Cluster Computing
https://doi.org/10.1007/s10586-022-03726-9

Quantifying the impact of data replication on error propagation

Zuhal Ozturk[1] · Haluk Rahmi Topcuoglu[1] · Mahmut Taylan Kandemir[2]

Received: 20 May 2022 / Revised: 31 July 2022 / Accepted: 22 August 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract

Various technological developments in the microprocessor world make modern computing systems more vulnerable to soft errors than in the past, and consequently fault tolerance techniques are becoming increasingly important in various application domains. While in general fault tolerance methods are known to achieve high levels of reliability, they can also introduce significant performance, energy, and memory overheads, which can be reduced by employing such techniques selectively, as opposed to indiscriminately. Data Replication is used to prevent error propagation across hardware components and application program data structures by replicating application program's data. When using data replication, many factors need to be taken into account, including which data structures/elements to replicate, how many times to replicate a given data element, and which threads to protect (in a multithreaded application). These and similar factors define what can be termed as "replication space". This study defines a replication space, and systematically explores protection techniques of various strengths/degrees, quantifying their impacts on memory consumption, performance, and error propagation. Our experimental analysis reveals that different degrees of protection levels bring different outcomes based on the application specifics. In particular, while error propagation is limited, to a certain extent, when employing data replication in multithreaded applications where the thread do not communicate/share data much, the speed of error propagation across threads can be quite fast in applications where threads are more tightly coupled. Additionally, our results indicate that in certain cases where error propagation is low, the effect of data replication on error propagation can be negligible.

Keywords Soft errors · Error propagation · Data replication · Memory and performance overheads · Multithreading

1 Introduction

Soft errors are one-time unpredictable events that cause the application program execution to enter an undesirable state. Cosmic radiation, alpha particles from packaging materials, voltage fluctuations, and high temperatures are among the main causes of soft errors [1]. Various technological developments in the microprocessor world in recent years, such as smaller transistors, increased operating frequencies, and increasingly more aggressive power and performance optimizations, have rendered different types of computing systems more vulnerable to such errors.

To counter this increasingly pressing soft error problem, various fault tolerance methods have been introduced in the literature. The fault tolerance techniques can be implemented at hardware [2, 3] or software layers [4, 5]. There also exist studies that target both software and hardware layers [6, 7]. It is to be emphasized that, while fault tolerance methods help to achieve more reliability in general, indiscriminate use of such methods can also introduce significant performance, energy, and memory overheads. Motivated by this observation, several prior works have also explored what is called "selective reliability", in an attempt to reduce the overheads brought by the fault tolerance methods [8].

✉ Haluk Rahmi Topcuoglu
haluk@marmara.edu.tr

Zuhal Ozturk
zuhal.ozturk@marmara.edu.tr

Mahmut Taylan Kandemir
mtk2@psu.edu

[1] Computer Engineering Department, Faculty of Engineering, Marmara University, 34854 Istanbul, Turkey

[2] Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, USA

Published online: 13 September 2022

"Data replication" is one of the popular software-based fault tolerance methods that aim to increase reliability by replicating data as well as the operations performed on them. Error detection/correction is provided by comparing the redundant copies for consistency. One of the drawbacks of data replication is the performance and memory consumption overheads it involves. In particular, aggressively replicating all the data used by an application program can increase the program's memory consumption/footprint significantly. Assume as an example that we have an application that operates on 10 integer variables and we employ data replication for error detection. This means that a copy should be created for all 10 variables, thus doubling the memory consumption. Moreover, if it is desired to provide error correction by using the data replication approach, the memory consumption would further increase since at least 3 copies are needed for each and every data element. In addition, since the redundant instructions that operate on replicated data, comparison instructions, and instructions that are executed for error detection would also be added to the code, and this further increases application execution latencies.

In an attempt to reduce such additional memory/execution time costs, "partial/selective data replication" can be employed. In this strategy, certain data elements and/or structures are considered to be more critical than others, in terms of application correctness or error propagation speed. For example, if a data element/location is involved with data dependencies with a large number of other data elements/locations, the error in this data can be expected to spread faster. Based on this observation, replicating only "critical data", instead of all data, helps to reduce the replication overheads. Further, since the user-perceived reliability is closely related to the "criticality" of data, replicating them can improve reliability [9]. As a result, it may be possible to achieve substantial reliability enhancements while limiting the incurred overheads.

In this paper, the impacts of selective data replication at different levels on error propagation, memory consumption and performance are examined in a quantitative fashion. More specifically, this paper makes the following main contributions:

- To capture the degree of protection provided via selective data replication, we define percentage of replicated data, number of replicas, items that are replicated and threads that are protected as our "protection parameters". These parameters define our "replication space". Further, to specify the faulty environment, we define erroneous data rate, bit span, number of threads that contain errors, and threads in which errors are created as additional parameters.
- We define several data replication options with different degrees of protection by using the proposed protection parameters, and then, they are compared in terms of their impact on the error propagation, memory consumption, and execution time by performing extensive fault injection experiments.
- We study the behavior of the defined data replication options against different errors. The overheads brought up by replication on memory and performance is different for different protection scenarios generated with different values of our protection parameters. Examining these scenarios on several faulty environments helps us understand when the introduced overhead is acceptable.
- We investigate the problem of improving reliability under certain memory constraints by modulating the data and threads that are selected for data replication. If we change the replicated variable(s) and the protected thread(s), the memory consumption remains the same since the amount of the redundant memory is not changed. By changing the variables and threads, we explore different options with the goal of reaching a more efficient data replication scenario.

