respectively.

2.2 Dependency Graph of Multithreaded Program

The Dependency Graph of Multithreaded Program consist of a collection of single-thread depend graphs that each represent a single thread, and some special kinds of dependency arcs to model thread interactions between different threads of a program. These dependency arcs are based on: 1) synchronization between thread synchronization statements and 2) communication between shared variables of the program threads.

1) ***Synchronization Dependencies:*** Running each process on its own address space had the advantage of reliability since no process can modify another process's memory. However, all of a process's threads run in the same address space are able to access to all of the same resources, including memory**Error! Reference source not found.**. While this makes it easy to share data among threads, it also makes it easy for threads to step on each other. Multithreaded programs must be specially programmed to ensure that threads donot step on each other. Moreover, synchronization is necessary to avoid race conditions and deadlockswhen multiple threads want to access shared resources. Fig.2 shows some additional synchronization arc to model synchronization between threads. As shown in Fig. 2, a critical section containing shared data in both basic block N_3 of thread t_1 and N_1 of thread t_2 should be access exclusively through the use of synchronization methods.

2)

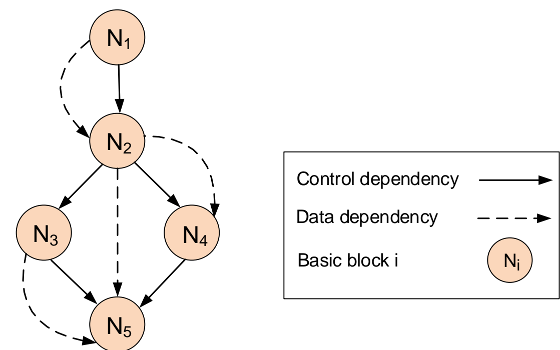


Fig. 1. *Dependency graph of a single thread*

DDR3 Controller

Fig. 2. *Dependency graph of multithreaded program in a Multicore processor*

3) ***Communication Dependencies:*** Communication dependency is used to capture dependency relations between different threads because of inter-thread communication. As shown in Fig. 2, node $N_i$of a thread is directly communication-dependent on node N_j in another thread where *i,j* are basic block number if the value of a variable computed at N_i has direct influence on the value of a variable computed at N_j through an inter-thread communication. Shared memoryis often used to support communication among threads. Communications may

occur when two parallel executed threads exchange their data via shared variables.

4) ***Constructing the Dependency Graph of Multithreaded Program:*** To construct the dependency graph of multithreaded program, the single-thread dependency graph of all program threads using synchronization and communication dependency arcs should be combined. For this purpose, firstly, single-thread version is constructed with considering control dependency between basic blocks and data dependency among variables of each basic block and secondly multithread version is created based on synchronization and communication dependencies between different threads of multithreaded program as shown in Fig. 2. In this figure, bolded dotted and dashed arcs are synchronization and communication dependencies, respectively.

3. THE PROPOSED SELF-ORGANIZATION AND SELF-MANAGEMENT CFE MITIGATOR (SSCM)

The SSCM technique recovers both CFEs and data errors which are occurred in multithreaded programs in multi-core architectures using dependency graph consideration and partial check pointing. Two phases are employed to recover both CFEs and data errors in multithreaded programs.

3.1 Control-flow Error Detection in SSCM

As shown in Fig. 3, source signature of thread *j* (SST_j)is a shared variable in a shared memory which contains the run-time signature of thread *j* and continuously updated in executed nodes, where *j* shows thread number of multithreaded program and finally stored the signature of the basic block in which a CFE has occurred. Each shared variableSST_jwhich keeps signature of thread *j* is allowed to be updated only in thread *j*. Another variable *T* keeps number of thread which is currently executed and it determines source thread in the case of an inter-thread CFE. The $SST_j$is updated with unique number in each basic block $N_i$where *i* shows basic block number of thread *j*, for checking the control-flow of the program. For updating SST_j, XOR operation is used. This *operation* is performed *between last $SST_j$and a number which is calculated at compile time.* Under normal execution of the program, SST_jshould equal calculated signature in node N_i at ompile time. If SST_jcontains a number different from the calculated signature in node N_i, an error has occurred in the program.

