Two Efficient Software Techniques to Detect and Correct Control-flow Errors

Hamid R. Zarandi, Mohammad Maghsoudloo, Navid Khoshavi
Department of Computer Engineering and Information Technology
Amirkabir University of Technology
Tehran, Iran
E-mails: {h_zarandi, m.maghsoudloo, navid.khoshavi}@aut.ac.ir

Abstract—This paper proposes two efficient software techniques, Control-flow and Data Errors Correction using Data-flow Graph Consideration (CDCC) and Miniaturized Check-Pointing (MCP), to detect and correct control-flow errors. These techniques have been implemented based on addition of redundant codes in a given program. The creativity applied in the methods for online detection and correction of the control-flow errors is using data-flow graph alongside of using control-flow graph. These techniques can detect most of the control-flow errors in the program firstly, and next can correct them, automatically. Therefore, both errors in the control-flow and program data which is caused by control-flow errors can be corrected, efficiently. In order to evaluate the proposed techniques, a post compiler is used, so that the techniques can be applied to every 80X86 binaries, transparently. Three benchmarks quick sort, matrix multiplication and linked list are used, and a total of 5000 transient faults are injected on several executable points in each program. The experimental results demonstrate that at least 93% and 89% of the control-flow errors can be detected and corrected without any data error generation by the CDCC and MCP, respectively. Moreover, the strength of these techniques is significant reduction in the performance and memory overheads in compare to traditional methods, for as much as remarkable correction abilities.

Keywords: Control-flow error, Data error, Control-flow checking, Error detection, Error correction

I. INTRODUCTION

Following the trends of decreasing feature sizes, low supply voltage and high frequency, future microprocessors will become more susceptible to transient errors (also called soft errors, or single-event upsets) 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 [1]. Moreover, it has been shown that between 33% and 77% of transient hardware faults occurred in all of the microprocessor components lead to control-flow errors [2].

A Control-Flow Errors (CFE) can be caused by Transient or permanent faults in hardware components such as program counter (PC), address circuits, steering and control logic, or the memory system which affect the correct and predictable execution of the processors as the sequence of instructions [3]. CFEs can be divided into two types: inter nodes and intra nodes. An intra node CFE is an illegal movement within a node (basic block), and an inter node CFE is an illegal movement between two nodes, or an illegal movement from a node to an unused spaces of memory which is called partition block.

Control-Flow Checking (CFC) is a key method that is used for monitoring the flow of a program which partitions the program code into basic blocks (that are branch free groups of instructions terminated by a branch), and then adds redundant hardware/software components for checking the correct execution flow of the program. Several methods have been designed for detecting each type of the CFEs in a program code. From implementation point of view, methods are divided into software-based and hardware-based techniques. Hardware-based methods use an external hardware like watchdog (checker) processor to monitor state and performance of the main (master) processor [4]. On the other hand, software-based techniques make use of redundant software instead of hardware and are based on signature assignment of each basic block. Signatures are calculated at runtime and next compared with the original ones which were calculated at compile time [4].

In the systems where hardware cost is another major concern that can be considered beside safety and reliability (such as real-time and embedded systems), using hardware-based redundancy is not recommended, and software-based redundancy is more convenient to be utilized.

Most of the software-based techniques are handled only for CFEs detection through signature assignments to basic blocks [4], [5], [3], [6], [7]. CFEs detection has been widely studied in literature, however, a few published works considered CFEs correction [2], [8]. After the CFE is detected, control should be transferred back to the block, where illegal branch has occurred in. However, correcting the CFE is not sufficient and it cannot be sure that the program will not fail since there may be some data errors generated by the CFEs. Therefore, any data errors caused by CFE should be repaired before correcting the CFE.

In order to correct data errors, programs need to gets some checkpoints during code execution, and then restore and re-execute from the last checkpoint location in the program. However, this may not be possible in systems, because getting checkpoint, restoring program states and re-executing impose significant area cost and latency [2].

