**FACTA UNIVERSITATIS**
Series: **Electronics and Energetics** Vol. 28, N$^{o}$ 3, September 2015, pp. 309 - 323
DOI: 10.2298/FUEE1503309K

# TWO CONTROL-FLOW ERROR RECOVERY METHODS FOR MULTITHREADED PROGRAMS RUNNING ON MULTI-CORE PROCESSORS

**Navid Khoshavi, Hamid R. Zarandi, Mohammad Maghsoudloo**

Amirkabir University of Technology (Tehran Polytechnic)

**Abstract**. *This paper presents two control-flow error recovery techniques, CFE Recovery using Data-flow graph Consideration and CFE Recovery using Macro block-level Check pointing. These techniques are proposed with regards to thread interactions in the programs. These techniques try to moderate the high memory and performance overheads of conventional control-flow checking techniques. The proposed recovery techniques are composed of two phases of control-flow error detection and recovery. These phases are designed by means of inserting additional instructions into program at compile time considering dependency graph, extracted from control-flow and data-flow dependencies among basic blocks and thread interactions in the programs. In order to evaluate the proposed techniques, five multithreaded benchmarks are utilized to run on a multi-core processor. Moreover, a total of 10000 transient faults have been injected into several executable points of each program. Fault injection experiments show that the proposed techniques recover the detected errors at-least for 91% of the cases.*

**Key words**: *control-flow checking, control-flow error recovery, multi-threaded programs, multi-core processors.*

## 1. INTRODUCTION

Recently, multi-core processors have introduced as viable way to keep performance improvement rates within a given power budget [11]. Multithread programming energized performance of multi-core processors by extracting thread level parallelism from the sequential program flow. When a sequential program is parallelized conventionally, the programmer or compiler needs to ensure that threads are free of data dependences. If data dependences do exist, threads must be carefully synchronized to ensure that no violations occur. Additionally, advances in CMOS technology have provided reduction in transistor size and voltage levels. Reduction in transistor size and voltage levels coupled with increased sensitivity of microprocessors to transient faults. One of the major threats in

Received February 9, 2015
**Corresponding author**: Mohammad Maghsoudloo
Computer Engineering and IT Department, Amirkabir University of Technology (Tehran Polytechnic), No. 424, Hafez St., Tehran, Iran
(e-mail: m.maghsoudloo@aut.ac.ir)

modern microprocessors is transient faults which induced by energetic particle strikes, such as high-energy neutrons from cosmic rays, and alpha particles from decaying radioactive impurities in packaging and interconnect materials [13]. It has been shown that considerable fraction of transient faults, between 33% and 77%, reflects control-flow errors, such as possible errors in program counter (PC), address circuits, steering and control logic [12]. A Control-flow Error (CFE) is said to have occurred if the processor executes an incorrect sequence of instructions [1].

Numerous software-based CFE detection techniques have been devised to assess processor errors [2], [3], [5], [6], [7], [8], [9], [14]. In these approaches firstly, program code is partitioned into basic blocks and secondly, extra instructions are added to each basic block in order to verify the flow of code execution. Basic block includes a maximal set of ordered non-branching instructions (except in the last instruction) [2]. A unique signature is assigned to each basic block at design time. Signatures also are calculated at run-time and next compared with the original ones. If any mismatch has observed (by the added instructions), an error is detected and reported.

Unfortunately, only a few published works have concentrated on CFEs correction [4], [10]. After the CFE is detected, control should be transferred back to the block in which illegal branch was occurred. However, correcting the CFE is not sufficient and the program may fail since there may be some data errors generated by the CFEs [4]. Therefore, any data errors caused by CFE should be corrected after or during correcting the CFE, as well. Error recovery techniques are classified into two broad categories: forward error recovery (FER) and backward error recovery (BER). FER techniques detect and correct the errors without requiring roll-back to a previous correct state. The primary cost of FER schemes is the redundant hardware. Backward Error Recovery (BER) techniques periodically save (checkpoint) system state and roll-back to the latest validated checkpoint when a fault is detected.

In multi-core systems, since all processors share a single view of data and the communication between processors, the method which corrects CFEs and data errors should take into account synchronization and communication dependencies between threads of multithreaded program. Furthermore, the high memory and performance overheads of these techniques can be problematic for real-time embedded systems which have tight memory and performance budget.

Therefore, regarding the importance of handling the CFEs, unsuitability of the conventional related techniques in the modern processors and high memory/performance overheads of previous CFE recovery techniques, a BER CFE recovery technique is proposed in this paper. While previous techniques utilized two set of instructions at the beginning and end of each basic block, the proposed CFE detection method only use a set of checking instructions at the end of each basic block and it has fewer checking instructions in compare to mentioned techniques. To correct CFE and data errors in our approach, we also use a checkpoint-based method like MCP technique [Ref], but checkpoint instructions are added to particular basic blocks regarding the location of basic blocks in dependency graph and acceptable latency for CFE recovery.

Simulation fault injection is used to evaluate recovery capability of the proposed technique. To evaluate the technique, five modified multithreaded benchmarks are used and the GNU debugger, *GDB* [15] has been used to inject faults on the program. It has been shown that using the approaches presented in the paper, can recover more than 91% of the detected errors with about 67% performance overhead and 89% memory overhead.

