electronics

MDPI

Article

A Checkpointing Recovery Approach for Soft Errors Based on Detector Locations

Na Yang [1,*] and Yun Wang [2,*]

[1] School of Information Engineering, Tianjin University of Commerce, Tianjin 300134, China
[2] School of Computer Science and Engineering, Southeast University, Nanjing 211189, China
* Correspondence: yangna@tjcu.edu.cn (N.Y.); ywang_cse@seu.edu.cn (Y.W.)

Abstract: Soft errors are transient errors caused by single-event effects (SEEs) resulting from a strike by high-energy particles acting on sensitive areas of integrated circuits. Soft errors frequently occur in the space environment, adversely affecting the reliability of aerospace-based computing. A recovery process is launched to recover the program when soft errors are detected. A periodic checkpointing recovery approach is widely utilized to prevent soft errors. However, this approach does not consider the detector locations, resulting in a large time overhead. This paper proposes a checkpointing recovery approach for soft errors based on detector locations called DLCKPT. DLCKPT reduces the time overhead by considering detector locations. The experimental results show that the percentage decrease in the time overhead between the DLCKPT and the periodic checkpointing recovery approach is 13.4%. The average recovery rate and average space overhead are 99.3% and 44.4% for the periodic checkpointing recovery approach and 99.4% and 34.6% for the DLCKPT. These results show that the DLCKPT and the periodic checkpointing recovery approach produce comparable results for the recovery rate. The DLCKPT has a lower time overhead and a slightly lower space overhead than the periodic checkpointing recovery approach, demonstrating its effectiveness.

Keywords: soft error; single event effect; software reliability; fault tolerance; transient fault

Citation: Yang, N.; Wang, Y. A Checkpointing Recovery Approach for Soft Errors Based on Detector Locations. *Electronics* **2023**, *12*, 805. https://doi.org/10.3390/electronics12040805

Academic Editor: Manuel Mazzara

Received: 31 December 2022
Revised: 21 January 2023
Accepted: 3 February 2023
Published: 6 February 2023

1. Introduction

Soft errors are caused by energetic particles striking sensitive areas of integrated circuits, such as alpha particles and cosmic neutrons generated by packaging materials or radiation [1,2]. The first reports of failures attributed to cosmic rays emerged in 1975 when space-borne electronics malfunctioned [3,4]. Today's integrated circuits have higher integration and operating speeds. Therefore, they are more sensitive to soft errors [5]. Soft errors are transient faults. They do not permanently damage hardware, and the hardware operates normally after transient faults have occurred. However, soft errors influence data stored in registers and memory cells, impacting the accuracy of software computations.

Soft errors are a significant threat to software reliability. Therefore, soft error protection is essential [6]. Soft error protection includes soft error detection and recovery. The former detects soft errors, and the latter ensures recovery from soft errors. Soft error protection is conducted at both the hardware and software levels [7–9]. Hardware-based technology usually changes the processor architecture or attaches special-purpose hardware modules to the processor. It is expensive and typically not portable [10]. Due to the low-cost and applicability requirements, most soft error protection solutions have been developed and widely applied at the software level [11].

Soft error recovery ensures that a program operates correctly after a soft error has occurred. Software-based soft error recovery is conducted at the compiler level and the source code level. At the compiler level, instruction duplication and signature-based mechanisms are applied. The instruction duplication mechanism duplicates instructions and performs majority voting to recover from soft errors [12,13]. The signature-based

mechanism detects soft errors using a signature of a program block. The system recovers from soft errors by transferring control back to the program block that was executed before the soft errors occurred [14].

At the source code level, duplication-based and checkpoint-based mechanisms are applied. The duplication-based mechanism uses redundant variables to execute a recovery. The checkpoint-based mechanism performs recovery by checkpoint and recovery operations [15]. The checkpoint operation generates a checkpoint with data representing a copy of the current state of a process. The recovery operation is launched in the event of a failure. This operation recovers the process to a previously stored state by a checkpoint and continues from there. The soft error recovery mechanism at the source code level has been widely studied and implemented because it is straightforward and has a low cost. However, the following two challenges exist.

(1) The duplication-based recovery approach at the source code level generates redundant copies of variables. Soft errors are handled by utilizing redundant copies and original copies. However, it requires users to modify the source code of programs; thus, this is not a lightweight approach. Figure 1 provides an example; x is the original copy, and y is the AN-encoded copy of x, which is set to $3x$. AN-codes are a class of arithmetic codes where the codeword is the original data multiplied by a constant. When $3x$ is not equal to y, an error has occurred. If y is divisible by 3, then x is corrupted. In this case, y is used to correct x. Otherwise, y is corrupted, and x is used to correct y. If x is 2, y is set to 6. As shown in Figure 1, after the execution of line 1 and line 2, the values of x and y are stored in registers or memory cells. During the execution of line 3 and line 4, when a soft error occurs in x, the 3rd bit of x is changed from 0 to 1 so that the value of x is 10 (00001010) rather than 2 (00000010). In the following program execution, the error is detected in line 5 when the relationship is not satisfied, i.e., $y \neq 3x$. In this case, the value of y is used to correct x (lines 6–7). After the recovery, the value of x is corrected to 2. Figure 1 shows that the source code has been modified, and multiple statements have been inserted. The original three lines of codes increases to nine lines of codes.

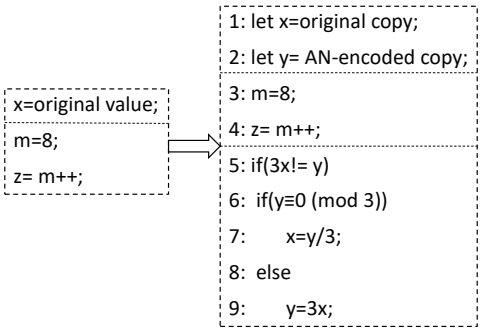

Figure 1. A pseudo-code of duplication-based recovery.

(2) Periodic checkpointing recovery is a popular checkpoint-based approach. It does not require users to modify the program's source code. A shell script is usually used to launch the checkpoint and recovery operations to save the current state of a process and recover it to a previously stored state. Figure 2 provides an example; *cr_run*, *cr_checkpoint*, and *cr_restart* are APIs provided by the checkpointing tool Berkeley Lab Checkpoint/Restart (BLCR) [15], *cr_run* starts the program execution, *cr_checkpoint* launches the checkpoint operation for generating a checkpoint file, and *cr_restart* launches the recovery operation to resume the process execution by the checkpoint file. The periodic checkpointing recovery approach does not modify the program codes. However, its time overhead cannot be substantially decreased because it does not consider detector locations when it deploys checkpoints. For example, in Figure 3, T is the program execution time, and the checkpoint interval is $T/4$. C_1, c_2, and c_3 are checkpoints that are set during T. D_1, d_2, $\cdots$, and d_5 are detectors that are executed during the program execution of $[3T/4, T]$. The first two detectors have ranges of $[t_p, t_q]$, and the last three detectors have ranges of $[t_m, t_n]$.

The rectangular box shows the code that is executed during $[3T/4, T]$. The five assertion statements represent five detectors, d_1, d_2, $\cdots$, and d_5. When d_3, d_4, and d_5 detect an error, the error recovery time at the recent checkpoint c_3 is very long because the program is rolled back to c_3 and continues running from c_3. The time of rolling back to c_3 is at least equal to the difference between t_m and $3T/4$.

```
1: cr_run ./p_name, p_data
2: get_pid
3: sleep checkpoint_interval
4: cr_checkpoint $pid
    .....
5: if an error is detected
6:    cr_restart ./checkpoint_file
7: fi
```

Figure 2. An example of a shell script describing a recovery.