In this paper, two techniques for automatic CFEs detection and correction are presented: 1) a technique which is called Miniaturized Check-Pointing (MCP) that takes less
performance overhead than previous work, and 2) a technique which is called Control-flow and Data Errors Correction using Data-flow Graph Consideration (CDCC) that takes less memory overhead than previous work. In order to detect CFEs, a signature is used for each node, and then some instructions are specified and inserted at the beginning and at the end of the nodes for calculating and checking signatures at runtime. For automatic correction of CFEs an error-handler function is prepared at design time to do that. This function is implemented through considering both Data-Flow Graph (DFG) and Control-Flow Graph (CFG) of the program.

To evaluate the techniques, three well-known benchmarks which are utilized and emulated in 80X86-based EMU emulator [9] are used. The results of injecting about 5000 transient faults on the program code reveal that 93.5% and 89.6% of CFEs are detected and corrected by CDCC and MCP methods without any data errors, respectively. The experimental results show that performance and memory overheads of CDCC and MCP are considerably less than previous related work like ACCED [2] which is based on duplication methods to correct the data errors.

The structure of this paper is as follows: section 2 gives related work and introduces terminologies. Section 3 gives a brief description about some definitions. Section 4 describes the detection technique which is used in proposed techniques. Section 5 explains the automatic correction methods. Section 6 describes the tool and presents the experimental results, and also contains a discussion about results. Finally, Section 7 concludes the paper.

II. RELATED WORK

Various works related to detect and correct CFEs have been presented yet. This section only concentrates on software-based methods due to the similarities in the base of idea that exist between proposed methods and previous works.

The Software-based error detection Technique using Encoded Signatures (SWTES) was proposed in [7]. First of all, the SWTES technique assumes that the program is partitioned into the blocks that include basic blocks and partition blocks. Also, seven types of CFEs which can happen between these blocks during execution of the program are defined and considered. This technique, like as most of the software-based methods, is based on assigning signatures to each of the basic blocks and checking signatures at the end of them. However, two major parts of this technique (the labeling algorithm and the signature generating step) should be explained to clarify concept of this method. In the labeling algorithm, a unique label is assigned to each of the basic blocks through using two lists of signatures (valid and black lists). These lists are defined and utilized to perform signature assignment efficiently and precisely. The generated signature for each node has four fields. The most significant field called Status field contains the status of the label of the node which is related to its successor node labels. The next field is the Entry/Exit bits. These bits indicate the last status of entry and exit flags. The third field called ID specifies the label of the current block, and the last variable field of signature, named Successors field which determines possible successors of current block uniquely. Through these algorithms for labeling and signature assigning, and detection routine which are applied in the method, it can detect all of the seven presumed types of CFEs.

The Block Signature Self Checking (BSSC) technique presented in [4] is purposed to check the flow of a program between branch free blocks (basic blocks) by defining a signature for each basic block and check it at the end of the blocks. In this technique, at the beginning of the blocks, a subroutine is called and address of the first instruction in the node is pushed into top of the stack as the signature, or it is stored in a static variable. A subroutine at the end of the basic block compares the embedded signature with the signature stored by the entry routine. Therefore, it can detect an illegal jump between two nodes caused by a CFE.

In Error Capturing Instructions (ECI) technique [4], errors which cause erroneous execution in the unused spaces and data spaces are detected. This technique fills unused space (partition block) with branch instructions to an infinite loop or with software interrupt instructions. So, with execution of these instructions, occurring a CFE can be indicated.

These two techniques i.e., BSSC and ECI, can be combined in order to increase the error detection coverage, because the BSSC technique can only detect the CFEs in the program space and the ECI can only detect the CFEs in the unused and data spaces.