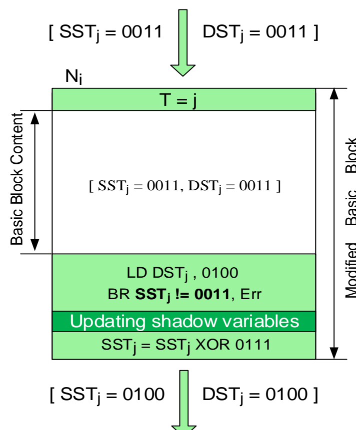


Fig. 3. *Basic block scheme in the SSCM*

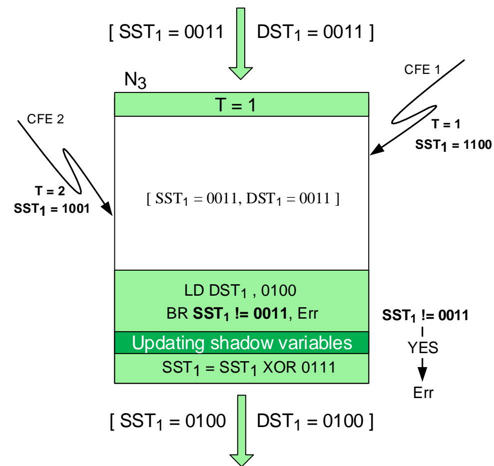


Fig. 4. *Intra and inter thread detection*

As shown in Fig. 4, *CFE1* is an intra-thread CFE because source thread of CFE is itself (*T=1*). If an illegal branch jumpedto instructions beforechecking instructions at the end of basic block (*CFE1* in Fig.4) and control transferred to it illegally,then *CFE1* can be detected by comparing the stored value in the SST_1(as the signature of the node) with another one calculated at compile time. The*CFE2* is an inter-thread CFE (an illegal branch from one basic block of threadt_2 to a basic block of thread t_1 in the same processor (*T=2*)). These types of CFE can be detected by comparing last updated signature of executed thread with expected value. The signature of last executed basic block of thread t_1 is stored in SST_1 before switching the CPU from thread t_1 to thread $t_2$and it is equal 1001. The inter-thread *CFE2* can be detected by comparing last updated SST_1 (1001) with expected value at the end of basic block.

3.2 Control-flow Error Recovery in SSCM

If the data get dirty due to CFE, the previous proposed methods cannot guarantee that the data will be corrected through transferring back the program flow to the source basic block whereby illegal branch has occurred. There are two well-known recovery methods offering data error correction [8][9][32]:

1) Forward Error Recovery: In forward error recovery techniques, the nature of errors and damage caused by faults must be completely evaluated.
2) Backward Error Recovery: In backward error recovery techniques, the nature of faults is not needed to be predicted. If any error occurs in the system, the process state is restored to previous error-free state.

In this work, we utilized the backward error recovery which consists of three steps as following:

1) Periodical check pointing the error-free state,
2) Restoration when a fault occurred,
3) Restart from the restored state.