The structure of this paper is as follows: Section 2 introduces dependency graph in multithreaded program. Section 3 introduces control-flow error detection technique. Section 4 describes different check-pointing used in our approach. The proposed recovery technique is described in section 5. Simulation environment and experimental results are presented by section 6. Finally Section 7 concludes the paper.

## 2. DEPENDENCY GRAPH IN MULTITHREADED PROGRAM

A multithreaded program, running on the multi-core systems, has a number of threads that each one has its own control-flow and data-flow. These flows are not independent since inter-thread synchronizations and communications may exist in the program. In order to represent multithreaded program, we present a dependency graph. This graph is composed of connecting graphs of all single threads in the program, using dependency arcs between different threads.

### 2.1. Single-threaded dependency graph

The single-threaded dependency 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 $V=\{N_1, N_2, ..., N_i, ..., N_n\}$ and $E=\{e_1, e_2, ..., e_i, ...,e_n\}$. Each node $N_i$ represents a basic block and each edge $e_i$ represents the branch $br_{i,j}$ from $N_i$ to $N_j$. As shown in Fig. 1, CFGs and DFGs are depicted at compile time and represented control conditions and data dependencies between basic blocks.

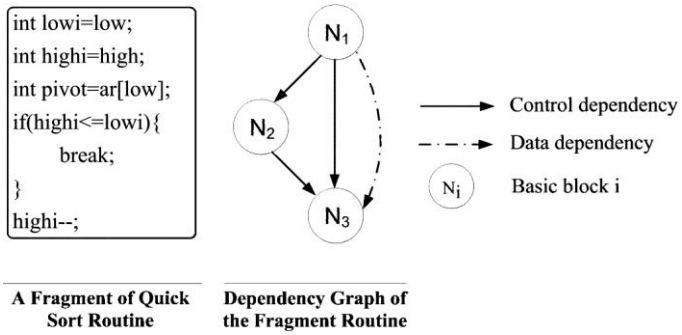



**Fig. 1** Single thread dependency graph.

### 2.2. Multi-threaded dependency graph

Extracting the CFG from relations among basic blocks of a program code is always considered as prerequisite step in both of software- and hardware-based CFC methods. Any incorrectness and limitation in capturing the control dependencies among nodes of the CFG causes that the flow of a given program will not be precisely followed in checking phase. The multithreaded program dependency graph consist of a collection of single thread dependency graphs that each represent a single thread, and some special kinds of dependency arcs to model thread interactions. These dependency arcs are based on: 1) synchronization between thread synchronization statements and 2) communication between shared variables of the program threads.

*2.2.1. Synchronization dependencies*

Multithreaded programs must be specially programmed to ensure that threads do not step on each other. A section of a code that modifies data structures shared by multiple threads is called a critical section. It is important that a critical section should be accessed exclusively by each thread. Synchronize access ensure that only one thread can execute in a critical section at a time. Synchronization dependency among different threads may be caused in two ways: create/join relations, lock/unlock relations. Fig. 2 shows some additional synchronization arc to model synchronization between threads.

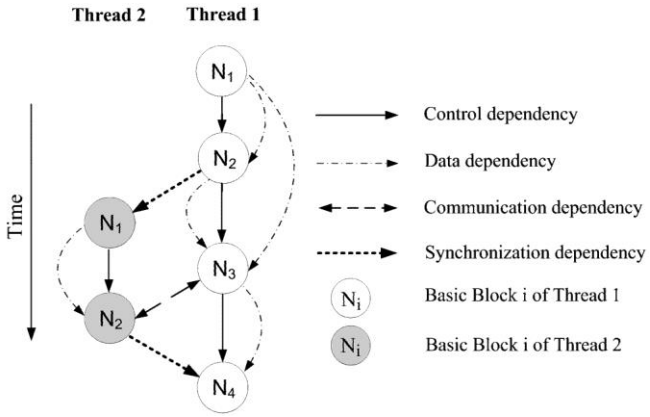



**Fig. 2** Multithreaded program dependency graph.

*2.2.2. Communication dependencies:*

Communication dependency is used to capture dependency relations between different threads because of inter-thread communication. If the value of a variable computed at node $N_i$ of a thread has direct influence on the value of a variable computed at node $N_j$ of other thread through an inter-thread communication, there is a communication dependency among mentioned threads. Shared memory is often used to support communication among threads. To construct the dependency graph of a multithreaded program, firstly, single thread dependency graph is extracted and next, synchronization and communication dependencies are considered between different threads of multithreaded program as shown in Fig. 2. In this figure, bolded dotted and bidirectional dashed arcs are synchronization and communication dependencies, respectively.

## 3. CONTROL-FLOW ERROR DETECTION SCHEME

The CFE detection methods used in the CRDC and the CRMC are quite similar, and the differences between the proposed methods which have emerged in Fig. 3, are only generated because of applying different types of recovery. CFEs can be divided into three types in multithreaded programs: intra-node, inter-node/intra-thread and inter-thread.

An intra-node CFE is an illegal movement within a basic block (CFE2 in Fig. 4), and inter-node or intra-thread CFE is an illegal movement between two blocks of a thread

(CFE1 in Fig. 4). Inter-thread CFE is an illegal jump from basic block of a thread to basic block of another thread in the same processor (CFE3 in Fig. 4). While our CFE detection approach is capable to detect inter-node/intra-thread and inter-thread CFEs, as well as possible, it does not have enough power to detect intra-node CFEs.