Although the data replication technique can be used for both single-threaded and multi-threaded applications, we consider only the multi-threaded applications in this study. Several multi-threaded applications are utilized for the fault injection experiments, where data replication is implemented by providing private copies of shared variables to threads. In this way, the error propagation between threads via shared variable is prevented and the error propagation speed is slowed down. The block-wise matrix multiplication, two-dimensional Jacobi computation, and the heat diffusion applications are used as target codes in our fault injection experiments. The collected experimental results show that error propagation is slowed down when different protection levels are utilized. These protection levels have also different costs, and it is observed that the memory consumption caused by the data replication can increase up to 9 times. On the other hand, this costly protection may not be acceptable for many applications and execution scenarios. Experiments with different faulty environments demonstrate that, when the error propagation is limited, the decrease of the error propagation provided by the protection can be negligible (i.e., error rate decreases from 0.1 to 0.05). Moreover, in applications where the threads are tightly coupled, e.g., heat diffusion, we observe that providing private copies to threads does not help much in preventing error propagation. Thus, in such applications, increasing reliability by isolating faulty thread(s) from others is difficult to achieve. Besides proposed error and protection parameters, additional fault

injection experiments are performed with different thread counts and different implementations. The results show that when the implementation details are changed, that affects the error propagation and makes it possible to slow down error propagation in certain cases. For instance, in two-dimensional Jacobi iteration when a different data distribution is used the error rate increases from 0.36 to 0.42.

The rest of the paper is organized as follows. We summarize the prior relevant work in Sect. 2. Section 3 explains our framework, as well as error and protection parameters, in detail. Section 4 presents the results from our experimental evaluations, and Sect. 5 concludes the paper by summarizing our major findings.

2 Related work

In this section, we summarize the related work on vulnerability factors and fault propagation metrics. It is followed by the error-related metrics and analysis methods for the Silent Data Corruption (SDC). Finally, we discuss the data replication techniques proposed in the literature.

Vulnerability factors are utilized to evaluate techniques that attack soft errors, and different vulnerability metrics are proposed to measure the error vulnerability at different layers [10–14]. The Architectural Vulnerability Factor (AVF) represents the probability that an error in processor structure is propagated to the output [10]. Cache Vulnerability Factor (CVF) [11] and Register Vulnerability Factor (RVF) [12], on the other hand, represent the probability that errors in caches and registers, respectively, are transitioned to the output. As opposed to these approaches that quantify/address vulnerability from a hardware perspective, Program Vulnerability Factor (PVF) measures the vulnerability of the application regardless of the underlying architecture [13]. Thread Vulnerability Factor (TVF) is proposed to measure the vulnerability of the threads in multithreaded applications [14]. Note that TVF considers data dependencies between threads when measuring error vulnerability.

Additionally, several error propagation metrics have been proposed and evaluated in the literature. For example, Error Propagation Speed (EPS) refers to processes with erroneous data per iteration in MPI-based applications [15]. Error masking events are recorded and their frequency values are measured in another study [16].

The FlipTracker framework is designed to measure the error resiliency of HPC applications [17]. This framework analyzes several HPC applications and computational patterns that provide error resilience for such applications. SpotSDC is another framework that monitors and visualizes SDC in HPC applications [18]. Error propagation has also been studied with analytical models in the literature. The Support Vector Regression is used for error detection, and an error detection strategy is presented based on the model found by regression [19]. In comparison, in TRIDENT framework, SDC predictions are performed analytically where propagation is modeled based on static instructions, control flow, and memory dependencies [20]. The TRIDENT framework is improved to work with more than one input [21], and also adapted to work with GPU applications as well [22].

Software Implemented Fault Tolerance (SWIFT) is a software-based single-threaded fault tolerance technique that aims to provide reliability using redundancy [4]. The values used by the program are calculated twice and compared. S-SWIFT-R [23], an improved version of SWIFT, provides selective hardening by replicating only selected registers.

Prior works have also employed different transformation rules[1] to provide data-flow protection [24, 25]. In particular, several techniques have been proposed by using different combinations of the transformation rules [26], and they are compared in terms of error detection rate, execution time, and memory footprint. A hybrid method by combining the transformation rules [24] with a hardware hardening technique has been proposed in [27]. A different set of transformation rules [25] have been suggested to modify high-level languages, where such transformation rules have been enhanced to protect flow-control structures and pointers [28].

Various data replication based approaches have been used in combination with different fault tolerance strategies targeting different execution scenarios. Signature analysis is one of the fault tolerance techniques combined with data replication [29]. Data and Control Flow Error Detection method uses a control variable to check for control-flow errors while duplicating instructions and registers to prevent data-flow errors [30].

There also exist studies that take advantage of the features specific to multicore processors and multithreaded applications. For example, modular redundancy-based fault tolerance replicates the annotated C/C++ code parts, where reliability is ensured by running them in different threads [31]. Another work has combined Duplication with Compare with Triple Modular Redundancy (TMR) [32]. The program is executed twice with the redundant data, and if the results do not match, TMR is triggered, the program is re-executed, and the majority voting is performed.

In addition, analytical models have been developed to calculate the criticality of the data to reduce the overheads

[1] A set of rules that modifies the code according to the data replication technique without affecting its functionality.

of variable duplication techniques. Critique Function *CF(v)* is defined to measure the criticality of the variable 'v'—it represents the sensitivity of the correctness of the program result to the error that occurs in variable v [9]. The PRASE analytical model examines the assembly code of the application and analyzes the probabilities for the error occurrence [33]. Finally, a method called Optimum Data Duplication has been proposed with the goal of protecting the most sensitive variables of the program revealed by PRASE.