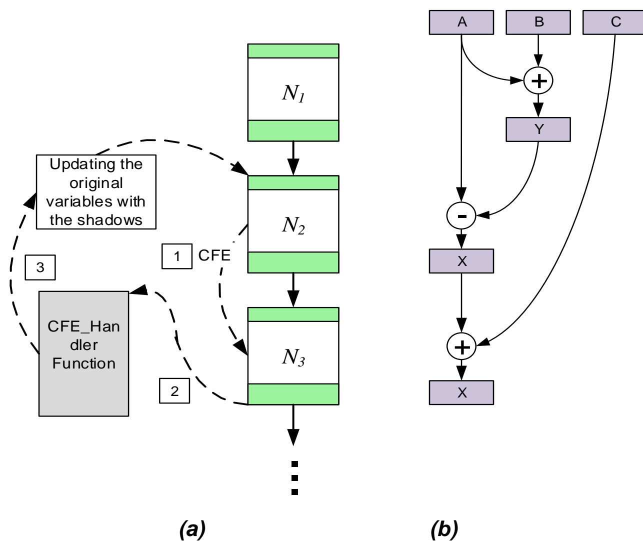


Fig. 5. (a) *Intra-thread CFE correction, (b) DFG generated from program code*

In addition, we combined the backward error recovery with user triggered checkpointing strategy in which the user has the knowledge of the computation being performed and can decide the location of the checkpoints. Thus, the error recovery mechanism is triggered at the end of each basic block in which the corresponding original variables has been modified as shown in Fig. 3. Therefore, the shadow (checkpoint) variables always contain the true values of the original ones. These values are trustable for correcting the generated data errors in correction phase.

1) Intra-thread CFE Correction: If an illegal branch jumps from one block (*source block*) to another block (*destination block*) in the same thread, original variables which are used in both source and destination block may corrupt. So, source signature of thread *j* (SST_j) and destination signature of thread *j* (DST_j) are used to keep signature of source and destination block, respectively for correcting corrupted variables in both source and destination block when a CFE occurred in thread *j*. Shadow (checkpoint) variables always contain the true values of the original ones and they are updated at the end of basic block to store values of variables which are modified in basic block. When a CFE is detected through added instructions, control is transferred to *CFE_handler function*. This function can transfer back the control to the source basic block and restore the original variables with considering the relation between shadow variables, shared variable *T* and both $SST_j$and DST_j. Fig. 5 shows three basic blocks from the set of basic blocks in a program code as well as the dependency generated from data dependencies among variables and control dependency between basic blocks. Using the SSCM, the control is transferred to *CFE_handlerfunction* as soon as CFE is detected (step 2 in Fig. 5). *CFE_handler function* determines type of CFE by comparing *T* and expected thread number. If an intra-thread CFE had occurred, original variables in destination and source basic block are updated with their shadow ones. Finally, control is transferred to the address of first instruction in the source basic block (step 3 in Fig. 5) and thecode is re-executed from this point. For optimizing the SSCM, the temporal and local variables can be ignored to avoid the shadows updating operation. Thus, the shadows are only considered for global variables which are alternatively used in the program code.

2) Inter-thread CFE Correction: An inter-thread CFE can be caused by stack pointer faults result in a faulty procedure return. In order to mitigate inter-thread CFEs the concept of create/join relations are required to be considered. In create/join relations, the main thread should wait for the slave ones until the execution of them run out, if the main thread needs the results which will be provided by the slave ones. This operation has been also simplified through a predefined function called the *pthread_join.* If an illegal branch jumped to the sequential section after basic block containing "*join*" instruction before the synchronization execution, main thread continue with incorrect results which have provided by the slave ones. For example, assume an illegal branch occurred from $N_2$of thread $t_1$toN_4 of thread t_2. This inter-thread CFE can be detected by comparing last updated $SST_{destination\ thread}$ with expected value at the end of basic block (SST_2!= 0100) as illustrated in Fig. 6. After the CFE detection, shared variable *T* and both signature of thread ($SST_j$and DST_j) are givento*CFE_handler function* to correct control-flow and data errors. If an inter-thread CFE had occurred, original variables in destination basic block of destination thread are updated with their shadow ones and the program flow can be re-executed from the last error-free point.