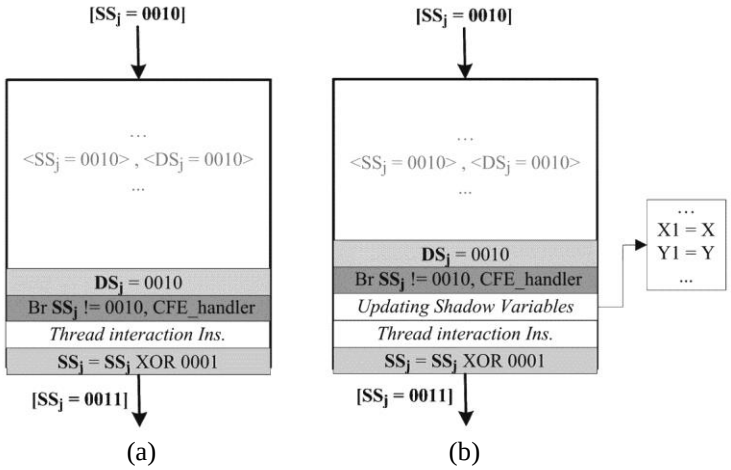



**Fig. 3** Illustration of added instructions for methods: (a):CRDC (b):CRMC.

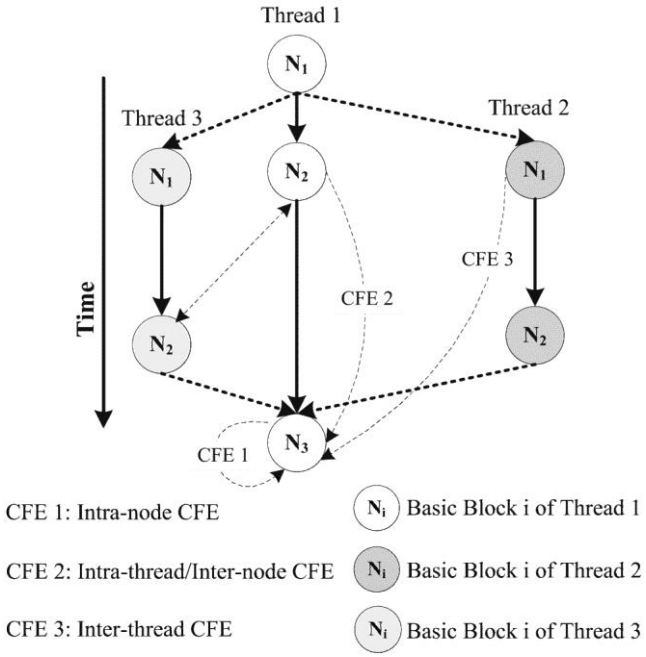



**Fig. 4** Illustration of CFE types in CFG scheme.

### 3.1. Intra-thread/Inter-node CFE detection

After determining control dependencies among basic blocks of the program, each node of the dependency graph should be labeled by a unique signature. The sequence of these signatures is checked at run-time by the instructions added at the end of each basic block. The checking instructions compare the value of the run-time signature with the pre-defined value assigned to each block at compile time. The run-time signatures should be updated, after checking instructions confirm the correct execution. Fig. 3 shows the added instruction to the basic blocks due to methods implementation. If an illegal jump occurs before added instructions at the end of the basic block and control is transferred to it illegally, then the CFE can be detected by comparing the stored value in the $SS_j$ (as the signature of the node) with another one calculated in compile time. If they are not equal, the CFE is detected and the function used for recovery is called. Source Signature of thread *j* ($SS_j$) is a shared variable of thread *j* which is continuously updated in executed nodes (where *j* shows thread number of multithreaded program). *SSj* finally stores the signature of the basic block in which a CFE has occurred. Destination Signature of thread *j* ($DS_j$) is a shared variable of thread *j* which is continuously updated, and finally stores the signature of the basic block that control is transferred to it incorrectly. Shadow variables update instructions are placed in some basic blocks based on an algorithm that has explained in the proposed CRMC section. Additionally, interaction instructions like *pthread_create*/*pthread_join* exist in some basic blocks based on the type of program and they direct the flow of program to other thread legally. Thereupon if an illegal branch jumped to the block including interaction instruction, it cannot be detected before thread interaction. So these instructions are placed after $DS_j$ update and checking instructions to prevent thread interaction before CFE detection. Both source and destination signatures are used in *CFE_handler* function of both proposed techniques to recover CFE and data errors.

### 3.2. Inter-thread CFE detection

Each thread of multithreaded program has particular signature identifier to avoid possible interference by the threads in updating and checking phase. Thereupon, signature of thread *j* is allowed to be updated only in thread *j* and each illegal signature updating in thread *j* considered as CFE. As illustrated in Fig. 5, an inter-thread CFE occurred from $N_2$ of thread *1* to $N_2$ of thread 2 before the signature of thread *2* updated at the end of $N_1$. This CFE can be detected by comparing last updated $SS_{destination\ thread}$ with expected value at the end of $N_2$ in thread 2.