Our work departs from the existing work as it examines the trade-off between reliability and performance/memory space overheads by conducting fault injection experiments with different levels of data replication techniques and study their impact on the reliability. Moreover, the effectiveness of data replication techniques in different faulty environments is also studied.

3 Data replication framework

In this work, we analyze the error propagation on different applications by performing fault injection. Then, by utilizing several protection mechanisms, the error propagation is slowed down. To "reshape" error propagation, our proposed approach employs data replication. A sample data replication in a multithreaded application code is illustrated in Fig. 1. Specifically, Fig. 1a shows the original multi-threaded application executed with four threads. The synchronization instructions are not included in the code for clarity of presentation. In the original code, all threads share variable *x* which is initialized by Thread 0. Suppose that an error occurs when computing the value of the *x*. Since this is a shared variable the error would propagate to all threads and their data would be corrupted. To enhance reliability, a redundant copy of *x* can be provided to all threads. In Fig. 1b, a protected version of the original code is given. In this version, all threads have their local

```
Thread 0        Thread 1         Thread 2      Thread 3
x = 1;          ...              ...           ...
a = x;          ...              ...           ...
...             c = x * 2;       d = x;        e = x;
b = a + 1;      ...              ...           ...
...             ...              ...           ...
```

(a)

```
Thread 0        Thread 1         Thread 2      Thread 3
x0 = 1;         x1 = 1;          x2 = 1;       x3 = 1;
a = x0;         ...              ...           ...
...             ...              ...           ...
b = a + 1;      c = x1 * 2;      d = x2;       e = x3;
...             ...              ...           ...
                ...              ...           ...
```

(b)

Fig. 1 Example data replication in a multithreaded code

copy of *x*, and perform their computation by using their local copy. Therefore, if an error occurs in one of the redundant copies of *x*, the error propagation stays within one thread (see Fig. 1b).

An important issue when employing data replication is the "trade-off" between the reliability enhancement achieved and performance/memory space overheads incurred. In particular, replicating all variables in a given program can provide maximum reliability; however, doing so can also increases the memory consumption substantially. On the other hand, if the program is executed without any data replication, this would lead to no memory overhead, but no reliability either. To strike a balance between these two extreme approaches, "partial data replication" can be used. In an application, several data may be more important to the user than the others, in terms of, for example, the correctness of the output. To give a more concrete example, in a face recognition application, the face part of the image can be considered more critical than the background. In partial data replication, only critical data are replicated as opposed to all data elements, and as a result, both memory consumption and performance overhead can be decreased. To find the optimal point under given reliability requirements and memory consumption constraints, we first need to understand and characterize the impact of data replication. In particular, understanding the impact of data replication techniques on error propagation is crucial for choosing the most suitable option. Also, in addition to error propagation, it is important to quantify the overheads introduced by data replication. We can consider the overhead from two angles—memory consumption and performance. Note that, to provide error correction at least 3 redundant copies are required for each data element to be protected. Consequently, if error correction is provided for all the data manipulated by an application program, the memory consumption would be tripled. Further, to ensure consistency, all the operations performed on the original data must also be performed on the copies. Therefore, the execution time and memory consumption can dramatically increase when employing data replication.

While using data replication under a certain memory consumption constraint/limit, different alternatives that can be considered. For example, consider a code segment that uses a 10×10 data array. If the user allows 10% extra memory usage for the purpose of reliability enhancement, which means 10 extra data elements (that are not part of the original program) would be created. However, a critical question at this point is *how to spend/use these 10 extra data elements to maximize reliability, i.e., how to optimize reliability under a memory consumption bound.* In our simple example, one option would be selecting 10 data elements (from a total of 100 data elements accessed by the program) and replicate each of them once. Another option

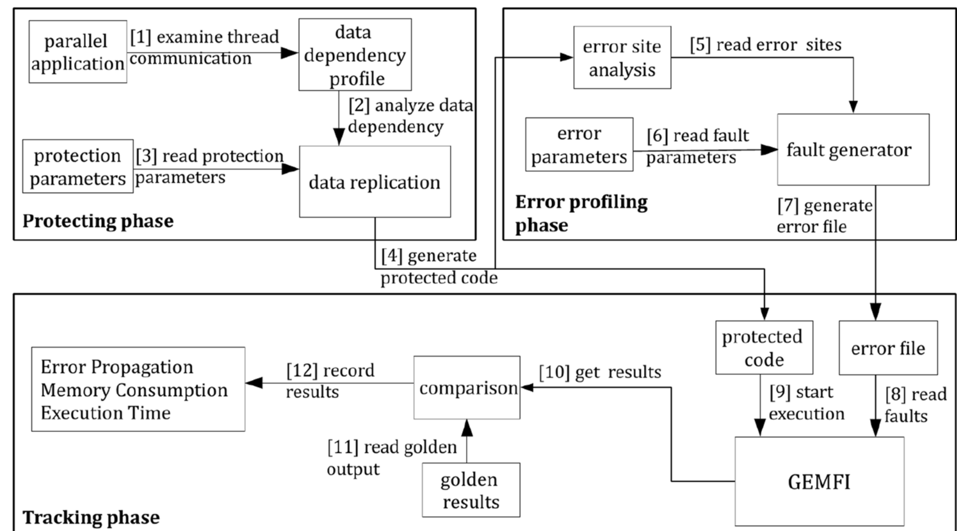


Fig. 2 Fault injection framework

would be selecting 5 data elements and replicating each of them twice. While these two options have the same extra memory consumption, depending on application code (control and data flows), their resulting reliability can be quite different. One can imagine that the options would quickly multiply if we consider multiple data structures (e.g., arrays).