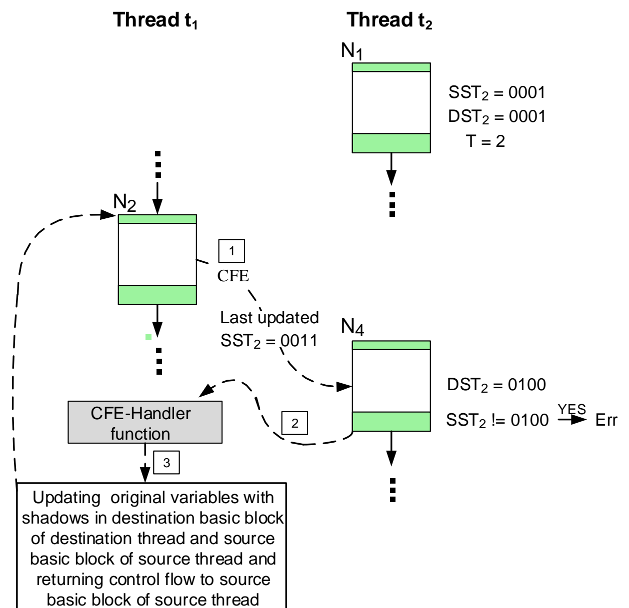


Fig. 6. *Inter-thread CFE correction*

4 EXPERIMENTAL RESULTS

Inorder to evaluate the SSCM, the GCC, a GNU compiler, has been used for running the program on a multi-core system (CPU=i7-740QM, RAM=6GB, OS=Linux Ubuntu10). Three well-known benchmarks utilized by previous CFE recovery techniques have been used to execute on multi-core processor system. These suite benchmarks are as following:

1) **Quick Sort (QS):** in which the main thread first partitions the 100-elements array of integers into two parts, by performing one round of the quick sort algorithm, then assigns each sub-arrays to a slave thread in order to sort each part separately and simultaneously.

2) Linked List (LL): in which the main thread makes slave threads responsible to build half of the linked list separately and concurrently. After joining the slave threads to the main thread, it connects two splits in order to build a single linked list.

3) Matrix Multiplication (MM): in which the main thread makes slave threads responsible to compute each elements of the product separately and concurrently.

About 5000 transient faults have been injected on the several points of the programs. The considered fault models were**Error! Reference source not found.**:

- Branch insertion: it had occurred when one of the non-control instructions in the program was replaced by a control instruction, and the control instruction always causes a taken branch,
- Branch deletion: it had occurred when one of the control instructions in the program was changed to a NOP instruction.
- Branch target modification: This type of CFE occurred when target address of a control instruction is modified as the result of a fault and the control instruction actually causes a taken branch.

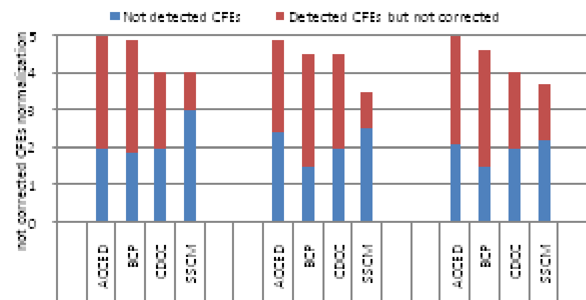
These fault model are used for both intra and inter thread CFEs. In the 80x86 processor, a complete thread switching can be implemented using twelve assembly language instructions. Registers of the processor are pushed onto the stack at the beginning, then stack pointer is changed, and finally they are popped at the end for injecting inter-thread CFEs. Error correction coverage shows the benefits of the SSCM to mitigate both inter-thread and intra-thread CFEs.It is said that a CFE has been corrected if the program exits normally with correct output. Table 1 shows the error correction coverage of the SSCM compared to ACCED [28], CDCC [14] andBasic Check-Pointing (BCP). It is found that about 7.6% of the injected faults return correct output without using any technique.