## 4. AUTOMATIC RECOVERY PHASE

In the previous section, some problems of prior methods used for recovery are described. Moreover, as showed in critical applications the recovery methods which only concentrate on the CFEs, is not applicable. So, the data errors should be considered and finally recovered. The techniques for recovering the data errors by duplicating instructions are presented in [2], [10], [11], [12]. However, this type of data errors recovery has high overhead because of duplicating and comparing. In the rest of this section, the proposed recovery techniques are explained.

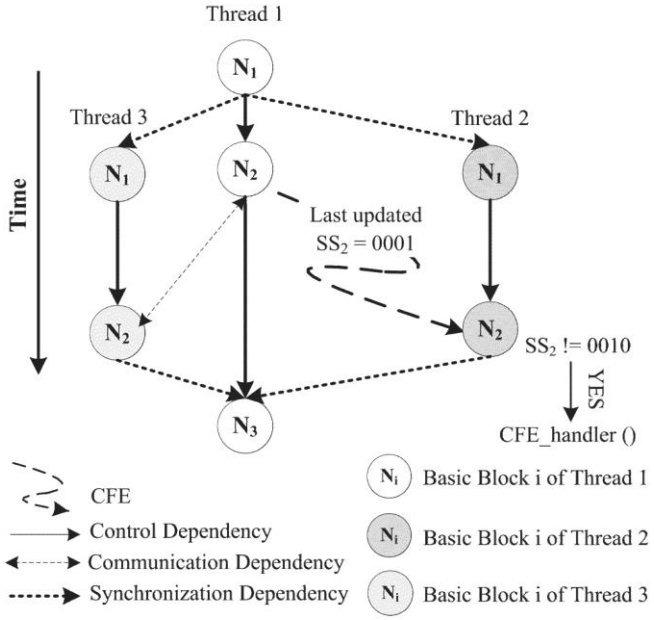



**Fig. 5** Inter-thread CFE detection.

### 4.1 The proposed CRDC technique

When a CFE is detected through added instructions, the control is transferred to *CFE_handler* function. This function is implemented by considering the DFG and CFG of the program at design time. The signatures of the source and destination basic blocks are given to *CFE_handler* function as inputs. This function can relocate the control to the nearest block from which re-executing the program corrects the CFE, and all of the affected variables between source and destination will be re-initialized.

Fig. 6 (a) shows three basic blocks from the set of basic blocks of a thread in a program code as well as the DFG extracted from data dependencies among variables in these basic blocks. Fig. 6 (b) illustrates the process of the correction used by the proposed techniques. Regarding them, if *CFE1* has occurred in *basic block 2* and the control is transferred from *basic block2* to *basic block3* (step 1 in Fig. 6(b)), then the values stored in variables *X* and *Z* cannot be reliable, because of the problems previously explained. For example, suppose that the source basic block is *basic block2* and the destination one is *basic block3*, also the variables modified by the CFE (*X* and *Z*) are initialized in *basic block1* and *basic block2*. For CFE and data errors recovery, the control should be transferred to *basic block1* (step 3 in Fig. 6(b)). Therefore, the modified variables are re-initialized and their corresponding computations are re-executed after this transmission. By re-executing the code from *basic block1*, the first value which was stored in variable *Z* is re-loaded again. Also, after completing *basic block1* and in *basic block2*, the first value of *X* is re-loaded.