Going deeper, it can be observed that identifying the data elements to replicate itself involves three sub-problems: (i) selecting the data structures (e.g., data arrays in an array-based application program) from which to replicate, (ii) deciding the data elements to replicate from the selected data structures, and finally (iii) for each data element to be replicated, deciding how many times to replicate it. As a result, given a program with multiple data structures, the potential “replication space” can be enormous. To examine the replication space in a systematic fashion, we define several protection parameters that capture different dimensions; percentage of replicated data, number of the replicas, and items that are replicated. In addition to these parameters, since we study the multithreaded applications, we also define thread related protection parameter: threads that are protected. It represents the threads that have redundant copies and help to decide the critical threads in terms of error propagation.

Besides the data replication alternatives, we also study the improvement of the reliability provided through data replication in different so called “faulty environments”. To capture/represent a faulty environment, we use the error parameters: erroneous data rate, bit span, number of threads that contain error, and thread in which errors are created.

To study different data replication alternatives and their impact on the different faulty environments, we have implemented error injection experiments. In the experiments, we use different error and protection parameters, and study the impact of these parameters on reliability, memory consumption, and execution time. In each experiment, we set the error parameters (erroneous data rate, bit span, number of threads that contain error, threads in which errors are created) and protection parameters (percentage of replicated data, number of the replicas, items that are replicated, threads that are protected), and the experiments performed measure error propagation, memory consumption and execution time for given code and parameter setting.

Figure 2 illustrates the workflow of our fault injection framework. Our framework consists of three phases, namely, protection phase, error profiling phase, and tracking phase. The first two phases are used to set the experimental configuration by considering two dimensions: *protection configuration* and *error configuration*. Protection parameters specify the degree/level of protection provided to the system via data replication and error parameters identify the state of the faulty environment. In the protection phase, firstly, we examine the application code and identify the data dependency and decomposition among threads. Since local copies are provided to the threads for the shared variables, it is important to decide which threads access which parts of data and determine data dependencies across threads. After performing the code analysis, the protection configuration is determined, and the application code is *modified* according to the protection parameters. For example, in the code given in Fig. 1a, the variable x is read by all threads. If an error occurs in this variable, this error affects all threads. Error isolation is achieved by *replicating* variable x across all threads. Note that the local copies of x are provided to threads according to the protection parameters.

In the error profiling phase, firstly, the protected code generated by the protection phase is analyzed. In our study, error propagation is discussed in terms of data structures. In other words, the error is introduced to the system to alter the marked data, and the error propagation is examined in terms of the application data. Therefore, dynamic instructions related to the variables/data structures are determined for the fault injection. The dynamic instruction refers to the assembly code generated at runtime for the high-level language. After determining the dynamic instructions and error parameters, error files are created using them.

Finally, in the tracking phase, error injection experiments are performed, and metrics including error propagation, memory consumption, and execution time are collected for the given protection and error parameters.

The errors are determined by comparing the error-injected run results with the golden results. In this context, the golden results refer to the results obtained from the error-free execution. As an example, assume that an error is injected to the first line of the code given in Fig. 3. This error will propagate during runtime (erroneous registers are shown in red and italic in Fig. 3b). In the tracking phase, the *st* (memory store operation) commands obtained from the error-free execution (line 4 in Fig. 3a) and erroneous execution (line 4 in Fig. 3b) are compared and the erroneous data (if any) are detected.

The error is introduced to the system as a single-bit flip. Although multi-bit errors have been seen frequently in recent years, studies show that single-bit errors cause SDC as much as double and multi-bit errors, and in some cases even more [34, 35]. Since our work is examined data errors, the errors are injected to the system to corrupt the data values. The errors that will cause the program to terminate are outside the scope of this study.

```
(1) $reg1 = ld 0($reg2);
(2) $reg3 = ld 4($reg2);
(3) add $reg1, $reg1, $reg3;
(4) 4($reg2) = st $reg1;
```

(a)

```
(1) $reg1 = ld 0($reg2);
(2) $reg3 = ld 4($reg2);
(3) add $reg1, $reg1, $reg3;
(4) 4($reg2) = st $reg1;
```

(b)

Fig. 3 (a) An example code for the golden results, (b) its faulty version for tracking phase where fault injected instructions are italicized

We use a relatively high error rates in our empirical study. Since the applications in our experiments have short execution times, employing low error rates would end up injecting only a few errors. This accelerated error injection allows us observe the effect of errors more clearly and let us have an idea about what effect the errors would have and how effective our approach would be in more realistic scenarios with long application execution latencies. Note that employing accelerated error injection is a frequently-used strategy in the literature [36–39].

4 Experimental evaluation

4.1 Applications

We utilize different applications to test the effectiveness of our proposed approach, which are block-wise matrix multiplication, one-dimensional heat diffusion, and two-dimensional Jacobi applications. The algorithms in the experimental study are implemented based on different parallel programming patterns [40]. Program Structuring Patterns are models that provide templates to the programmer for writing parallel codes. They define models for different aspects of parallelism, such as data decomposition and threads communication. Matrix multiplication application works with the Single Program Multiple Data (SPMD) pattern, whereas heat diffusion and two-dimensional Jacobi applications are modeled with the Master/Worker pattern.