Table 1. *Experimental fault injection results for multithreaded benchmarks*

Benchmarks	Original		ACCED [28]		BCP		CDCC [14]		SSCM	
	Wrong Results	*Correct Results*	*Wrong Results*	*Correct Results*	*Wrong Results*	*Correct Results*	*Wrong Results*	*Correct Results*	*Wrong Results*	*Correct Results*
	%	%	%	%	%	%	%	%	%	%
QS	92.4	7.6	5.5	94.5	8.8	91.2	7.5	92.5	7.9	92.1
MM	93.7	6.3	4.7	95.3	6.6	93.4	5.2	94.8	6.2	93.8
LL	91.3	8.7	6.1	93.9	8.4	91.6	8.5	91.5	8.1	91.9
Average	92.4	7.6	5.1	94.9	7.7	92.3	7.1	92.9	7.4	92.6

As shown in Table 1, about 92.6% of the injected faults return correct output using SSCM. On average, about 7% of the faults resulted in segmentation faults for the under experimenttechniques. Segmentation faults are generated if the illegal branches jump to signature update statements or a *push* or *pop* instruction re-execution due to CFE correction. The CFEs which are not corrected in SSCM can be classified as follows:

1) Not detected CFEs are included:
   - Inter-thread CFE to basic block of another thread where if CPU switch to that thread, start from that basic block
   - Intra-node CFE (an illegal movement within a basic block)
   - Illegal branch to *CFE_handler function*
2) Detected CFEs but not corrected are included:
   - Illegal jump to *critical instructions*
   - CFEs in communication-dependent threads when executed thread work with shared data which are corrupted in previous thread



The Fig. 7 illustrates the normalization of not corrected CFE for all four methods. The *CFE_handler function* of SSCM is larger than other three methods which results in the probability of illegal branch to this section significantly increase. However, SSCM offers higher inter-thread CFE recovery compared to other methods due to employing novel approach to detect and mitigate them. C*FE detection latency* is equal to the time between fault occurrence and the time at which the CFE caused by the injected fault is detected. The CFE detection latencyof SSCM is slightly more than other techniques because of employing checking instruction at the end of each basic block.

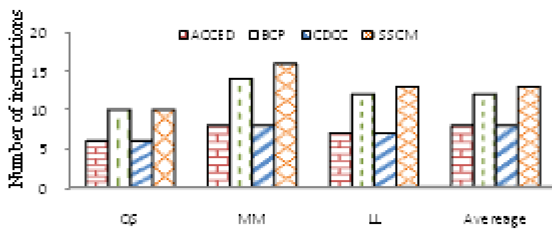


Fig. 8 shows the CFE detection latency observed during fault injection in terms of the number of instructions executed before the CFE is detected. *CFE correction latency* of SSCM can be computed as the summation of the following factors:

- CFE detection latency
- Latency of the *CFE_handler function* execution
- Latency of updating corrupted variables with shadow ones
- Latency of control transferring to source basic block

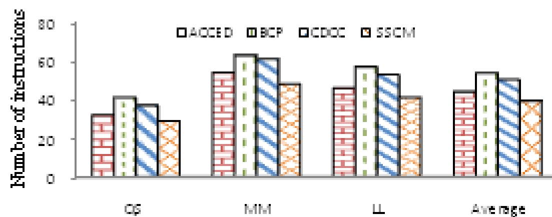


Fig. 9 illustrates the CFE correction latency observed during fault injection in terms of the number of instructions executed for restoring the CFE and correcting data errors. The execution latency of SSCM is less than other techniques because using an optimized checkpointing and*CFE_handler function.*

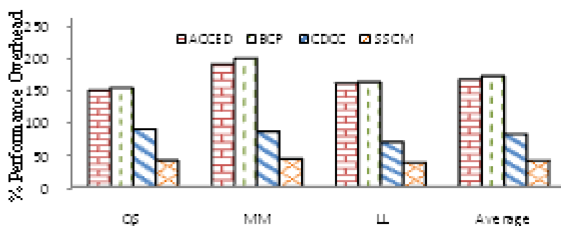


Fig. 10 illustrates the *performance overhead* comparison between SSCM and three other techniques. Yet again, the checkpinting and *CFE_handler function* optimization are the main reasons that the performance overhead of SSCM is less than three other methods.