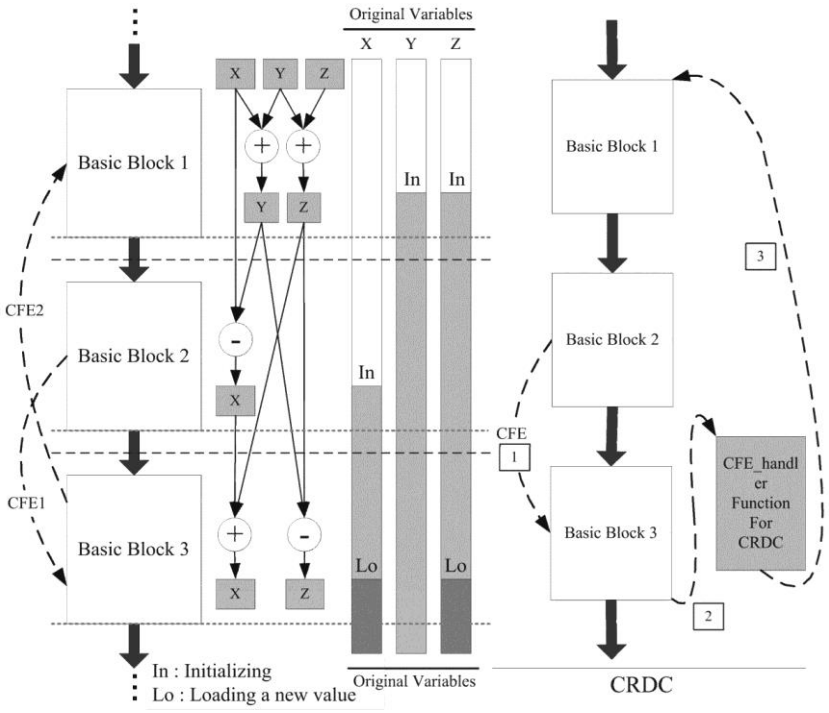



**Fig. 6** (a): CFG and DFG generated from program code, (b): Scheme of CRDC methods

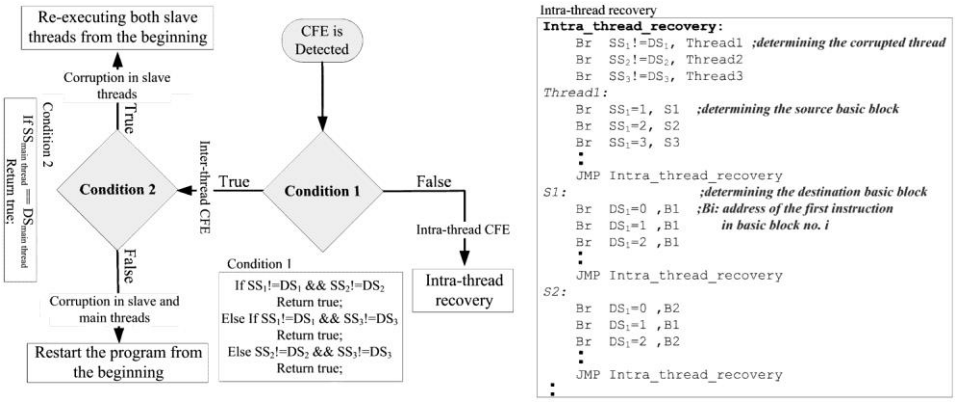
Another example is when CFE2 occurs, then the source basic block is *basic block3* and the destination one is *basic block1*. The variables affected by this CFE are *X*, *Y*, and *Z*. The initialization of *X* is done in *basic block2*, and the initialization of variables *Y* and *Z* are done in *basic block1*. Hence, returning to *basic block1* leads to load the initialization values to variables and re-execute computations by which the variables had been used.

In multithreaded programs, since threads act on each other, recovering one thread in the case of CFE does not mean the whole of program is recovered. In many cases, several threads should rollback to special locations to provide consistency and true execution in re-execution process. Threads which were created in our benchmarks were entirely independent function and there was no need to rollback several threads to a previous state except in the case when inter-thread CFE would happen. In this case, corrupted threads are discovered and rollback process is done based on relations among slave threads and main thread.

Furthermore, in this technique for detection and correction of illegal jumps to unused space (partition block), the partition block is filled-up with branch instructions to *CFE_handler* function. Zero (Null) is reserved as the destination signature value for the partition block to distinguish it from the other blocks in the program code. If the illegal jump occurs to it, The *CFE_handler* function ignores the destination, because it contains no computation related to the program.

*4.1.1. The proposed CRDC CFE_handler function*

Fig. 7 (a) shows scheme of the *CFE_handler* function defined for a program including three threads in CRDC technique. Determining the type of CFE (intra/inter thread CFE) is the first step after transferring control of program to the *CFE_handler* function. As shown in condition 1 code, if the $SS_j$ and $DS_j$ in two different threads were not equal, the occurred CFE is inter-thread type. In this case, another situation should be considered that is whether the slave threads have only been corrupted or the main thread has also been corrupted. In spite of re-starting the program in the situation where the main thread has corrupted, the program can resume from the thread creation instructions in main thread when only slave threads have corrupted. As shown in Fig. 7 (b), to recover intra-thread CFEs, the CRDC function first determines the corrupted thread by comparing $SS_j$ and $DS_j$ of each thread. Next, it specify source basic block by comparing the value stored in $SS_j$ with the signatures assigned to each basic block at design time. Then, the control is transferred to sub-sections which are separately defined for each source basic block. In these subsections the destination basic block can be determined as similar as determining the source one, and finally the control can be transferred to the basic block in which the first initialization of the affected register is done. This transition can be performed by conditional branches to the first instruction of the basic blocks. When an illegal jump occurs to the *CFE_handler* function statements, the function can gives the control back to the source basic block, by executing the first subsection. The last lines of the subsections (jump instructions to the first line of the function) were defined to correct this type of CFEs.




(a) (b)

**Fig. 7** CRDC Scheme ((a): Sample flowchart, (b): CFE-Handler function).

*4.1.2. Optimization of the CRDC CFE_handler function*

In some applications, imposing less memory overhead may be more important than the other issues, due to the area-efficiency of their designs. Therefore, the structures of this function should be optimized under the memory constraint. As shown by Fig. 8, the targets of the final branches, used in the last phase of the recovery, are specified taking to account the pair of the source and destination basic blocks. So, to reduce the number of instructions in the functions, the phase of determining the destination basic block can be omitted. For

example, regarding Fig. 8, if the source basic block of an illegal jump is *basic block 1*, then, the target of the final branches is also *basic block 1*, independent of the destination basic block. In the other case (for example the fourth line of the matrix in Fig. 8), considering the topmost basic block, nearest basic block to the beginning of the program, from the set of the targets for one specific source basic block causes that one of the phases in the recovery process (checking the value of the $DS_j$ for determining the destination) is omitted. These optimizations in designing the structures of the functions lead to effective improvements in the percentages of the memory overhead of the proposed technique.