Block-wise Matrix Multiplication The block-wise matrix multiplication algorithm divides the input and output matrices into blocks called sub-matrices and performs matrix multiplication using the sub-matrices. More specifically, in the $C = A \times B$ multiplication, matrices A, B and C are divided into blocks; so the matrix multiplication equation can be rewritten as:

$$C^{ij} = \sum_{k} A^{ik} \times B^{kj}. \quad (1)$$

An example of the block-wise matrix multiplication is illustrated in Fig. 4. The matrix C is divided into sub-matrices, and each sub-matrix is assigned to a thread for computation. The first row of matrix A and the the first

A1	A2	A3
A4	A5	A6
A7	A8	A9

B1	B2	B3
B4	B5	B6
B7	B8	B9

C1	C2	C3
C4	C5	C6
C7	C8	C9

Fig. 4 Block-wise matrix multiplication computation for one sub-matrix

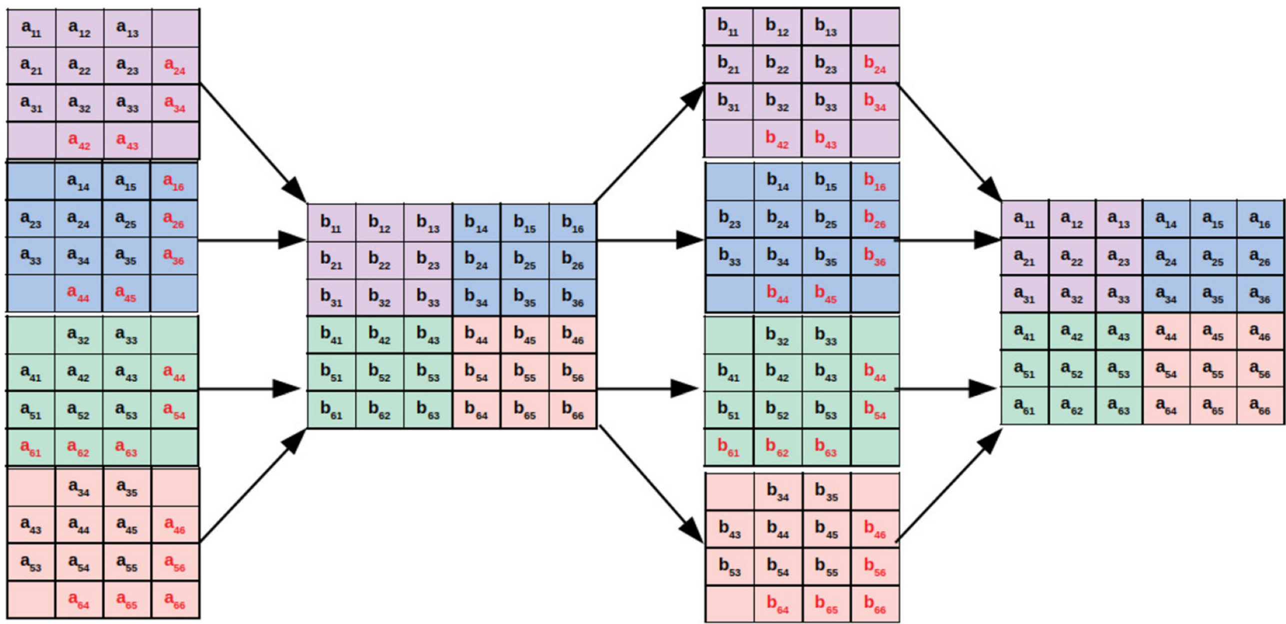


Fig. 5 Data decomposition and replication for 2D Jacobi computation

column of matrix B is used to compute C1 block. Note that this block-based code is one of the most frequently used implementations of matrix multiplication. The computation of each block of the matrix C is independent and can be performed concurrently with others. When the application is modified to employ data replication, each thread gets a copy of blocks A and B that it will use, and performs computation using its local data (as a result, there is no communication between threads via shared variables).

Heat Diffusion The one-dimensional heat diffusion problem models heat transmission in a pipe. In the beginning, the heat of the pipe is stable and fixed. At time 0, both ends of the pipe are set to different temperatures, and their heat remains fixed during the execution. The temperatures change in the rest of the pipe and are computed over time. At each iteration $x[i]$ (ith element of the pipe) is computed by using $x[i-1]$ and $x[i+1]$. Mathematically, the algorithm solves a *one-dimensional differential equation* representing the heat diffusion:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}. \tag{2}$$

In the parallelized version used in our evaluations, the pipe is divided into chunks, and these chunks are assigned to the different threads. The algorithm requires communication among threads since boundary data (data located boundary at the chunks) are shared across multiple threads. When modifying the heat diffusion algorithm to accommodate data replication, the boundary data are replicated in appropriate threads. Threads then compute their chunks by using their private copies. At each iteration, the heat of an array item is computed by using its right and left neighbors; so, threads require elements from other (neighboring) threads to compute the boundary data, and the updated boundary data are received from the other threads.

Two-Dimensional Jacobi Iteration This application performs Jacobi computation over two-dimensional data with a 5-point stencil pattern. It iteratively updates array elements using a fixed pattern. Jacobi application simply takes the average of four neighboring cells:

$$\begin{aligned} array^{t}_{(i,j)} = 0.2 * (&array^{t-1}_{(i,j)} + array^{t-1}_{(i-1,j)} \\ &+ array^{t-1}_{(i+1,j)} + array^{t-1}_{(i,j-1)} \\ &+ array^{t-1}_{(i,j+1)}) \end{aligned} \tag{3}$$

In our parallel implementation, Jacobi computations are performed as two steps, and two arrays A and B are utilized for computations. In the first step, threads compute array B by using array A and send it to the main thread. Following that, the main thread provides a local copy of array B to each thread, and then threads compute (the new values of) array A using array B. An example data decomposition and replication is given in Fig. 5. Each color represents a thread and red items represent the replicated data. Similar to the heat diffusion problem, when the different parts of matrices are assigned to the thread for computation, the boundary data is used by multiple threads. Therefore, when replicating the data, the boundary data is replicated; and threads get their private copy for these elements.