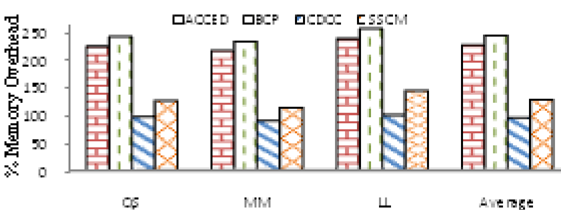


Fig. 11 illustrates the comparison among *memory overhead* percentages of the programs due to applying the methods. The memory overhead (cost) consists of the set of instructions which are added at the beginning and at the end of the basic blocks and the other set added for implementing the *CFE_handler function*. The memory (performance) overhead of the ACCED is comparatively (about 100%) higher than the SSCM because of adding duplicated instructions and executing the set of instructions used for comparing the results to obtain correct output.Performance overhead of the *Matrix Multiplication* and the memory overhead of the *Linked List* are totally higher than the others. The matrix multiplication has many computational operations, and the basic blocks of its code are larger than the other ones. Consequently, for correcting a CFE, the instructions which should be re-executed is more than other benchmarks. In contrast to matrix multiplication, linked list program has the fewest computational operations, and the number of the basic blocks separated in the linked list is more than the other benchmarks. Therefore, the total of added instructions at the end of the basic blocks is a large number in compare to the others.

5 Conclusion

In this paper, a software technique to detect and correct CFEsin multithreaded programs was proposed. This technique was implemented through considering control and data dependencyin an individual thread besidesynchronization and communicationdependency in multi-thread program at compile time. Furthermore, the SSCM corrects data errors generated by CFEsthat can cause considerable corruptions in the systems (especially in the safety critical applications).Fault injection experiments showed that the SSCM, when applied on the programs, produce correct results in over 92.6% of the cases.The latency and the additional memoryrequired for correcting the CFEs and the data errors are considerably less than the duplication based and checkpoint based methods which have been recently published.

References

[1] M. Fazeli, R. Farivar and S. G. Miremadi, “Error Detection Enhancement in PowerPC Architecture-based Embedded Processors,” Journal of Electronic Testing: Theory and Applications, vol. 24, pp. 21-33, 2008.

[2] A. Roohi, R. F. DeMara, N. Khoshavi, “Design and Evaluation of an Ultra-Area-Efficient Fault-Tolerant QCA Full Adder,” Elsevier Journal of Microelectronics, pp. 531-542, 2015.

[3] MFL De Volder, SH Tawfick, RH Baughman, AJ Hart, “Carbon Nanotubes: Present and Future Commercial Applications,” American Association for the Advancement of Science, pp. 535-539, 2013.

[4] R. Venkatesan, M. Sharad, K. Roy, A. Raghunathan, “Energy-Efficient All-Spin Cache Hierarchy Using Shift-Based Writes and Multilevel Storage,” ACM Journal on Emerging Technologies in Computing Systems, 2015.

[5] A. Mohamed, M. Zulkernine, "A Control Flow Representation for Component-Based Software Reliability Analysis," IEEE Sixth International Conference on Software Security and Reliability, 2012.

[6] R. Ashraf, A. Al-Zahrani, N. Khoshavi, R. Zand, S. Salehi,

A. Roohi, R. F. DeMara and M. Lin “Reactive Rejuvenation of CMOS Logic Paths using Self-Activating Voltage Domains,” IEEE International Symposium on Circuits and Systems, pp. 2944-2947, 2015.

[7] M. Saber Golanbari, S. Kiamehr, M. Ebrahimi, M. B. Tahoori, “Aging guardband reduction through selective flip-flop optimization,” 20th IEEE European Test Symposium, 2015.

[8] N. Khoshavi, R. A. Ashraf, and R. F. DeMara, “Applicability of Power-Gating Strategies for Aging Mitigation of CMOS Logic Paths,” IEEE 57th International Midwest Symposium on Circuits and Systems, pp. 929-932, 2014.