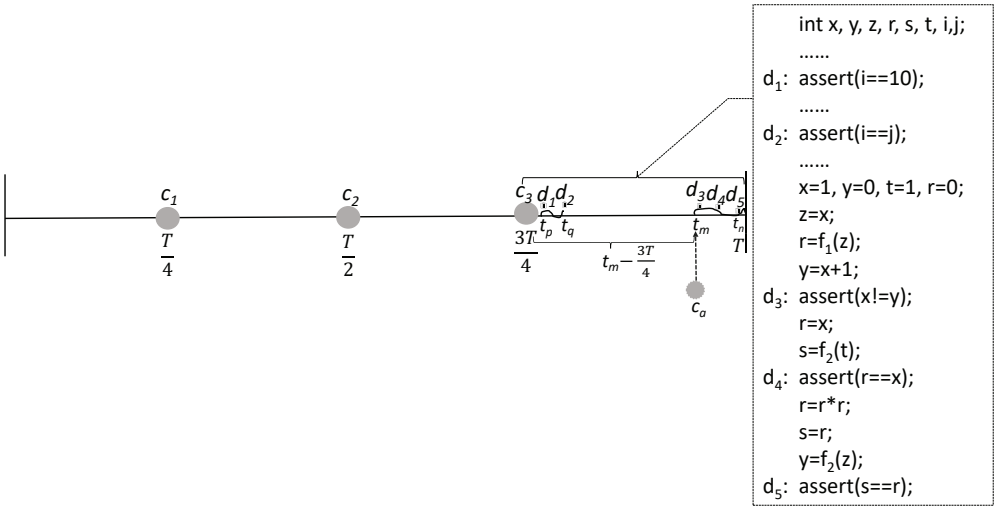

Figure 3. An example of checkpoints.

This paper proposes a checkpointing recovery approach for soft errors based on detector locations referred to as DLCKPT. This method executes soft error recovery with a low time overhead. DLCKPT does not require users to modify source codes, and its process is straightforward. Our main contributions are summarized as follows:

(1) A novel approach called DLCKPT is proposed for soft error recovery. DLCKPT recovers from soft errors by using checkpoint mechanism. DLCKPT deploys checkpoints by considering the location of detectors to reduce the time overhead. For example, as shown in Figure 3, when the error recovery time detected during $[t_m, t_n]$ and using c_3 is longer than that of using the checkpoint c_a, which is set before t_m, the additional checkpoint c_a is set before t_m to reduce the time.

(2) A program is divided into multiple segments using checkpoints. The time overhead of each program segment is determined, including the checkpointing time and recovery time. The change in the time overhead of each program segment is evaluated when checkpoints are inserted or deleted, and the change is used to adjust the checkpoint deployment.

(3) Experiments are conducted with various programs to evaluate the effectiveness of DLCKPT. The result shows that DLCKPT reduces the time overhead by considering detector locations and does not adversely affect the recovery rate and space overhead, demonstrating its effectiveness.

The remainder of this paper is organized as follows: Section 2 briefly reviews related work. Section 3 introduces the preliminaries. Section 4 presents the overview of the

proposed DLCKPT approach. The detailed process of DLCKPT is given in Section 5. Section 6 describes the experiment and results. Section 7 concludes the paper and discusses future work.

2. Related Work

Soft error recovery is an essential part of soft error protection. It recovers programs after soft errors have occurred. There is a significant body of work on software-based soft error recovery conducted at the compiler level and source code level.

The instruction duplication mechanism at the compiler level recovers programs from soft errors by applying redundant instructions. SWIFT-R [13] is a typical instruction duplication approach. It intertwines three copies of the instruction and uses majority voting before critical instructions, such as *store* instructions. Figure 4 presents an example. Before interacting with the memory cells, SWIFT-R determines the accuracy of the data using a 2-of-3 majority voting operation, which is represented by the "majority" function in Figure 4. In line 8, the $majority(r1, r1^{'}, r1^{''})$ means that the data in the three registers are set to the majority value. If $r1 = r1^{'}$ and $r1 \neq r1^{''}$, $r1^{''}$ is set to $r1$ to correct the data in $r1^{''}$. SWIFT-R performs soft error detection and recovery simultaneously on three copies. The cost of SWIFT-R is high because it adds many redundant instructions. S-SWIFT-R [16] is a selective SWIFT-R. It selects different register subsets from the microprocessor register file to be protected rather than protecting all registers. S-SWIFT-R is more flexible than SWIFT-R. However, there are still many redundant instructions in S-SWIFT-R. SWIFT-R and S-SWIFT-R only check the operands before executing critical instructions. They do not protect the result of the critical instructions. NEMESIS [17] was developed to improve soft error resilience and reduce the performance overhead. It compares the results rather than the operands of critical instructions. Although many efforts have been made to refine the duplication space, the cost of instruction duplication remains quite high.

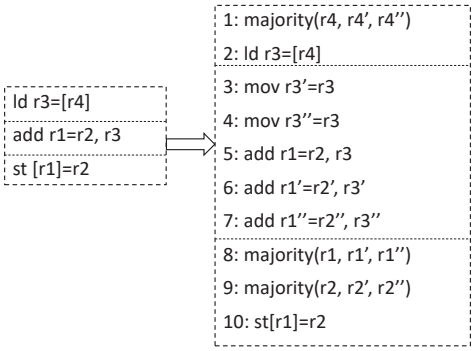

Figure 4. An example of SWIFT-R.

Soft error recovery approaches at the source code level can be classified into duplication-based and checkpoint-based approaches. TRUMP [13] is a typical duplication-based approach. Figure 1 shows the recovery pseudo-code of TRUMP. x is the original copy, and y is the AN-encoded copy of x. y and x have a predefined relationship during normal program execution: $y = 3x$. An error is detected if the relationship is not satisfied, namely, $y \neq 3x$. In the recovery mechanism, the corrupted copy is inferred by the relationship and is corrected by using the correct copy. The concept of duplication-based approaches is simple; however, many statements are required for insertion into source codes.

Checkpoint-based approaches achieve soft error recovery by preserving and restoring a historical state of a program. Coordinated checkpointing is widely used in parallel systems, where the nodes collectively reach a barrier that serves as a consistent state for restarting from a checkpoint. One problem of coordinated checkpointing is that it produces massive amounts of input/output (I/O) from nodes at the checkpoint barrier, resulting in a considerable time overhead. Amrizal et al. proposed a speculative checkpointing technique to tackle this problem [18]. They predicted whether a memory write was the last

one before the next checkpoint and thus can be speculatively checkpointed early. Although this approach performed well, it is only applicable to parallel systems, limiting its wide applicability. The graph theory has been widely used in many research fields [19,20]. Sharanyan et al. used the graph theory and integer linear programming to set checkpoints for programs [21]. They partitioned the control flow and data flow graphs of the programs. Only variables used for further program execution were saved at the partition points. Although this approach reduces the size of checkpoints, i.e., the space overhead, it does not consider the time overhead. Periodic checkpointing recovery sets checkpoints at a fixed interval. However, the work of [22] demonstrated theoretically that this approach is not the optimal checkpointing strategy for reducing the time overhead. The authors did not propose a solution to optimize the method and did not focus on implementation.

3. Preliminaries

We first describe the fault tolerance process in checkpointing recovery to clarify the time overhead. Then, the necessity for considering the detection locations in deploying checkpoints is explained. The meanings of frequently used terms are listed in Table 1.

Table 1. Meanings of terms

Name	Meaning
Checkpointing time	The time required to preserve a program's running state.
Recovery time of a historical state	The time required to restore the data of the latest checkpoint.
Recovery time of a current state	The time required to execute a program from the latest checkpoint to the place where an error is reported by a detector.
Recovery time	The time required to recover the historical state and current state.
Fault tolerance time	The checkpointing time and recovery time.
Overall program execution time	The original program execution time and fault tolerance time.