The Control-flow Error Detection through Assertions (CEDA) technique is presented in [3]. In this technique like previous techniques, extra instructions are automatically embedded into the program at compile time in order to update run-time signatures continuously, and to compare them with pre-assigned value. Previous techniques insert multiple instructions in the basic blocks whenever they want to update the run-time signature. However, this technique inserts fewer instructions than the previous works, because of calculating signatures differently. Therefore, this technique has less overhead and is more efficient than the prior works. Decreasing overheads and increasing effectiveness are main purposes of this approach.

The Automatic Correction of Control-flow Errors (ACCE) technique [2] partitions the program code into functions which include one or more basic blocks. ACCE uses CEDA technique for CFEs detection. After the detection phase, a predefined function called error-handler is automatically executed, and the program control is transferred to the function and then to the basic block in which the illegal jump has been occurred. An extension of ACCE, called Automatic Correction of Control-flow Errors with Duplication (ACCED), is used for data error detection and correction through duplicating instructions. Although this method can detect and correct data errors as well as the technique mentioned in [2], not all of data errors can be detected. Moreover, the area overhead of methods is more than 100 percent which is significant.
III. PRIMITIVE DEFINITIONS

This section introduces some basic definitions which are necessary for describing the proposed methods.

In order to develop the idea, two flow graphs have been used. One of them is known as a Control-Flow Graph (CFG) which is a graph with set of nodes (basic blocks) and directed edges that show the right transmission between basic blocks, and another one is Data-Flow Graph (DFG). DFG is a graph with set of nodes (operations and variables) and directed edges which present data dependencies between variables.

Furthermore, two types of errors (which can occur during execution of the programs) are Control-Flow Error (CFE) and data error. A CFE happens if the sequence of instructions which are executed in the presence of a fault is different from the fault free sequence. CFEs can also be one of the causes of the data errors generation. Fig. 1 illustrates how the data errors can be generated due to a CFE. When a CFE occurs (like Branch1 in Fig. 1), it can be detected at the beginning of the basic block or at the end of it by added instructions, then the function called CFE-handler can transfer control to the block in which the CFE has occurred (Branch3). This type of correction has been used in the previous published technique [2] which can generate data errors. With regards to Fig. 1, the data errors can happen because of executing the set of instructions resided in Region 1 (which are not executed at regular time), or the set of instructions resided in Region 2 (that are performed wrongly) or the set of instructions resided in Region 3 (which are executed additionally). So, by returning exactly to the block in which the CFE has occurred, data errors are probable to happen in the output results. Data errors in so many applications such as safety critical applications may cause destructive results in output. Therefore, in these situations, only focusing on correction of CFEs is not sufficient and applicable.

In correction phases of the methods, signatures of the source and the destination basic blocks are required.

Therefore, these signatures are maintained within two registers (Source Signature Register and Destination Signature Register). Source Signature Register (SSR) is a run-time register which is continuously updated, and finally stores the signature of the basic block in which a CFE has occurred, and Destination Signature Register (DSR) is a run-time register which is continuously updated, and finally stores the signature of the basic block that control is transferred to it incorrectly due to a CFE. Also, in correction phase of the MCP technique, the variables which are known as shadow variables have been used for storing copies of the original variables. These copies are continuously updated at the end of the basic blocks if the original variables have been changed in.

IV. CFES DETECTION MECHANISM

Fig. 2 demonstrates that how the CFE can be detected by the added instructions to the basic blocks, and Fig. 3 shows the added instructions to each basic block because of the methods, separately. The CFE detection methods used in the MCP and the CDCC 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 correction.

If an illegal jump occurs before added instructions at the beginning of the basic block (CFE1 in Fig. 2) and control is
transferred to it illegally, then the CFE can be detected by comparing the stored value in the SSR (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 correction is called. For CFE2 (in Fig. 2), which is an illegal jump to the middle of the basic block, added instructions at the end of the node can detect CFE2 similar to the case of the additional instructions at the beginning of the node.