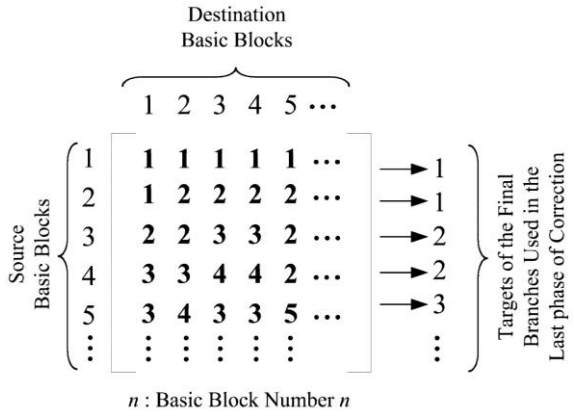



**Fig. 8** Schematic of an algorithm for reducing the memory overhead of the CFE-handler function.

### 4.2. The proposed CRMC technique

In the first section, some problems (potential of imposing high overhead and high latency) of checkpoint-based methods were explained. In CRMC technique, the program gets checkpoints at some points during code execution. At these times, the values stored in variables (such as registers and memory blocks) should be sustained in shadow variables. Then, for CFEs and data errors recovery, it needs to re-execute from the last trustable checkpoint, after loading correct values stored in shadow variables to original ones. Through the CRMC, the shadow variables always contain the true values of the original ones. If shadow variables updated at the end of each basic block in which the corresponding original variables has been modified, a noticeable performance and memory overheads are imposed to the system. On the other hand, since thread interaction instructions such as synchronization or communication change some variables in different threads, these modifies should be considered in the proposed recovery technique. Therefore, the shadow variables are divided to two different shadows:

- *Local shadows*: Local shadows are used to accelerate recovery process while the source and destination basic blocks of CFE are from two different threads. The contents of the local shadow are chosen by the application programmer with respect to the information provided by the DFG of the program and there is no need to save the entire system variables or any other information related to the hardware or the operating system. To reduce the imposed overheads due to

shadow variables, we specify a boundary of consecutive executed basic blocks which are free of thread interaction instructions as macro basic block for each thread. The local shadow variables are updated at the end of each macro basic block; then a snapshot of the thread state is taken. If the macro basic block placed within a loop, a variable is used to specify the iteration number and trigger shadow variables updating at acceptable iterations. Regards to the points which have mentioned above, the pace of executing thread interaction instructions can be introduced as macro basic block size:

$$\text{Macro basic block size} = \frac{\#\ of\ basic\ blocks\ of\ program}{\#\ of\ BBs\ including\ interaction\ Ins. \times \#\ of\ threads} \quad (1)$$

Assume a program consists of 24 basic blocks that 3 of them included thread interaction instructions. The macro basic block size will be approximately equal to 3 based on the equation 1. Fig. 9 shows the scheme of this macro basic block. The shadow variables are updated at the end of third basic block as illustrated in Fig. 9. This optimization directly leads to noticeable reduction in the overheads of our method in compare to checkpoint-based methods.

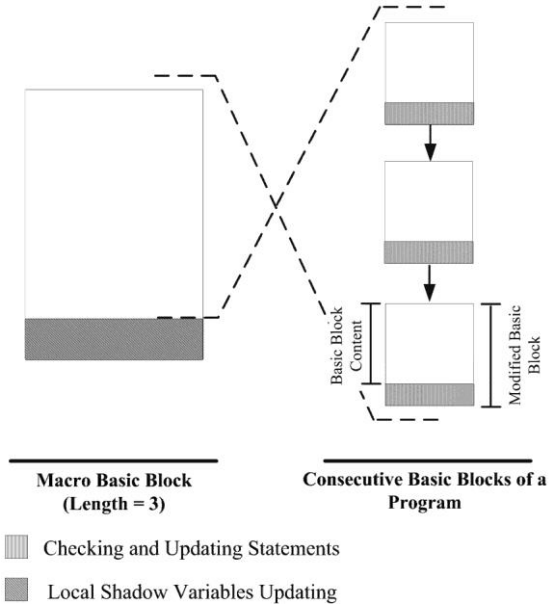
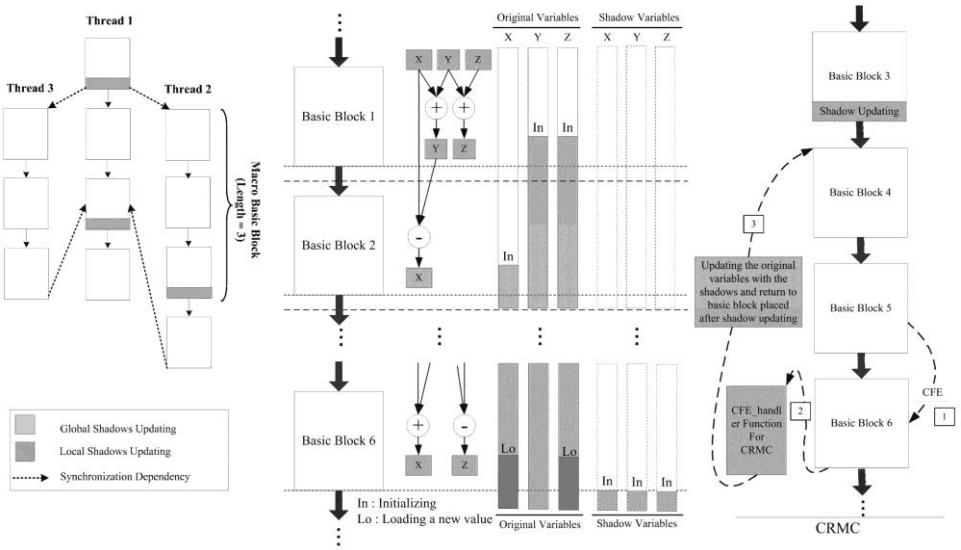



**Fig. 9** Illustration of the shadow variables updating location.

- *Global shadows:* The places where the global shadows are updated should correspond to a consistent state of the application. We considered synchronization/communication points of the application like at the beginning and end of create/join and lock/unlock relations as natural consistent global states. A miniaturized snapshot of entire system saved at global shadows and it will be used when the global consistency needed.