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

[10] A. Mahmood and E. J. McCluskey, “Concurrent Error Detection Using Watchdog Processors-a survey,” IEEE Transactions on Computers, vol. 37, no. 2, pp. 160-174, 1988.

[11] N. Khoshavi, H. R. Zarandi, M. Maghsoudloo, “Control-Flow Error Detection Using Combining Basic and Program-Level Checking in Commodity Multi-core Architectures,” 6th IEEE Symposium on Industrial Embedded Systems (SIES’11), pp. 103-106, 2011.

[12] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy and J. A. Abraham, “Design and Evaluation of System-level Checks for On-line Control Flow Error Detection,” IEEE Transactions on Parallel and Distributed Systems, vol. 10, no. 6, pp. 627-641, 1999.

[13] O. Goloubeva, M. Rebaudengo, M. R. Sonza and M. Violante, “Soft-error Detection Using Control Flow Assertion,” 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 57-62, 2003.

[14] H. R. Zarandi, M. Maghsoudloo and N. Khoshavi, “Two Efficient Software Techniques to Detect and Correct Control-flow Errors,” 16th IEEE Pacific Rim International Symposium on Dependable Computing, pp. 141-148, 2010.

[15] M. Prvulovic, Z. Zhang and J. Torrellas, “ReVive: Cost Effective Architectural Support for Rollback Recovery in Shared Memory Multiprocessors,” 29th International Symposium on Computer Architecture, pp. 111-122, 2002.

[16] M. Maghsoudloo, H. R. Zarandi, S. Pourmozaffari, N. Khoshavi, "Soft Error Detection Technique in Multi-threaded Architectures Using Control-flow Monitoring," 14th EUROMICRO Conference on Digital System Design Architecture, Methods and Tools, pp. 789-792, 2011.

[17] K. Wu, W. K. Fuchs and J. H. Patel, “Error Recovery in Shared Memory Multiprocessors Using Private Cashes,” IEEE Transactions on Parallel and Distributes Systems, vol. 1, Issue 2, pp. 231-240, 1990.

[18] T. Lanfang, T. Ying, X. Jianjun, "CFEDR: Control-flow error detection and recovery using encoded signatures monitoring," IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, 2013.

[19] R. Vemu and J. A. Abraham, “CEDA: Control-flow Error Detection through Assertions,” 12th IEEE International On-Line Testing Symposium, July, pp. 151–158, 2006.

[20] I. Majzik, W. Hohl, A. Patericza and V. Sieh, “Multiprocessor Checking Using Watchdog Processors,” Journal of Computer Systems - Science & Engineering, vol. 11, No. 5, pp 123-132, 1996.

[21] M. Maghsoudloo, H. R. Zarandi, N. Khoshavi, “Low-Cost Software-Implemented Error Detection Technique,” 3rd International Symposium on Electronic System Design, India, 2011.

[22] A. Rajabzadeh and S. G. Miremadi, "CFCET: A Hardware-Based Control Flow Checking Technique in COTS Processors Using Execution Tracing," Elsevier Journal of Microelectronics and Reliability, vol. 46, pp. 959-972, 2006.

[23] A Paya, D Marinescu“Energy-aware Load Balancing and Application Scaling for the Cloud Ecosystem,”IEEE Transactions on Cloud Computing, DOI 10.1109/TCC.2015.2396059.

[24] A. Rajabzadeh, S. 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.

[25] N. Khoshavi, H. R. Zarandi, M. Maghsoudloo, "Control-Flow Error Recovery Using Commodity Multi-core Architecture Features," 17th IEEE International On-Line Testing Symposium, pp. 190-191, 2011.

[26] P. Bernardi, L. V. Bolzani, M. Rebaudengo, M. S. Reorda, F. Vargas and M. Violante, “On-line Detection of Control-Flow Errors in SoCs by means of an Infrastructure IP core,” 35th International Conference on Dependable Systems and Networks, pp. 50-58, 2005.