The signature of the destination basic block (that control is transferred to it illegally) is stored in DSR at the beginning and at the end of the node. Pair of the source and the destination signatures will be used for correcting CFEs in the correction phase. When an illegal jump occurs to middle of the node, the stored value in DSR cannot be reliable, if the destination signature is stored only at the beginning. For example in Fig. 2 when CFE2 occurs, the signature of the previous basic block (source) was stored in DSR (this value can mislead the CFE-handler function) while the added instructions at the end of the basic block change this situation and store the true value in the DSR.

The all expressed points which should be considered in the implementations of the techniques are actualized by the added instructions illustrated in Fig3. In the MCP, at the end of the basic blocks, the shadow variables should be updated if the original ones of them have been changed in. Therefore, the shadow variables almost always store the true value of the original ones. Furthermore, in the correction phase, these values are trustable for correcting the generated data errors. In the next section, the manner of correction methods used in the techniques will be described in details.

V. AUTOMATIC CORRECTION PHASE

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

A. The Proposed MCP Technique

In the first section, some problems of checkpoint-based methods were explained. In those methods the program gets checkpoints at some times during code execution. At these times, the values stored in variables (such as registers and memory blocks) should be sustained in shadow variables, and for CFEs and data errors correction, it needs to re-execute from last checkpoint after loading the correct values stored in shadow variables to original ones. However, this method may not be possible in some systems, because of high overhead and high latency for getting checkpoint, restoring program states and re-executing.

In the MCP technique, only some of the shadow variables are updated at the end of the each basic block in which the corresponding original variables has been modified. Through the MCP, the shadow variables always contain the true values of the original ones which are reliable. Also, it can considerably optimize the overheads of

---

**Figure 4. CFG and DFG Generated from Program Code**

**Figure 5. Differences between MCP and CDCC Methods**
checkpoints-based methods. Information about DFG of the program is already available at design time and can be specified updating shadows segments with respect to it. Fig. 4 shows three basic blocks from the set of basic blocks in a program code as well as the DFG generated from data dependencies among variables in these basic blocks. Suppose that variables \( Y \) and \( Z \) are initialized in **basic block1**, and variable \( X \) is initialized in **basic block2**. With regards to Fig. 4, the time of updating the shadow variables with the original ones can be found. For example, the value of variable \( Y \) is only changed in **basic block1**. Therefore, the shadow of \( Y \) can be updated only at the end of this basic block instead of updating at the end of all basic blocks.

After a CFE has occurred, it can be detected by added instructions. As described in the previous sections, the CFE can change the stored values in variables used in computations in the source and the destination. Therefore, it can generate data errors in the results. After detection, control is transferred to CFE-handler function (step 2 in Fig. 5). When the CFE is detected, the signature of the source basic block is already available in SSR and the signature of the destination is already available in DSR. These two values are given to CFE-handler function as inputs. This function can update the affected original variables in the source and the destination with shadow ones, and also it can determine address of the first instruction in the source basic block. Finally control is transferred to this address (step 3 in Fig. 5) and the code is re-executed from this point. Consequently, both of the CFE and the generated data errors can be corrected. For improving the MCP, the temporal and local variables can be ignored, and the act of making shadows for them can be omitted. By this improvement, shadows are only made for global variables which are alternatively used in the program code.

### B. The Proposed CDCC Technique

When a CFE is detected through added instructions, control is transferred to CFE-handler function. This function can relocate the control to the basic block, from which re-executing the program corrects CFE and guarantees that the generated data errors in the program will be recovered. This approach is implemented by considering DFG of the program at design time. The signature of the source basic block (from which the control transferred incorrectly) and the signature of the destination one (to which the control transferred illegally) are given to CFE-handler function as inputs. Finally the function can transfer control to the nearest basic block wherein the modified variables between source and destination are initialized. The information needed to implement the function is already available at design time and the function can be implemented by considering them.