A program is blocked during a checkpoint operation and resumes after the operation. As shown in Figure 5, when no soft errors are detected during program execution, the overall program execution time is $T + t_0 + t_1 + t_4$, where T is the original program execution time, and t_0, t_1 and t_4 are the checkpointing times of the three checkpoints. $T + t_0 + t_1 + t_4$ represents the sum of the original program execution time and the checkpointing time. The fault tolerance time includes the checkpointing time and recovery time. In this case, the checkpointing time is the fault tolerance time since the recovery time is zero. If a soft error occurs and is detected by a detector during program execution, the program is rolled back to the latest checkpoint and resumes. If a soft error is detected at F, the program rolls back to c_2 and then continues executing until it ends. At this time, the overall program execution time is $T + \sum_{i=0}^{4} t_i$, where t_2 is the time of restoring the data of c_2, i.e., the recovery time of the historical state, and t_3 is the execution time from c_2 to F, i.e., the recovery time of the current state. $T + \sum_{i=0}^{4} t_i$ represents the sum of the original program execution time, the checkpointing time and the recovery time, i.e., the sum of the original program execution time and the fault tolerance time.

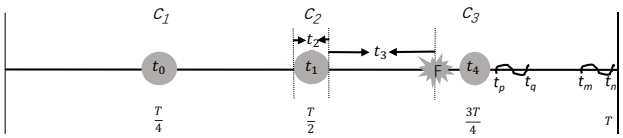

Figure 5. The time overhead of a program.

This description of the time overhead indicates that the overall program execution time decreases with a decrease in the fault tolerance time because the original program execution time does not change. Besides, a decrease in the overall program execution time also reflects a decrease in the fault tolerance time.

Soft errors can be categorized as crash, hang, benign, and silent data corruption (SDC) [10]. Crash and hang refer to a hardware exception and program timeout, respec-

tively [23]. SDC means that an error causes an incorrect program result without any warnings [24]. Being benign means that an error is masked during program execution and does not affect the program results. Different variables or instructions produce different soft error types [9,24,25]. Detectors are placed at program points where errors with adverse effects or of high concern to users may occur to improve detection efficiency. As a result, the distribution of detectors in a program is asymmetric. Periodic checkpointing recovery does not consider the detector locations when setting checkpoints; thus, the time overhead of this method is high. As shown in Figure 5, the recovery time of the current state is very long when an error is detected in $[t_m, t_n]$. An additional checkpoint can be placed at t_m when the checkpointing time at t_m is short to reduce the time overhead. Therefore, it is necessary to consider the detection location in checkpointing recovery. The objective of this study is to reduce the time overhead of fault tolerance. Therefore, the checkpoints in the periodic checkpointing recovery approach are redeployed. The redeployment is conducted by evaluating the time overhead and considering the detector location.

4. Overview of the DLCKPT Approach

An overview of the DLCKPT approach is presented in Figure 6. It includes four parts. The first two parts perform preparation, such as deploying the initial checkpoints and assessing the time overhead. The last two parts redeploy the checkpoints based on the results of the first two parts.

The first part deploys initial checkpoints and generates program segments. Initial checkpoints are deployed for a program using the checkpointing interval (denoted by d in Figure 6) of the periodic checkpointing recovery approach. Then, the program is divided into multiple segments based on the initial checkpoints. As a result, every program segment has one checkpoint. The handling time of the errors detected by the detectors in a program segment is called the time overhead of the program segment. In the second part, the time overhead of each program segment is assessed by considering the detector locations. The time overhead includes the checkpointing time and recovery time.

The third part handles each program segment. It determines whether the checkpoints of a program segment are adequate and inserts additional checkpoints when they are inadequate. Additional checkpoints are initially inserted into the current program segment, and the time overhead is evaluated. If the time overhead is not reduced, the checkpoints are adequate. Otherwise, the checkpoints are inadequate. When the checkpoints are adequate, the process moves to the fourth part. Otherwise, additional checkpoints are inserted, and the next program segment is handled. It should be noted that adequate (inadequate) refers to the adequate (inadequate) number of checkpoints.

The fourth part determines whether the checkpoints of a program segment are redundant, in which case they are deleted. The checkpoints of a program segment are initially deleted, and its time overhead is evaluated. If the time overhead is reduced, the checkpoints are redundant and are deleted. Otherwise, the checkpoints are not redundant and cannot be deleted. When the current program segment has been handled, the process moves to the next program segment. When all program segments have been dealt with, the DLCKPT process ends.

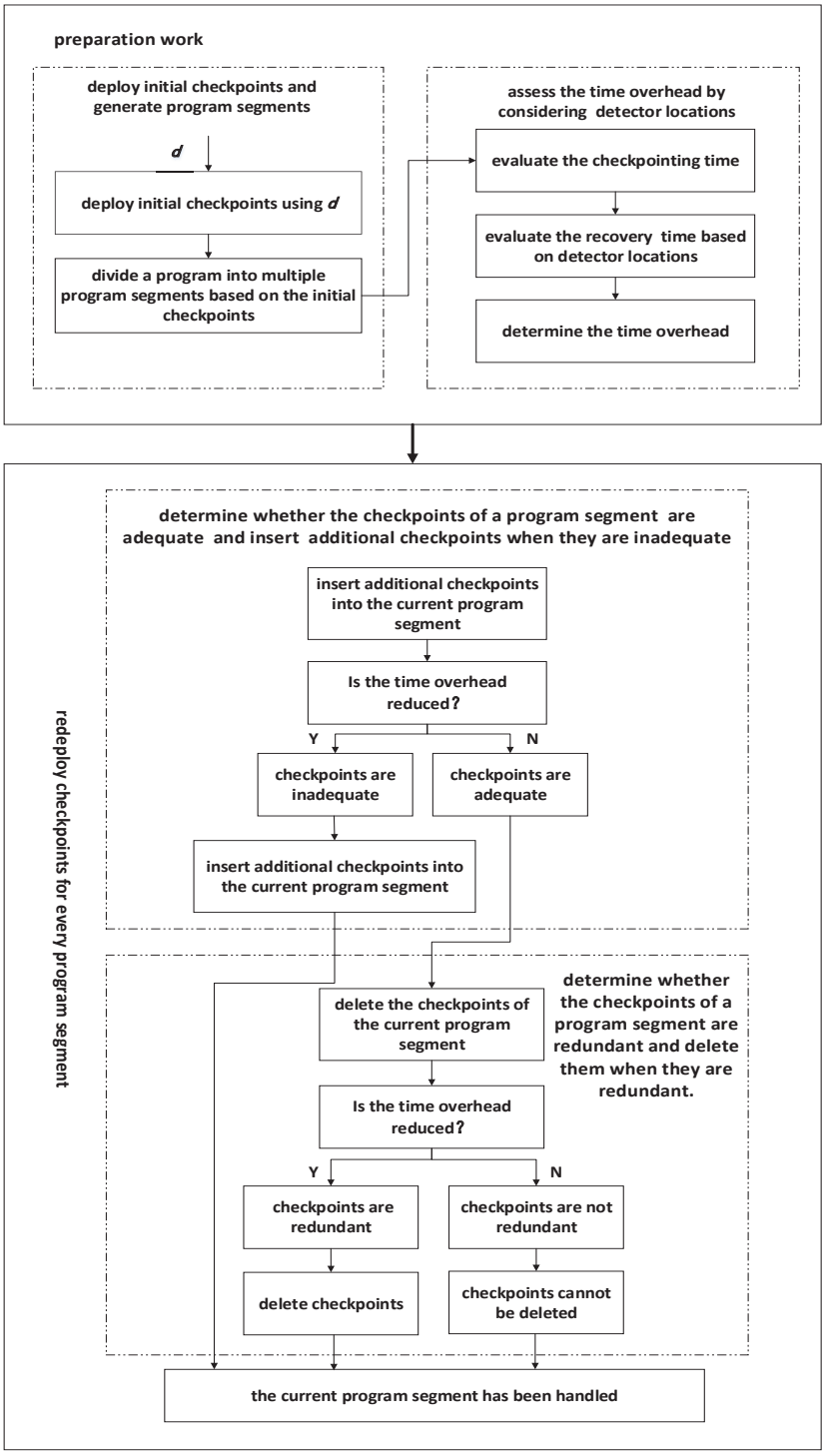

Figure 6. The overview of the DLCKPT approach.