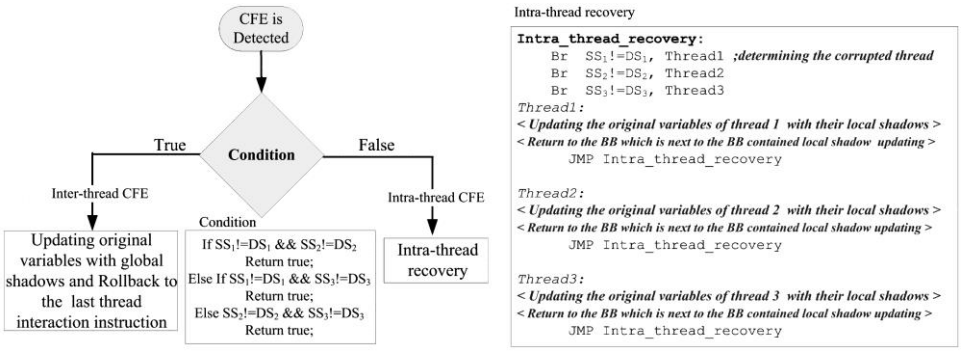
*4.2.1. The proposed CRMC CFE_handler function*

Regarding Fig. 10 (a), the time of updating the local and global shadows with the original ones is illustrated. As shown in Fig. 10 (b), suppose that variables *Y* and *Z* are initialized in *basic block1*, and variable *X* is initialized in *basic block2*. For example, the values of variables *Y*, *Z* and *X* are changed in the different basic blocks of macro basic block. Therefore, the local shadow of all modified variables should be updated only at the end of macro basic block (instead of updating at the end of the all basic blocks). After detection phase, the control is transferred to *CFE_handler* function (step 2 in Fig. 10 (c)). At this time, the signatures of the source and destination basic blocks are already available in $SS_j$ and $DS_j$, respectively. These two values are given to *CFE_handler* function as inputs. As shown in condition code in Fig. 11 (a), if the $SS_j$ and $DS_j$ in two different threads were not equal, the occurred CFE is inter-thread type. In this case, the program should be updated with global shadows and resumed from that point. Otherwise, the occurred CFE is intra-thread type and the function can update the affected original variables in the source and the destination with shadow ones as demonstrated in Fig. 11 (b). Finally, the control is transferred to the address of basic block which is placed next to the basic block contained local shadow variables updating (step 3 in Fig. 10 (c)) and the code is re-executed from this point. Consequently, both of the CFE and the generated data errors can be corrected.




(a) (b) (c)

**Fig. 10** CFG and DFG generated from program code ((a): local and global shadows places, (b): local shadow updating, (c): scheme of CRMC)




(a) (b)

**Fig. 11** CRMC scheme ((a): sample flowchart, (b): CFE-handler function)

## 5. Experimental Results

In order to evaluate the proposed technique, five multithreaded benchmarks Quick Sort, Matrix Multiplication, Bubble Sort, Linked List and Fast Fourier Transform 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. Branch deletion, branch insertion and branch target modification used as considered fault models. Table 1 represents a comparison of associated overheads and error recovery coverage in different methods. The memory and performance overheads of the proposed techniques are lower than other previous works ([10], [4]). The memory/performance overhead of the ACCED is comparatively higher than the proposed techniques because of adding duplicated instructions and executing the set of instructions used for comparing the results to obtain correct output. Moreover, the memory and performance overheads of the proposed techniques are slightly increased, when the running threads of the programs increase. This is due to the utilizing different checkpoint level and concept of macro block in CRMC and using less checking instruction at the CFE detection phase in CRDC.