In the experimental study, we consider two different versions for two-dimensional Jacobi iteration and analyze

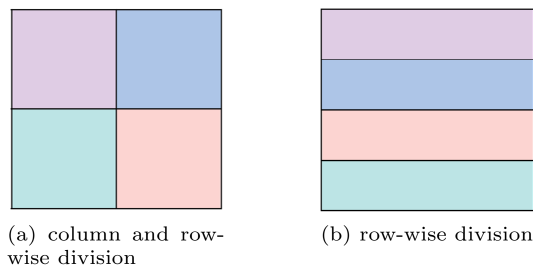


Fig. 6 Data distribution of two-dimensional Jacobi iteration

error propagation and the effect of the data replication on them. The difference between the implementations is the distribution of the arrays across the threads. In the first implementation, a two-dimensional distribution is considered. More specifically, arrays A and B are divided into sub-arrays both column and row-wise and then each sub-array is assigned to a different thread. In the second implementation on the other hand, a row-wise distribution is considered. The array distributions of two implementations are given in Fig. 6 where different colors represent different threads.

4.2 Experimental settings

Our fault injection experiments have two dimensions. The first is the *error dimension*. We define different error parameters to specify the errors injected for the experiments – erroneous data rate, bit span (the number of bits affected from bit flip operations), number of threads that contain error, and threads in which errors are created. By using these parameters, errors in different locations and rates are injected into the application execution, and the resulting error propagation is examined. The second is *protection dimension*. Protection parameters are defined to implement different levels of protection – percentage of replicated data, number of replicas, items that are replicated, and threads that are protected. By using these parameters, the degree/level of protection that would be provided to the application is determined and the effects of different protection alternatives on error propagation and cost, in terms of execution latency and memory consumption, are examined.

The first step of our experimental study is to determine the values of protection parameters (and as a result come up with different protected versions of a given application program). Then, by determining the different values for the error parameters, the fault injection space is determined and the errors are injected into the different alternatives of the application program according to the fault injection scenarios with different error properties. Finally, by examining error propagation, execution time and memory consumption, the relationship between error reduction (fault tolerance provided by data replication) and its cost (execution time and memory consumption) is studied.

Table 1 Default error parameters used in the fault injection experiments

Error parameter	Value
Erroneous data rate	0.1%
Bit span	Single
Number of threads that contain errors	T/4
Threads in which errors are created	Randomly chosen threads

Table 2 Default protection parameters used in the fault injection experiments

Protection parameter	Value
% of replicated data	50%
Number of the replicas	T
Items that are replicated	Randomly chosen data
Threads that are protected	Randomly chosen threads

The default values of the error and protection parameters are given in Tables 1 and 2, respectively. To quantify the effect of each parameter separately, a sensitivity analysis is carried out where we change the value of only one parameter in each experimental configuration. The protection and error configurations used in the experiments are given in Tables 3 and 4, respectively. Error configurations in Table 3 are generated by changing one parameter of Error Configuration 1. In these tables, T refers to the number of threads used by to execute an application.

In Table 4, in cases where the 'number of the replicas column' is equal to T, a local copy is created for each thread. Therefore, in configurations where the number of the replicas is equal to T, the 'threads that are protected' parameter should be ignored. In cases where the 'number of replicas' column is $T/2$, half of the threads have their own local copies, while the other half use the shared variable. In Table 4, 'items that are replicated' and 'threads in which errors are created' parameters are not shown; instead, in the experiments, randomly chosen items and threads are utilized to analyze their effects. When creating the protection configurations, we consider the application behavior and only replicate the data accessed by different threads. For instance, when the 'percentage of replicated data' is equal to 100% and 'number of replicas' is equal to the thread count for block-wise matrix multiplication, the threads get redundant copies of the blocks *they use* in computation instead of the whole matrices.

To compare the different protection alternatives fairly, the error is injected into the protected and unprotected

Table 3 Error configurations used in the experimental study

Error configuration #	Erroneous data rate	Bit span	Number of threads contain errors
1	0.1%	Single	T/4
2	0.1%	Single	**T/2**
3	0.1%	Single	**1**
4	0.1%	**Double**	T/4
5	**5%**	Single	T/4
6	**20%**	Single	T/4

Configurations are generated by changing one parameter, where the changed parameter is given in bold font

Table 4 Protection configurations used in the experimental study

Protection configuration #	% of Replicated data	Number of replicas
1	0	–
2	100	T
3	75	T
4	75	T/2
5	50	T
6	50	T/2
7	25	T
8	25	T/2