5. The DLCKPT Approach

The four parts of the proposed DLCKPT approach are described in detail in Sections 5.1–5.4, and the overall process of DLCKPT is presented in Section 5.5.

5.1. Deploy Initial Checkpoints and Generate Program Segments

The checkpoints of a program are deployed based on the checkpointing interval of the periodic checkpointing recovery approach. These checkpoints are called initial checkpoints. Then, the program segments are generated based on the initial checkpoints. Figure 7 shows the division of the program segments. At the program level, c_1–c_n are checkpoints. They

divide a program into n program segments, p_1–p_n. At the instruction level, the dynamic instructions of the program are also divided into n instruction segments, s_1–s_n. i_1–$i_{n*\Delta s}$ represents the dynamic instructions, where Δs refers to the number of instructions in every instruction segment. The program execution time between two checkpoints is the same in the periodic checkpointing recovery. Therefore, the instruction segments typically have the same size when dynamic instructions are considered. The instructions of instruction segment s_j are $i_{m(s_j)}$–$i_{j*\Delta s}$, where $m(s_j)$ is the number of the first instruction of s_j. The value of $m(s_j)$ is expressed by Equation (1). As shown in Equation (1), $m(s_j)$ is related to the number of instructions in every instruction segment, i.e., Δs and the number of s_j, i.e., j.

$$m(s_j) = \begin{cases} (j-1) \times \Delta s + 1, & j > 1, \\ 1, & j = 1. \end{cases} \tag{1}$$

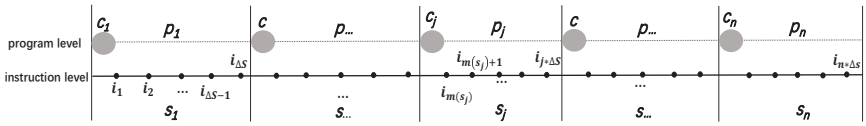

Figure 7. The division of program segments.

5.2. The Time Overhead of a Program Segment

The time overhead of a program segment refers to the handling time of the errors detected by the detectors in the program segment, i.e., the fault tolerance time. Figure 8 presents an example to illustrate the time overhead. It shows the segment "bitstrng" with the detectors, where "bitstrng" is a program in the Mibench benchmark suite that prints a bit pattern of bytes formatted as a string. The detectors are widely applied program invariant assertions and are represented by the assertion statements [4,10] in lines 2, 5, and 9. In this case, the time overhead of this segment is the handling time of the errors detected by the assertion detectors.

The time overhead of a program segment consists of the checkpointing time and recovery time. The latter includes the recovery time of the historical state and the current state. Next, the evaluation of the time overhead is presented in detail. In Figure 9, it is assumed that the detector at F detects an error. In this case, the time overhead consists of the time to generate c_j, i.e., the checkpointing time, the time to restore the data of c_j, i.e., the recovery time of the historical state, and the execution time from c_j to F, i.e., the recovery time of the current state. In this paper, we assume that the time to generate a checkpoint and the time to restore the checkpoint data are the same; therefore, the checkpointing time and the recovery time of the historical state are the same. The checkpointing time of c_j is denoted as $tc(c_j)$; thus, the recovery time of the historical state can be represented by $tc(c_j)$. Two checkpoints have different checkpointing times and recovery times of the historical state. They depend on the size of the checkpointing data. As mentioned above, the recovery time of the current state is the execution time from the latest checkpoint to the location where the error is detected. The recovery time of the current state can be represented by the execution time of the instructions between the latest checkpoint and the location where the error is detected. A program segment may have multiple detectors. Its recovery time of the current state is considered the average recovery time of the current state for all locations of the detectors. The recovery time of p_j is defined in Equation (2), where the second item represents the recovery time of the current state, $c(p_j)$ is the number of dynamic detectors in p_j, θ is the average execution time of one dynamic program instruction, and $a(p_j, k)$ is the instruction number of the k-th dynamic detector in p_j. A detector at the program level corresponds to multiple instructions at the instruction level. In this paper, the number of

the detector is the number of the first instruction corresponding to the detector. Finally, the time overhead of p_j is expressed by Equation (3).

$$tr(p_j) = tc(c_j) + (\frac{\sum_{k=1}^{c(p_j)} (a(p_j,k) - m(s_j))}{c(p_j)} \times \theta) \tag{2}$$

$$o(p_j) = tc(c_j) + tr(p_j) \tag{3}$$

```
1:  j = strwid - (biz + (biz >> 2)- (biz % 4 ? 0 : 1));
2:  assert( byze >= 1);
3:  for (i = 0; i < j; i++)
4:      *str++ = ' ';
5:  assert( biz >= -1);
6:  while (--biz >= 0)
7:  {
8:      *str++ = ((byze >> biz) & 1) + '0';
9:      assert(strwid>i);
10:     if (!(biz % 4) && biz)
            ......
    }
```

Figure 8. An example of a program segment.

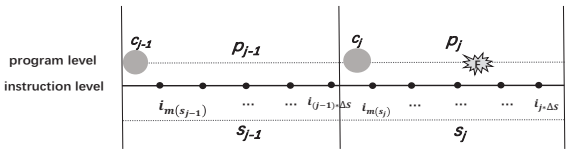

Figure 9. The time overhead of a program segment.

5.3. Determine the Adequacy of Checkpoints

The time overhead of a program segment is evaluated after adding an additional checkpoint to determine if the checkpoints of the program segment are adequate. The additional checkpoint is only inserted into the middle of the program segment. If the time overhead is reduced after adding the checkpoint, the checkpoints of the program segment are inadequate. In this case, a checkpoint is inserted into the middle of the program segment. Otherwise, the checkpoints of the program segment are adequate, and no additional checkpoints are inserted. Next, we evaluate the change in the time overhead of p_j when a checkpoint is inserted into the middle of p_j.

In Figure 10, a checkpoint called ac_j is inserted into the middle of p_j. Then, p_j is divided into two parts. The errors detected by the detectors in the second part of p_j is handled by utilizing ac_j rather than c_j, changing the recovery time of p_j. The new recovery time of p_j is represented by Equation (6), where A is the recovery time of the historical state, B is the recovery time of the current state, $u(p_j)$ in B is the number of detectors in the first half of p_j. The insertion of ac_j also changes the checkpointing time of p_j. The new checkpointing time of p_j is expressed by Equation (7).