With regards to Fig. 4, if CFE1 has occurred in **basic block2** and control is transferred from **basic block2** to **basic block3** (step 1 in Fig. 4), then the values stored in variables \( X \) and \( Z \) cannot be reliable, because of the problems which were 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 correction, control should be transferred to **basic block1** (step 3 in Fig. 4). 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.

Another example is when CFE2 occurs, then the source basic block is **basic block3** and the destination one is **basic block1**. We can find them from stored signatures in SSR and DSR. 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** is done in order to load the initialization values to variables and re-execute computations in which the variables had been used.

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 used as the destination signature value for the partition block in order to distinguish it from the other blocks in the program code. If the illegal jump

---

**Err:**

**DST:** : determining the destination basic block
- 
Br: DSR=1, D1
- 
Br: DSR=2, D2
- 
JMP SRC

**D1:** : updating the original variables which can be changed in the destination basic block with the shadow ones
- 
< Updating the original variable which can be changed in basic block no. 1 with their shadows >
- 
JMP SRC

**D2:**
- 
< Updating the original variable which can be changed in basic block no. 2 with their shadows >
- 
JMP SRC
- 
JMP SRC

**SRC:** : determining the source basic block
- 
Br: SSR=1, S1
- 
Br: SSR=2, S2
- 
JMP SRC

**S1:** : updating the original variables which can be changed in the source basic block with the shadow ones
- 
< Updating the original variable which can be changed in basic block no. 1 with their shadows >
- 
JMP B1: B1: address of first instruction in basic block no. 1

**S2:**
- 
< Updating the original variable which can be changed in basic block no. 2 with their shadows >
- 
JMP B2
- 
JMP SRC

---

Figure 6. Implementation of CFE-Handler Function for MCP
occurs to it, the CFE-handler function ignores the destination, because it contains no computation related to the program. So, at design time, this issue should be considered when implementing the CFE-handler function is desired.

C. CFE-handler Functions

Fig. 6 and Fig. 7 show implementation samples of the CFE-handler function defined for the MCP and the CDCC, respectively. Both of these functions have been implemented by considering the CFG and DFG which are extracted at the design time.

Fig. 6 illustrates the CFE-handler function used in the MCP technique. This function determines the destination basic block at first. Then the affected original variables in the destination are updated with shadow ones. After this, the source basic block is determined and the affected original variables in the source are updated with the shadows; and finally, control is transferred to the source basic block. Consequently, the affected original variables store true values and the program re-executes from the source basic block in which the CFE has occurred. If an illegal jump occurs to the CFE-handler function statements, the function can give 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 in order to correct these CFEs.

VI. EXPERIMENTAL RESULTS

In order to evaluate the proposed technique, the EMU simulator [9] has been used for simulating the program on the 80X86 microprocessors. This simulator can generate console programs that can be executed on any computer that runs 80X86 machine code (Intel/AMD architecture), transparently.

Three well-known benchmarks have been utilized:
- Quick sort (QS): a quick sort program, which sorts a 100-element array of integers.
- Linked List (LL): a linked list program, which adds 10 elements to linked list.
- Matrix Multiplication (MM): a matrix program, which multiplies two 4*4 matrixes.

Simulation-based fault injection method has been used and about 5000 transient faults have been injected on several points in the basic blocks of the programs. The considered fault models were:
- 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.

In order to detect the generated CFEs, the results of fault free execution have been compared with faulty ones. Correctability shows the power of the technique for correcting the CFEs. It is said that a CFE has been corrected if the program exits normally with correct output. Table 1 shows the correctability of the programs with the proposed technique in compare to correctability of the programs with ACCED and Complete Check-Pointing (CCP). In order to compare the proposed methods with the methods implemented based on complete check-pointing techniques; a method (CCP) has been developed and applied to the benchmarks, with considering the specifications of them (which are described in the sections 1 and 5). It is found that about 7.2% of the injected faults return correct output without using any technique. As shown in Table 1, about 93.5%, and 89.6% of the injected faults return correct output.
with CDCC and MCP, respectively. The correctability of MCP is a little less than the others, because of high susceptibility of the segments in which the shadow variables are updated. On average, about 7% of the faults cause segmentation faults for the suggested techniques. Segmentation faults are generated if the illegal jumps occur to signature update statements.