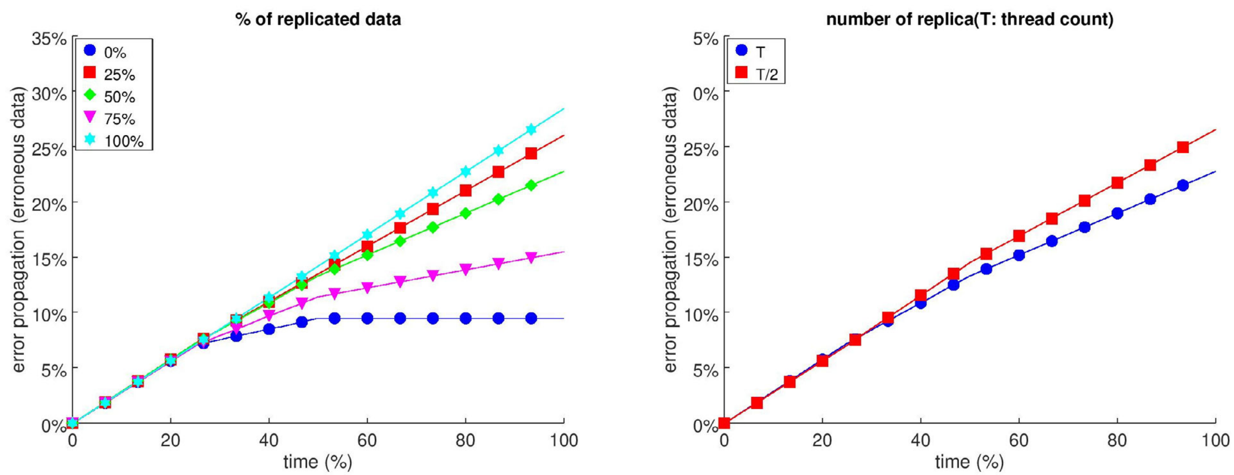
versions of an application by using the same error configuration. Since the 'bit span' and 'number of threads that contain error' parameters represent the number of bits affected by the error and the number of threads executing in the application, respectively, they can be the same for all versions, but an analysis is needed for the 'erroneous data rate' and 'threads in which errors are created' parameters. The erroneous data rate is used to calculate the number of data elements into which errors are injected. To compare the unprotected and protected code versions fairly, the number of incorrect data elements should be calculated according to the memory consumption of the different versions. As an example, for the block-wise matrix multiplication application, the memory consumption in the unprotected version equals to $(NxN) * 2$ where 'N' is the matrix size. On the other hand, for the full protected version (when A and B matrices are replicated in all threads), the memory consumption would be increased. A low error injection rate would cause only a few errors, since the running times of the experiments are low. Therefore, we consider relatively high error injection rates (ranging from 0.1% to 20%) to observe the impact of the errors more prominently.

The *'number of threads that contains error'* is a parameter that indicates the number of threads with errors. This parameter is utilized when each thread has private copy, where errors are injected to thread's memory space. On the other hand, when threads use shared variables, an analysis is needed to understand which thread accesses which part of the data. After such analysis is performed, the shared data used by selected threads are injected with errors.

To evaluate performance, reliability, and memory consumption of the different protection and error configuration degrees, we utilize a multicore system with the $x86$ full system mode. The fault injection experiments are performed by setting the number of cores equal to the thread number of threads. Each thread is assigned to a different core at the beginning of the execution; and this mapping is remained fixed during the execution.

4.3 Results and discussion

In this section, the results of the fault injection experiments are presented and discussed. GEMFI [41] is used as our error injecting tool. It is an error injection tool based on the Gem5 [42] simulator, and can be used to inject errors into the dynamic instructions of an application program. The error propagation curves for different applications are shown in Figs. 7, 9, 10 and 11. In collecting these results, we have considered the main portions of the application code, and omitted the variable initialization parts. The x-axis represents the execution time, and the y-axis represents the erroneous data rate, which corresponds to the fraction of output data that have been calculated incorrectly. Since we add *copy* and *comparison* instructions to provide protection, the execution time differs in created versions. Therefore, the execution time axis is shown in percentage to compare different executions.



(a) Error propagation for different '*% of replicated data*' values. For the experiments, replicas are provided to all threads.

(b) Error propagation for different '*number of replica*' values. For the experiments, 25% of the data are replicated.

Fig. 7 Error propagation results for block-wise matrix multiplication. Different protection configurations are presented in the figures

The plots in Fig. 7 give the error propagation results for the block-wise matrix multiplication. Figure 7a shows the impact of the "percentage of replicated data" parameter. In the experiments, we assign 0%, 25%, 50%, 75% and 100% values to the "percentage of replicated data" and provide all threads with a replica of the chosen data. To compare the different protection parameters fairly, we have utilized a fixed error setting (error configuration 2 from Table 3). Since the blocked matrix multiplication is an application where the threads work independently, providing private copies to the threads prevents the error from spreading from one thread to the others. As can be seen from Fig. 7a, when the percentage of replicated data parameter increases, more data are chosen to replicate, which in turn reduces the error propagation between threads, resulting in a decrease in the total error propagation. As can be observed from the results, the percentage decrease in error propagation is not proportional to the percentage increase in data replication. More specifically, if there is no replicated data, the error rate is 28%. When the percentage of replicated data is increased to 25%, the error rate decreases to 26%. Also, when the percentage of replicated data increases from 50% to 75%, a significant decrease is observed in the error rate – from 23 to 16%.

Figure 7b shows the impact of the "number of replicas" parameter. In the given experiments, firstly a replica is provided to all threads for the chosen data (in these experiments the percentage of replicated data is set to 25%), and then a replica is provided to only half of the threads whereas the remaining threads use the shared variables. Since the threads are tightly coupled, the errors propagate in a very short time for both cases. Therefore, the results plotted in Fig. 7b indicate that the number of the replicas has low impact on preventing the error propagation among tightly coupled threads.