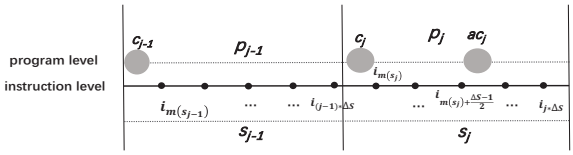

Figure 10. An example of adding a checkpoint.

$$A = \frac{tc(c_j) + tc(ac_j)}{2} \tag{4}$$

$$B = \frac{\sum_{k=1}^{u(p_j)} (a(p_j,k) - m(s_j)) + \sum_{k=u(p_j)+1}^{c(p_j)} (a(p_j,k) - (m(s_j) + \frac{\Delta s-1}{2}))}{c(p_j)} \times \theta \tag{5}$$

$$tr(p_j)' = A + B \tag{6}$$

$$tc(p_j)' = tc(c_j) + tc(ac_j) \tag{7}$$

The time overhead of p_j after inserting ac_j is defined in Equation (8), and the change in the time overhead of p_j is expressed by Equation (9). Equation (9) indicates that the time overhead of p_j is reduced when $ag(p_j) > 0$. In this case, the checkpoint of p_j is inadequate; thus, a checkpoint is inserted into the middle of p_j. Otherwise, the time overhead of p_j is not reduced, indicating that the checkpoint of p_j is adequate, and no additional checkpoints are required.

$$o(p_j)' = tr(p_j)' + tc(p_j)' \tag{8}$$

$$ag(p_j) = o(p_j) - o(p_j)' \tag{9}$$

5.4. Determine the Redundancy of Checkpoints

The time overhead of a program segment is evaluated when its checkpoints are deleted to determine if the checkpoints are redundant. If they are redundant, they are deleted and vice versa.

In Figure 11, c_j and c_{j-1} are the checkpoints of p_j and p_{j-1}, respectively. We merge p_j and p_{j-1} to create a new program segment, p_m. In p_m, the errors detected by the detectors in p_{j-1} are handled by utilizing c_{j-1}, and the errors detected by the detectors in p_j are handled by utilizing c_j. When c_j is deleted, the errors detected by the detectors in p_j are handled by utilizing c_{j-1} rather than c_j, changing the time overhead of p_j and p_m. Since deleting c_j does not affect the time overhead of p_{j-1}, the change in the time overhead of p_m can be considered the change in the time overhead of p_j. Next, we evaluate the change in the time overhead of p_m to determine whether c_j is redundant.

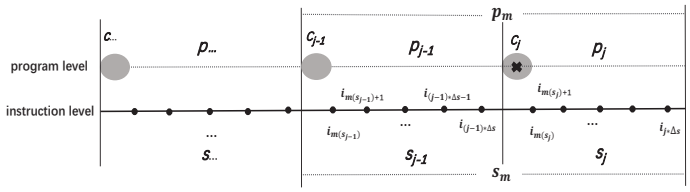

Figure 11. An example of deleting a checkpoint.

Equation (10) expresses the time overhead of p_m when c_j is not deleted. If it is deleted, the time overhead of p_m changes. The new time overhead of p_m is defined by Equation (11). Thus, the change in the time overhead of p_m is expressed by Equation (12). Equation (12) indicates that the time overhead of p_m is reduced when $dg(p_m) > 0$. In this case, the checkpoint of p_j is redundant and can be deleted. Otherwise, the time overhead of p_m is not reduced. Thus, the checkpoint of p_j is not redundant and cannot be deleted.

$$o(p_m) = \frac{\sum_{k=1}^{c(p_{j-1})} (a(p_{j-1},k) - m(s_{j-1})) + \sum_{k=1}^{c(p_j)} (a(p_j,k) - m(s_j))}{c(p_{j-1}) + c(p_j)} \times \theta + \frac{tc(c_{j-1}) + tc(c_j)}{2} + tc(c_{j-1}) + tc(c_j) \tag{10}$$

$$o(p_m)' = \frac{\sum_{k=1}^{c(p_{j-1})} (a(p_{j-1},k) - m(s_{j-1})) + \sum_{k=1}^{c(p_j)} (a(p_j,k) - m(s_j) + \Delta s)}{c(p_{j-1}) + c(p_j)} \times \theta + 2 \times tc(c_{j-1}) \quad (11)$$

$$dg(p_m) = o(p_m) - o(p_m)' \quad (12)$$

5.5. The Process of DLCKPT

This section describes the process of DLCKPT in detail. The processes of the first two parts of DLCKPT are presented in Algorithm 1. The inputs of Algorithm 1 are the checkpointing interval of the periodic checkpointing recovery approach and a program. They are denoted by Δt and P, respectively. The outputs of Algorithm 1 are the program segments and their time overheads, which are represented by the sets PS and OH, respectively.

Algorithm 1 The Processes of the First Two Parts of DLCKPT

Require: Δt, P
Ensure: PS, OH
1: acquire T and $di(P)$ by executing P
2: $n = T/\Delta t$
3: deploy initial checkpoints, c_1-c_n
4: generate program segments, p_1-p_n
5: $\Delta s = size(di(P))/n$
6: $\theta = T/size(di(P))$
7: **for** $j = 1 \rightarrow n$ **do**
8: determine $m(s_j)$ by (1)
9: evaluate $tc(c_j)$
10: determine $c(p_j)$ based on $di(P)$
11: $r = 0$, $z = 0$
12: **for** $k = 1 \rightarrow c(p_j)$ **do**
13: get $a(p_j,k)$
14: $r = r + (a(p_j,k) - m(s_j))$
15: **end for**
16: $z = \frac{r}{c(p_j)} \times \theta$
17: calculate $tr(p_j)$ by (2)
18: calculate $o(p_j)$ by (3)
19: put $o(p_j)$ into OH
20: put p_j into PS
21: **end for**

In Algorithm 1, the execution time and the dynamic instruction set of P are acquired by executing P (Line 1), which are denoted by T and $di(P)$, respectively. Then the number of initial checkpoints is determined based on Δt and T (Line 2). Next, the initial checkpoints are deployed, and program segments are generated (Lines 3–4). The size of every instruction segment and the average execution time of one program instruction are determined (Lines 5–6). The program segments are handled sequentially. First, the number of the first instruction is determined by Equation (1) (Line 8) for handling the current program segment. Then, the checkpointing time is evaluated (Line 9), and the number of dynamic detectors is determined (Line 10). Next, the sum of the number of the re-executed instructions in all detector locations is determined (Lines 12–15), which is represented by r. The re-executed instructions refer to the instructions that are re-executed during a recovery operation. The average recovery time for all detector locations is calculated by Equation (2) (Line 17). Further, the time overhead is calculated by Equation (3), and is inputted into

the set of the time overhead of program segments, namely, OH (Lines 18–19). Finally, the current program segment is input into the set of program segments, namely, PS (Line 20).

The processes of the last two parts of DLCKPT are presented in Algorithm 2. The inputs of Algorithm 2 are the outputs of Algorithm 1, i.e., PS and OH. The output of Algorithm 2 is the set of redeployed checkpoints, RC. In Algorithm 2, the program segments are handled sequentially. An additional checkpoint is initially set in the middle of the current program segment (Line 3). Then, the number of detectors in the first half is determined (Line 4). The checkpointing time of the additional checkpoint is evaluated (Line 5). Next, the new average recovery time of the historical state and the current state for all detector locations are determined by Equations (4) and (5), respectively (Line 6). Next, the new recovery time and checkpointing time are calculated by Equations (6) and (7), respectively (Line 7). The new time overhead is determined by Equation (8) (Line 8). Finally, the change in the time overhead is obtained from Equation (9) (Line 9).