**Table 1** Comparison of the memory and performance overheads and error recovery coverage

| Bench-marks Category | Bench-marks | ACCED[10] | | | CDCC[4] | | | MCP[4] | | | CRDC | | | CRMC | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *M.O.*[a] *(%)* | *P.O.*[b] *(%)* | *E.C.*[c] *(%)* | *M.O.*[a] *(%)* | *P.O.*[b] *(%)* | *E.C.*[c] *(%)* | *M.O.*[a] *(%)* | *P.O.*[b] *(%)* | *E.C.*[c] *(%)* | *M.O.*[a] *(%)* | *P.O.*[b] *(%)* | *E.C.*[c] *(%)* | *M.B.S*[d] *(%)* | *M.O.*[a] *(%)* | *P.O.*[b] *(%)* | *E.C.*[c] *(%)* |
| Dual-Threaded Programs | QS | 222.6 | 112.3 | 86.5 | 86.5 | 71.4 | 84.3 | 178.3 | 92.4 | 81.1 | 72.6 | 51.2 | 94.0 | 4 | 84.5 | 62.3 | 93.2 |
| | MM | 219.2 | 101.0 | 88.3 | 75.2 | 57.7 | 89.8 | 144.5 | 70.2 | 85.8 | 55.9 | 48.0 | 94.3 | 5 | 67.8 | 59.1 | 94.0 |
| | BS | 226.5 | 108.4 | 84.3 | 88.6 | 70.0 | 84.0 | 182.5 | 88.4 | 83.4 | 74.3 | 50.2 | 91.9 | 4 | 86.4 | 61.3 | 91.1 |
| | LL | 228.0 | 104.4 | 81.9 | 91.4 | 69.1 | 83.5 | 184.6 | 83.1 | 80.9 | 75.5 | 49.9 | 92.2 | 4 | 87.6 | 60.7 | 92.8 |
| | FF | 195.3 | 97.8 | 88.0 | 71.3 | 55.2 | 88.5 | 135.7 | 68.0 | 87.9 | 54.3 | 44.3 | 92.9 | 4 | 66.1 | 55.7 | 92.0 |
| | Avg. | 218.3 | 104.7 | 85.8 | 82.6 | 64.6 | 86.0 | 165.1 | 80.4 | 83.8 | 66.5 | 48.7 | 93.0 | 4 | 78.4 | 59.8 | 92.6 |
| Quad-Threaded Programs | QS | 232.0 | 119.2 | 85.3 | 104.1 | 88.6 | 83.1 | 189.0 | 108.6 | 80.8 | 88.3 | 60.6 | 93.1 | 3 | 99.1 | 71.9 | 92.6 |
| | MM | 231.7 | 117.5 | 87.6 | 91.5 | 63.3 | 88.3 | 162.0 | 86.8 | 83.3 | 67.2 | 51.5 | 93.2 | 4 | 78.7 | 62.5 | 92.6 |
| | BS | 238.1 | 115.9 | 82.0 | 107.8 | 79.7 | 83.4 | 193.6 | 102.3 | 81.0 | 85.1 | 59.1 | 90.1 | 3 | 96.4 | 70.7 | 89.3 |
| | LL | 242.6 | 113.8 | 80.1 | 110.7 | 76.5 | 81.7 | 196.2 | 96.4 | 79.0 | 83.7 | 58.7 | 91.3 | 3 | 94.2 | 69.1 | 91.9 |
| | FF | 217.5 | 102.5 | 86.7 | 89.0 | 61.4 | 86.3 | 157.9 | 82.7 | 84.2 | 64.6 | 47.2 | 92.1 | 3 | 75.4 | 58.8 | 90.0 |
| | Avg. | 232.3 | 113.7 | 84.3 | 100.6 | 73.9 | 84.5 | 179.7 | 95.3 | 81.6 | 77.7 | 55.4 | 91.9 | 3 | 88.7 | 66.6 | 91.2 |

a. Memory overhead b. Performance overhead c. Error recovery coverage d. Macro block size

## 6. CONCLUSIONS

In this paper, two software techniques to detect and correct CFEs in multithreaded programs are proposed. These techniques are implemented via considering control and data dependency in dependency graph beside synchronization and communication dependency at compile time. Also, proposed techniques correct data errors generated by CFEs that can cause considerable corruptions in the systems. Fault injection experiments showed that the proposed techniques, when applied on the programs, produce correct results in over 91.2% of the cases. The latency and the additional memory required 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] 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.

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

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

[5] R. Venkatasubramanian, J. P. Hayes and B. T. Murray, "Low-cost on-line Fault Detection Using Control Flow Assertions" In Proceedings of the 9th IEEE International On-Line Testing Symposium, 2003, pp. 137-143.

[6] R. Vemu and J. A. Abraham, "CEDA: Control-flow Error Detection through Assertions" In Proceedings of the 12th IEEE International On-Line Testing Symposium, July 2006, pp. 151-158.

[7] 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.

[8] Y. Sedaghat, S. G. Miremadi and M. Fazeli, "A Software-Based Error Detection Technique Using Encoded Signature", In Proceedings of the 21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2006, pp. 389-400.

[9] 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", In Proceedings of the 35th International Conference on Dependable Systems and Networks, 2005, pp. 50-58.

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

[11] D. Gizopoulos, M. Psarakis, S. V. Adve, P. Ramachandran, S. K. Hari, D. Sorin, A. Meixner, A. Biswas and X. Vera, "Architectures for Online Error Detection and Recovery in Multicore Processors" In *Proceedings of* Design, Automation and Test in Europe, 2011.

[12] 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", In Proceedings of the 22nd International Symposium on Fault Tolerant Computing, 1992, pp. 316-325.

[13] C. Bolchini, A. Miele, M. Rebaudengo, F. Salice, D. Sciuto, L. Sterpone and M. Violante, "Software and Hardware Techniques for SEU Detection in IP Processors", *Journal of Electronic Testing Theory and Application,* vol. 24, no. 1-3, pp. 35-44, 2008.
[14] R. Vemu and J. A. Abraham, "Budget-dependent Control-flow Error detection", In Proceedings of the 14th IEEE International On-Line Testing Symposium, 2008, pp. 73-78.
[15] Gnu debugger. http://www.gnu.org/software/gdb/.