[27] M. Maghsoudloo, N. Khoshavi, H. R. Zarandi, “CCDA: Correcting Control-flow and Data Errors Automatically,” 15th International Symposium on Computer Architecture and Digital Systems, pp. 105-110, 2010.

[28] R. Vemu, S. Gurumurthy and J. A. Abraham, “ACCE: Automatic Correction of Control-flow Errors,” IEEE International Test Conference, pp. 1-10, 2007.

[29] A Paya, DC Marinescu,“Energy-aware Load Balancing Policies for the Cloud Ecosystem,”IEEE 28th International Parallel & Distributed Processing Symposium Workshops, pp. 823-832.

[30] N. Khoshavi, H. R. Zarandi, M. Maghsoudloo, “Two

Control-flow Error Recovery Methods for multithreaded Programs Running on Multi-core Processors", FactaUniversitatis, Series: Electronics and Energetics, 2015.

[31] A. Maheshwari, W. Burleson and R. Tessier, "Trading Off Transient Fault-tolerance and Power Consumption in Deep Submicron VLSI Circuits," IEEE Transactions on VLSI Systems, vol. 12, no. 3, pp. 299-311, 2004.

[32] S. K. Gupta, R. K. Chauhan and P. Kumar, "Backward Error Recovery Protocols in Distributed Mobile System: A Survey," Journal of Theoretical and Applied Information Technology, pp. 337-347, 2008.

[33] M. Maghsoudloo, H. R. Zarandi, N. Khoshavi "On-line Control-flow Error Detection and Correction based on Monitoring Both Data-flow and Control-flow Graphs," The CSI Journal on Computer Science and Engineering, vol. 10, no. 2 & 4 (b), pp. 10-19, 2012.

[34] J. Ohlsson, M. Rimen and U. Gunneflo, "A Study of the Effects of Transient Fault Injection Into a 32-bit Risc with Built-in Watchdog," 22nd International Symposium on Fault Tolerant Computing, pp. 316-325, 1992.

[35] K. Nepar, N. Alves, J. Dworak and R. I. Bahar, "Using Implications for Online Error Detection," IEEE International Test Conference, pp. 1-10, 2008.

[36] A. Li and B. Hong, "On-line Control Flow Error Detection using Relationship Signatures among Basic Blocks," Journal of Computers and Electrical Engineering, pp. 132-141, 2010.

[37] N. Khoshavi, H. R. Zarandi, M. Maghsoudloo, "Two Control-flow Error Recovery Methods for Multithreaded Programs Running on Multi-core Processors," 28th International Conference on Microelectronics (MIEL'12), Serbia, 2012.

[38] R. Vemu, A. Jas, J. A. Abraham and S. Patil, "A Low-cost Concurrent Error Detection Technique for Processor Control Logic," Proceedings of Design, Automation and Test in Europe, pp. 897-902, 2008.

[39] DC Marinescu, A Paya, JP Morrison, P Healy "Distributed Hierarchical Control versus an Economic Model for Cloud Resource Management,"IEEE Transactions on Cloud Computing, 2015.

[40] M. Maghsoudloo, H. R. Zarandi, N. Khoshavi "An Efficient Adaptive Software-Implemented Technique to Detect Control-Flow Errors in Multi-Core Architectures," Elsevier Journal of Microelectronics Reliability, vol. 52, Issue 11, pp. 2812-2828, 2012.

[41] N. S. Bowen and D. K. Pradhan, "Virtual Checkpoints: Architecture and Performance," IEEE Transactions on Computers, vol. 41, no. 5, pp. 516-525, 1992.

[42] http://www.gnu.org/software/gdb/

[43] P Healy, S Meyer, J Morrison, T Lynn, A Paya, DC Marinescu"Bid-Centric Cloud Service Provisioning,"IEEE 13th International Symposium on Parallel and Distributed Computing, pp. 73-81.

[44] A Paya, DC Marinescu"Cloud-based Simulation of a Smart Power Grid,"IEEE 28th International Parallel & Distributed Processing Symposium Workshops, pp. 875-884.