Algorithm 2 The Processes of the Last Two Parts of DLCKPT

Require: PS, OH
Ensure: RC
1: $plas = p_1$
2: **for** $j = 1 \rightarrow n$ **do**
3: set an additional checkpoint ac_j in the middle of p_j
4: determine $u(p_j)$
5: evaluate $tc(ac_j)$
6: determine A and B by (4) and (5)
7: calculate $tr(p_j)'$, $tc(p_j)'$ by (6) and (7)
8: determine $o(p_j)'$ by (8)
9: obtain $ag(p_j)$ by (9)
10: **if** $ag(p_j) > 0$ **then**
11: set ac_j in the middle of p_j at last
12: put c_j and ac_j into RC
13: $plas = second(p_j)$
14: continue
15: **else**
16: $p_m = merge(p_j, plas)$
17: obtain $o(p_m)$ and $o(p_m)'$ from (10) and (11)
18: calculate $dg(p_m)$ by (12)
19: **if** $dg(p_m) < 0$ **then**
20: put c_j into RC
21: $plas = p_j$
22: **else**
23: delete c_j
24: $plas = p_m$
25: **end if**
26: **end if**
27: **end for**

When the change in the time overhead is greater than 0, an additional checkpoint is set in the middle of the current program segment, and the additional checkpoint and the original checkpoint of the current program segment are input into RC (Lines 11–12). The second half of the current program segment is considered the latest program segment, and the next program segment is handled (Line 13). At the beginning of Algorithm 2, the first program segment is considered the latest program segment. When the change in the time overhead is less than 0, the latest and current program segments are merged to create a new program segment, p_m (Line 16). Then, the time overhead of p_m is obtained from Equation (10). In addition, the time overhead of p_m after deleting the checkpoint of the current program segment is obtained from Equation (11) (Line 17). The change in

the time overhead of p_m is calculated by Equation (12) (Line 18). When the change in the time overhead of p_m is less than 0, the checkpoint of the current program segment remains and is input into *RC* (Line 20). The current program segment is then considered the latest program segment (Line 21). Otherwise, the checkpoint of the current program segment is deleted, and p_m is considered the latest program segment (Lines 23–24).

6. Experiment and Results

An experiment was conducted to evaluate the DLCKPT approach. The experimental setup is presented in Section 6.1, and the results are described in Section 6.2. The overall program execution time, recovery rate, and space overhead of the DLCKPT are compared with the periodic checkpointing recovery approach.

6.1. Experimental Setup

Fault injection is widely used to simulate single-event effects (SEEs) and the resulting soft errors in experiments [4,10]. Pin is a binary instrumentation framework for the IA-32 and x86-64 instruction sets, enabling the creation of dynamic program analysis tools [26]. Similar to [4,10,24], Pin was used for fault injection by altering one bit in the register or memory cell from 0 to 1 or from 1 to 0. Besides, Pin was also used to create the dynamic instrument tool that was used for generating dynamic instructions and program segments. The dynamic instrument tool tracks the program execution and divides the program into multiple segments based on the program execution and checkpointing interval. The original program execution time was T. Two independent experiments were performed with different checkpointing intervals of the periodic checkpointing recovery: $T/4$ and $T/3$. BLCR is a representative system-level checkpointing tool for the Linux platform[15]. It saves the entire state of a process to disk, such as the process ID, CPU registers and virtual memory, and later restores the state. The BLCR provides instructions to execute checkpoint and recovery operations. A user is only required to execute the instructions but does not have to implement them. We employed BLCR to execute the checkpoint and recovery operations. The platform for the experimental evaluation was Ubuntu 10.04, running on a Dell Workstation with an i7 processor. The pseudo-code of the procedure of checkpoint and recovery operations is shown in Figure 12. *Cr_run*, *cr_checkpoint*, and *cr_restart* are functions provided by the BLCR. First, *cr_run* is used to run a program. An identification number of the process related to the program is obtained. Then, *cr_checkpoint* is executed to generate a checkpoint file after a checkpoint interval. Next, a fault is injected into the process. Finally, a while loop is executed. The value of the variable *code* is obtained from the loop. It is not equal to zero when the assertion detector in the program detects and reports an error. In this case, *cr_restart* is used to launch a recovery opertion, and *context*.$*pid* is the name of the checkpoint file which is generated by *cr_checkpoint*.

```
cr_run ./p_name, p_data
get_pid
sleep checkpoint_interval
cr_checkpoint $pid
.....
inject.py $pid   $injection_seq
while true
do
   code=$?
    if [[code!=0]]
    then
      cr_restart ./context.$pid
      break
    fi
done
```

Figure 12. The pseudo-code of the procedure of checkpoint and recovery operations.

If a soft error occurs and is detected by the assertion detector, the program is rolled back to the recent checkpoint and continues running from the checkpoint until it ends. When a soft error occurs but is not detected by any detector, the program will continue running in a normal state. We chose four metrics to evaluate the performance of DLCKPT. (1) The overall program execution time when a soft error is detected. It is denoted as ep and defined as the percentage decrease in the overall program execution time when a soft error is detected compared to the periodic checkpointing recovery approach. Thus, it is the difference in the overall program execution time between the periodic checkpointing recovery approach and the DLCKPT divided by the overall program execution time of the periodic checkpointing recovery approach. (2) The overall program execution time when a soft error occurs. It is denoted as fp and defined as the percentage decrease in the overall program execution time when a soft error occurs compared to the periodic checkpointing recovery approach. Its expression is similar to that of ep. Detectors may not detect all soft errors. When a soft error occurs but is not detected by any detector, there are no reports, and no additional time is required for recovering from the error, but the checkpointing time still exists, increasing the overall program execution time. Therefore, we used this metric. (3) Recovery rate. It is the ratio of the number of errors detected by detectors and recovered by checkpoints to the number of errors detected by detectors. (4) Space overhead. It is the ratio of the size of the checkpoint to the size of the program. The programs used for the evaluation were obtained from the Mibench and Siemens benchmark suites. They were replace (which computes the input data statistics), bitstrng (which prints bit pattern of bytes formatted to string), rad2deg (which converts between radians and degrees), and isqrt (which is a base-two analog of the square root algorithm). The input data of the replace program are 7,970,000 random characters. The input data of bitstrng are 115,200 random numbers. The input data of rad2deg are 1,150,000 random numbers. The input data of isqrt are 2,150,000 random numbers.

6.2. Experimental Results and Evaluation

6.2.1. The Overall Program Execution Time When a Soft Error is Detected

Table 2 lists the evaluation results of the overall program execution time when a soft error is detected. All values of ep were greater than 0 for all programs, indicating that DLCKPT had a shorter overall program execution time than the periodic checkpointing recovery approach. When DLCKPT redeployed the checkpoints of the periodic checkpointing recovery approach with a checkpoint interval of T/4 (T/3), the average percentage decrease in the overall program execution time was 15% (11.4%). These results show the excellent performance of DLCKPT regarding the overall program execution time when a soft error is detected.

Table 2. The results of ep.

Programs	T/4	T/3
replace	22.1%	14.1%
bitstrng	4%	8%
rad2deg	10%	10%
isqrt	22.3%	13.5%
average	15%	11.4%

6.2.2. The Overall Program Execution Time When a Soft Error Occurs

Table 3 lists the evaluation results of the overall program execution time when a soft error occurs. All values of fp were all greater than 0 for all programs, indicating that DLCKPT had a shorter overall program execution time than the periodic checkpointing recovery approach when a soft error occurred. When DLCKPT redeployed the checkpoints of the periodic checkpointing recovery approach with a checkpoint interval of T/4 (T/3), the average percentage decrease in the overall program execution time was 16% (11%).

These results show the excellent performance of DLCKPT regarding the overall program execution time when a soft error occurs.

Table 3. The results of fp.

Programs	T/4	T/3
replace	25.6%	15.3%
bitstrng	3%	2%
rad2deg	11%	12%
isqrt	23.2%	14.5%
average	16%	11%

The evaluation results indicate that the average percentage decrease in the overall program execution time was 13.4% (averages of 15%, 11.4%, 16%, and 11%). These results show the effectiveness of DLCKPT in reducing the time overhead.