Fig. 8 (a) illustrates the comparison among performance overhead percentages of the programs due to applying the methods. The performance overhead (cost) consists of the latency of the CFE-handler function and the effects of the CFE-handler function on the execution flow. The performance cost of the methods is calculated as follow:

Performance Cost = \( \Delta \text{Performance} / \text{InitialPerformance} \). (1)

Fig. 8 (b) 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 cost of the methods is calculated as below:

Memory Cost = \( \Delta \text{MemoryUsage} / \text{InitialMemoryUsage} \). (2)

The memory (performance) overhead of the ACCED is comparatively (about 100%) higher than the proposed techniques because of adding (executing) duplicated instructions and adding (executing) the set of instructions used for comparing the results to obtain correct output. The performance overhead of the CDCC is a little higher than the MCP, because of re-executing the set of instructions resided between the first initialization of the affected registers and the source basic block in which the CFE occurred.

The overall cost is differently calculated depended on the type of the applications on which the techniques are applied. Two impact factors have been defined for costs: \( \alpha \) and \( \beta \). The impact factor of the performance cost in the system is estimated by \( \alpha \), and the impact factor of the memory cost is estimated by \( \beta \). In each system, sum of these impact factors should be equal to one (\( \beta = 1 - \alpha \)). In some applications the performance cost is more important than the memory one. In these situations, the value of \( \alpha \) should be rather than \( \beta \). On the contrary, if the importance of the memory cost is higher than the performance one, the value of \( \beta \) should be rather than \( \alpha \). The cost is estimated as follow:

Cost = \( \alpha \times \text{Performance Cost} + \beta \times \text{Memory Cost} \). (3)

Without loss of generality, let \( \alpha = \beta = 0.5 \), and this means that the importance of the performance and memory cost are equal in the estimation.

In order to estimate the efficiency of the methods, a metric (which is called Correction Coverage per Cost or CCC) has been defined, as below:

\[ \text{CCC} = \text{Correction Coverage} / \text{Cost} \] (4)
where Correction coverage (correctability of the methods) is the percentage of the injected CFEs which are corrected by the techniques.

According to Fig. 8 (c), the CCC of the proposed techniques is more than twice the CCC of the ACCED, and it means that the proposed techniques are more efficient than the ACCED which was recently published.

With regards to Fig. 8, the 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 beginning and at the end of the basic blocks is a large number in compare to the others.

VII. CONCLUSION

In this paper, two efficient software techniques to detect and correct control-flow errors (CFEs) were proposed. Moreover, these techniques were implemented through considering both of control-flow graph and data-flow graph at design time. Also, it is clear that relinquishing the data errors generated by the CFE can cause considerable corruptions in the systems (especially in the safety critical applications). Therefore, the methods used for correcting the CFEs should consider and handle the generated data errors, impressively. Several methods were designed for detecting each type of the CFEs in a program code, previously. However, most of the previous methods could not be capable to deal with the generated data errors. In order to develop the proposed techniques, first a signature was assigned to each node for CFE detection, and then some instructions were specified for calculating and checking them at runtime. A function was defined in order to correct the CFEs and the data errors, automatically. This function called CFE-handler function was implemented through considering the control-flow graph and the data-flow graph of the program code at design time. Fault injection experiments showed that the proposed techniques i.e., CDCC and MCP, when applied on the programs, produce correct results in over 93% and 89% of the cases, respectively. The latency and the additional memory required for correcting the CFEs and the data errors are considerably less than the duplication based and check-pointing based methods which have been recently published. A metric for estimating efficiency of the techniques was defined, and it was shown that the proposed techniques are more efficient than the duplication-based and check-pointing-based methods.

REFERENCES