6.2.3. Recovery Rate

Figure 13 shows the results of the recovery rate. As shown in Figure 13a, the average recovery rate was 99.4% for the periodic checkpointing recovery approach with a checkpoint interval of T/4 and 99.3% for DLCKPT. The difference in the recovery rate between the two methods was only 0.1%, which was negligible. Figure 13b shows that the average recovery rate was 99.2% for the periodic checkpointing recovery approach with a checkpoint interval of T/3 and 99.5% for DLCKPT. The two recovery rates were almost equal. The average recovery rate of 99.4% and 99.2% was 99.3% for the periodic checkpointing recovery approach, and that of 99.3% and 99.5% was 99.4% for DLCKPT, indicating a small difference between the two methods. The reason that the recovery rate is not 100% is that the data saved by the checkpoints is not always correct. In the following, we use the program segment in Figure 8 as an example to provide an explanation. We assume a soft error occurs when the statement in line 1 is executed; thus, the value of the variable *biz* is -2 (an incorrect value). After the execution of the statement in line 1, the assertion in line 2 is executed. The assertion represents a detector. It detects whether a soft error has occurred in the variable *byze*. In this case, the assertion is not false, indicating no errors have been detected. The for loop in line 3 and line 4 is executed subsequently. We assume that a checkpoint called c_1 is generated by the checkpointing strategy after executing the for loop.

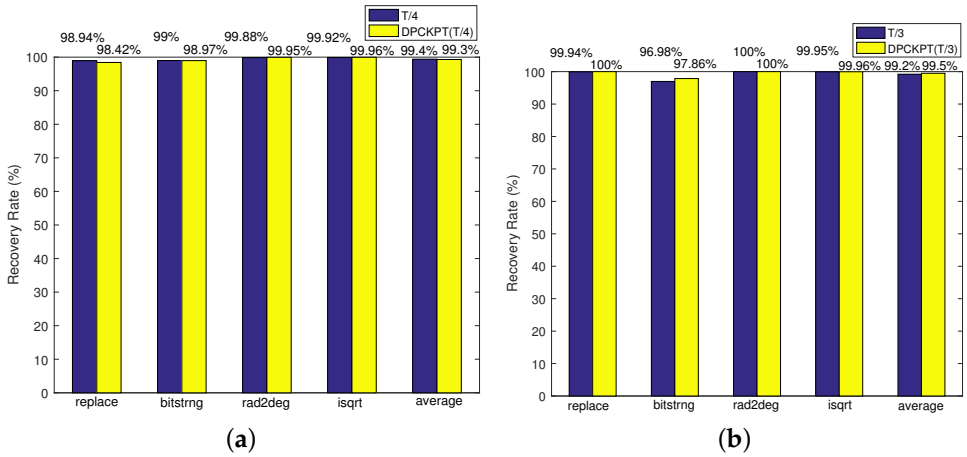

Figure 13. (**a**) The result of the recovery rate of the periodic checkpointing recovery approach with a checkpoint interval of T/4 and that of DLCKPT that redeploys the checkpoints of checkpoint interval of T/4. (**b**) The result of the recovery rate of the periodic checkpointing recovery approach with a checkpoint interval of T/3 and that of DLCKPT that redeploys the checkpoints of checkpoint interval of T/3.

It should be noted that the value of the variable *biz* saved by c_1 is incorrect. Next, the assertion in line 5 is executed. The assertion is false because *biz* is not greater than or equal to -1, i.e., an error in the variable *biz* has been detected. In this case, the checkpoint strategy rolls the program back to c_1, and the program continues running from c_1. Because the value of *biz* saved by c_1 is incorrect, the program rolls back to c_1 once again when it executes the statement in line 5 again; thus, the recovery fails.

Although the recovery rate was not 100%, it was higher than 99%, satisfying the needs of most users. The recovery rate is related to the data stored in a checkpoint. The more accurate the data stored in the checkpoint, the higher the recovery rate is. The reason for our accuracy values is that most of the data stored in a checkpoint by BLCR was correct. In this study, we focused on optimizing the time overhead, not the recovery rate. We will focus on the latter in a future study by storing more accurate data in a checkpoint.

6.2.4. Space Overhead

The results of the space overhead are presented in Figure 14. As shown in Figure 14a, the average space overhead was 44.5% for the periodic checkpointing recovery approach with a checkpoint interval of T/4 and 33.2% for DLCKPT. Figure 14b shows that the average space overhead was 44.2% for the periodic checkpointing recovery approach with a checkpoint interval of T/3 and 36% for DLCKPT. The average space overhead was 44.4% (44.5% and 44.2%) for the periodic checkpointing recovery approach and 34.6% (33.2% and 36%) for DLCKPT. Figure 14a,b also show that the space overhead of bitstrng was lower for the periodic checkpointing recovery approach than for DLCKPT. The reason is that DLCKPT did not consider the space overhead when redeploying checkpoints. Thus, the space overhead differed for the different programs. Regardless, the experimental results show that the averaged space overhead across the four programs was slightly lower for DLCKPT than for the periodic checkpointing recovery approach. Figure 14a,b indicate that the space overheads of the two approaches were not very low. The lowest space overhead was 39.3% for the periodic checkpointing recovery approach and 23.7% for DLCKPT. The reason is that the space overhead is related to the data stored in a checkpoint. The lower the data volume of the checkpoint, the lower the space overhead is. Many intermediate data points were generated during the execution of the programs and stored in a checkpoint by the BLCR, resulting in a large checkpointing data volume. The optimization of the space overhead will be considered in a future study.

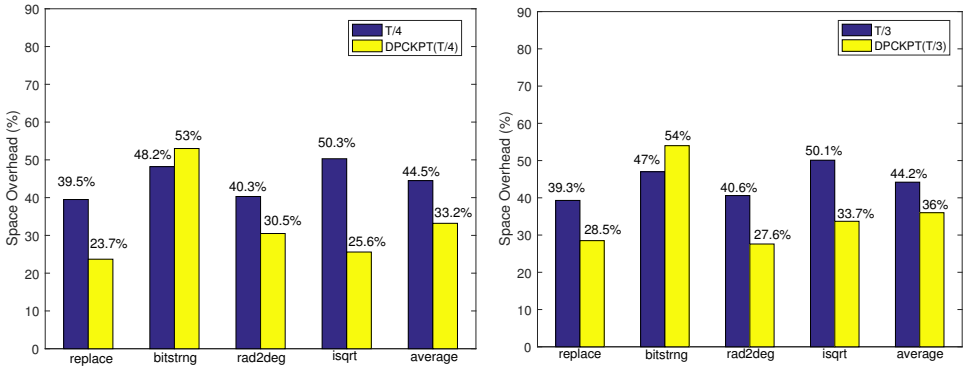

Figure 14. (**a**) The result of the space overhead of the periodic checkpointing recovery approach with a checkpoint interval of T/4 and that of DLCKPT that redeploys the checkpoints of checkpoint interval of T/4. (**b**) The result of the space overhead of the periodic checkpointing recovery approach with a checkpoint interval of T/3 and that of DLCKPT that redeploys the checkpoints of checkpoint interval of T/3.

7. Conclusions and Future Work

In this paper, we propose the DLCKPT approach, which deploys checkpoints by considering detector locations to reduce the time overhead. This method first deploys initial checkpoints and generates program segments based on the periodic checkpointing recovery approach. It handles each program segment sequentially and determines whether the checkpoints of the program segment are adequate or redundant by evaluating the change in the time overhead when additional checkpoints are inserted or deleted. In particular, the detector locations are considered in the evaluation of the time overhead.

An experiment was conducted to evaluate the DLCKPT. The result demonstrated that the DLCKPT had a lower time overhead than the periodic checkpointing recovery approach. The recovery rates were similar for the two methods, and the average space overhead was slightly lower for DLCKPT than for the periodic checkpointing recovery approach.

In a future study, we will optimize the space overhead and recovery rate of the DLCKPT by determining which data should be stored in the checkpoints. In this study, the checkpoints were only inserted into the middle of the program segments to determine the checkpoints' adequacy. Other locations of the program segments will be considered to reduce the time overhead. In addition, multiple program segments rather than only two program segments will be merged into one program segment to determine the checkpoints' redundancies.

Author Contributions: Conceptualization, N.Y. and Y.W.; investigation, N.Y. and Y.W.; methodology, N.Y. and Y.W.; software, N.Y.; supervision, Y.W.; validation, N.Y.; writing—original draft preparation, N.Y.; writing—review and editing, Y.W.; funding acquisition, Y.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Pre-research Project, China with grant No. WDZC20215250117.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Gupta, N.; Shah, A.P.; Kumar, R.S.; Gupta, T.; Khan, S.; Vishvakarma, S.K. On-Chip Adaptive VDD Scaled Architecture of Reliable SRAM Cell with Improved Soft Error Tolerance. *IEEE Trans. Device Mater. Reliab.* **2020**, *20*, 694–705. [CrossRef]
2. Hashimoto, M.;Liao, W. Soft Error and Its Countermeasures in Terrestrial Environment. In Proceedings of the 25th Asia and South Pacific Design Automation Conference (ASP-DAC), Beijing, China, 13–16 January 2020; pp. 617–622. [CrossRef]
3. Binder, D.; Smith, E.C.; Holman, A.B. Satellite Anomalies from Galactic Cosmic Rays. *IEEE Trans. Nucl. Sci.* **1975**, *22*, 2675–2680.
4. Ma, J.C.; Yu, D.Y.; Wang, Y.; Cai, Z.B.; Zhang, Q.X.; Hu, C. Detecting Silent Data Corruptions in Aerospace-Based Computing Using Program Invariants. *Int. J. Aerosp. Eng.* **2016**, *2016*, 1–10. [CrossRef]
5. Tan, C.Y.; Li, Y.; Cheng, X.; Han, J.; Zeng, X.Y. General Efficient TMR for Combinational Circuit Hardening Against Soft Errors and Improved Multi-Objective Optimization Framework. *IEEE Trans. Circuits Syst.* **2021**, *68*, 3044–3057. [CrossRef]
6. Kiani, V.; Reviriego, P. Improving Instruction TLB Reliability with Efficient Multi-bit Soft Error Protection. *Microelectron. Reliab.* **2019**, *93*, 29–38. [CrossRef]
7. Keller, A.M.; Wirthlin, M.J. Partial TMR for Improving the Soft Error Reliability of SRAM-Based FPGA Designs. *IEEE Trans. Nucl. Sci.* **2021**, *68*, 1023–1031. [CrossRef]
8. Didehban, M.; Lokam, S.R.D.; Shrivastave, A. InCheck: An In-application Recovery Scheme for Soft Errors. In Proceedings of the 54th ACM/EDAC/IEEE Design Automation Conference, Austin, TX, USA, 18–22 June 2017; pp. 1–6. [CrossRef]
9. Ma, J.C.; Duan, Z.T.; Tang, L. GATPS: An Attention-based Graph Neural Network for Predicting SDC-causing Instructions. In Proceedings of the 39th VLSI Test Symposium, San Diego, CA, USA, 25–28 April 2021; pp. 1–6.
10. Yang, N.; Wang, Y.F. Radish: Enhancing Silent Data Corruption Detection for Aerospace-Based Computing. *Electronics* **2021**, *10*, 61.
11. Benacchio, T.; Bonaventura, L.; Altenbernd, M.; Cantwell, C.D.; Düben, P.D.; Gillard, M.; Giraud, L.; Göddeke, D.; Raffin, E.; Teranishi, K.; et al. Resilience and Fault Tolerance in High-Performance Computing for Numerical Weather and Climate Prediction. *Int. J. High Perform. Comput. Appl.* **2021**, *35*, 285–311.
12. Didehban, M.; Shrivastava, A. A Compiler Technique for Processor-Wide Protection From Soft Errors in Multithreaded Environments. *IEEE Trans. Reliab.* **2018**, *67*, 249–263. [CrossRef]
13. Reis, G.A.; Chang, J.; August, D.I. Automatic Instruction-Level Software-Only Recovery. *IEEE Micro* **2007**, *67*, 36–47. [CrossRef]
14. Guo, Y.M.; Wu, H.; Chai, W.X.; Ma, J.Z.; Zhou, G.C. Integrity Checking based Soft Error Recovery Method for DSP. In Proceedings of the Prognostics and System Health Management Conference, Chengdu, China, 19–21 October 2016; pp. 1–4. [CrossRef]

15. Yang, N.; Wang, X.; Wang, Y.; Zhai, Q. Dependent and Heterogeneous Process Migration Based on Checkpoints. In Proceedings of the IEEE International Conference on Parallel and Distributed Processing with Applications, Big Data and Cloud Computing, Exeter, UK, 17–19 December 2020; pp. 84–92. [CrossRef]
16. Restrepo-Calle, F.; MartÍnez-Álvarez, A.; Cuenca-Asensi, S.; Jimeno-Morenilla, A. Selective SWIFT-R: A Flexible Software-Based Technique for Soft Error Mitigation in Low-Cost Embedded Systems. *J. Electron. Test.* **2013**, *29*, 825–838.
17. Didehban, M.; Shrivastava, A.; Lokam, S.R.D. NEMESIS: A Software Approach for Computing in Present of Soft Errors. In Proceedings of the IEEE International Conference on Computer-Aided Design, Irvine, CA, USA, 13–16 November 2017; pp. 297–304.
18. Amrizal, M.A.; Uno, A.; Sato, N.Y.;Takizawa, H.; Kobayashi, H. Energy-Performance Modeling of Speculative Checkpointing for Exascale Systems. *IEICE Trans. Inf. Syst.* **2017**, *12*, 2749–2760. [CrossRef]
19. Quezada-Sarmiento, P.A.; Elorriaga, J.A.; Arruarte, A.; Jumbo-Flores, L.A. Used of Web Scraping on Knowledge Representation Model for Bodies of Knowledge as a Tool to Development Curriculum. In *Trends and Applications in Information Systems and Techinologies*; Springer: Berlin, Germany, 2021; Volume 2, pp. 611–620. [CrossRef]
20. Ma, J.C.; Duan, Z.T.; Tang, L. Deep Soft Error Propagation Modeling Using Graph Attention Network. *J. Electron. Test.* **2022**, *38*, 303–319. [CrossRef]
21. Sharanyan, S.; Kumar, A. An optimized Checkpointing Based Learning Algorithm for Single Event Upsets. In Proceedings of the IEEE International Conference on Annual Computer Software and Applications, Seoul, Republic of Korea, 19–23 July 2013; pp. 395–400.
22. Subasi, O.; Krishnamoorthy, S. On the Theory of Speculative Checkpointing: Time and Energy Considerations. In Proceedings of the ACM International Conference on Computing Frontiers, Ischia, Italy, 8–10 May 2018; pp. 165–172. [CrossRef]
23. Sangchoolie, B.; Pattabiraman, K.; Karlsson, J. An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors in Programs. *IEEE Trans. Dependable Secur. Comput.* **2020**, *2020*, 1–18.
24. Yang, N.; Wang, Y. Predicting the Silent Data Corruption Vulnerability of Instructions in Programs. In Proceedings of the IEEE International Conference on Parallel and Distributed Systems, Tianjin, China, 4–6 December 2019; pp. 862–869. [CrossRef]
25. Li, G.P.; Pattabiraman, K.; Hari, S.K.S.; Sullivan, M.; Tsai, T. Modeling Soft-Error Propagation in Programs. In Proceedings of the IEEE International Conference on Dependable Systems and Networks, Luxembourg, 25–28 June 2018; pp. 27–38.
26. Ma, J.C.; Wang, Y. Characterization of Stack Behavior under Soft Errors. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, Lausanne, Switzerland, 27–31 March 2017; pp. 1534–1539.