Besides the 'percentage of replicated data' and 'number of replicas parameters', we also performed fault injection experiments to understand the impact of "items that are replicated" and "threads that are protected" parameters. The results (not presented due to space concerns) have revealed that, when we select different elements from the matrices (or different threads to provide redundant copies for), the error propagation remains same. The reason for this is that the result is equally sensitive to the error that occurs in different threads and variables. In other words, the work distribution between threads is similar, all threads calculate a block of the result matrix, and the variables are equally effective in calculating the result. The experimental configuration used in the fault injection experiments spans a 6 x 8 evaluation space that consists of 6 error configurations and 8 protection configurations. Similar results are observed with different configurations as well, and thus, only a certain parts of the results are presented in the paper.

In order to examine the effect of the number of threads on the error propagation, the block-wise matrix multiplication application is executed with 16 threads and 4 threads, and the error propagation curves for these versions are compared against one another. The error propagation curves for different thread counts are plotted in Fig. 8. In the experiments, the same error and protection configurations are used. While the error propagation error rate is 0.15 when the number of threads is 16, and it decreased to 0.13, when the number of threads is 4. Although the error propagation decreases slightly, there is a significant difference in the dynamic behavior of the error. Specifically, the error propagation is not observed in the first half of the program, as the error does not propagate until the faulty

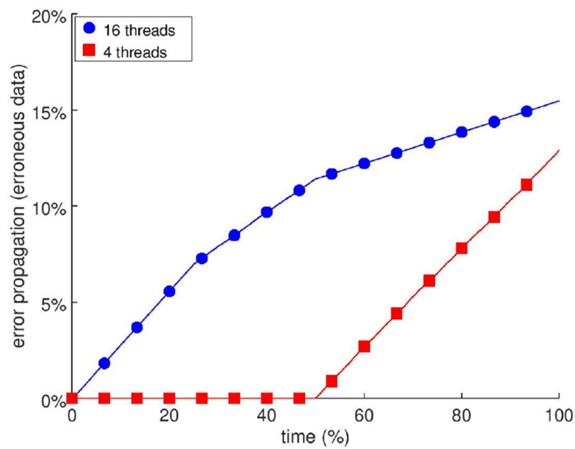


Fig. 8 Error propagation results for block-wise matrix multiplication with different '*thread counts*'

Table 5 Memory consumption, execution time and error rate values for different protection configurations

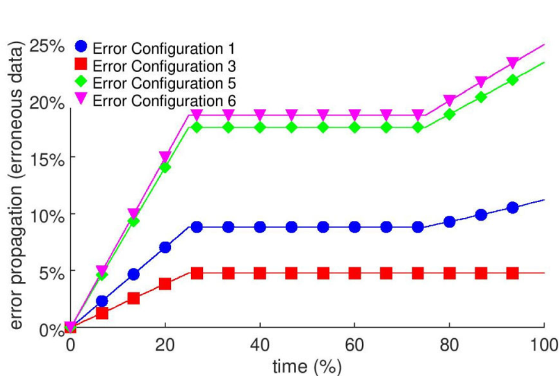
Protection configuration #	Memory consumption (Normalized)	Execution time (Normalized)	Error rate (%)
1	1	1	0.284
2	9	1.072	0.095
3	7	1.036	0.156
4	4.5	1.034	0.193
5	5	1.034	0.195
6	3.25	1.030	0.25
7	3	1.006	0.26
8	2.31	1.007	0.27

The memory consumption and execution time values are normalized to *the base case (Protection Configuration 1)* which has no data replication

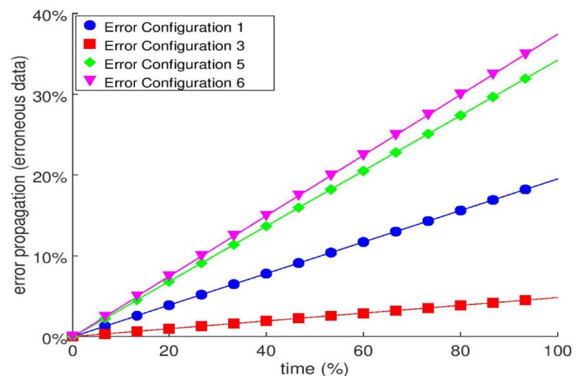
thread is executed. Then, once the faulty thread is executed, the error starts to propagate and spreads throughout the thread.

In addition to the error propagation, we have also examined the impact of the protection setting on the performance and memory consumption. Memory consumption, execution time and erroneous data rate on the output under different protection settings are given in Table 5. These values are obtained from the fault injection experiments when using Error Configuration 1 from Table 3, and normalized by using the values measured in experiments with no protection (Protection Configuration 1).

The graphs in Fig. 9 plot the error propagation curves obtained from the fault injection experiments performed with different error settings for the block-wise matrix multiplication. Error configurations 2 and 4 are not shown in the graph. Error propagation graphs for error configurations 4 and 1 are overlapping, and error propagation graphs for error configuration 2 with different protection configurations are given in Fig. 7a. The full protection is utilized in the results shown in Fig. 9a, that is, a local copy is provided to each thread for all data. No data replication is utilized in Fig. 9b. The details of the error configurations tested are given in Table 3. When the error is injected into a single thread, if no data replication is used, the error propagates to 0.1 of the data elements (see Fig. 9b), while in the case of full protection, this rate decreases to 0.05 (see Fig. 9a). On the other hand, when Error Configuration 6 is used as the error setting, a significant decrease in error propagation is observed. When data replication is not employed, the error propagates to 0.75 of the data, whereas the rate of erroneous data decreased to 0.25 when all data are replicated. With the data replication technique, private copies are provided to the threads; therefore, it prevents an error passing from one thread to another. In block-wise

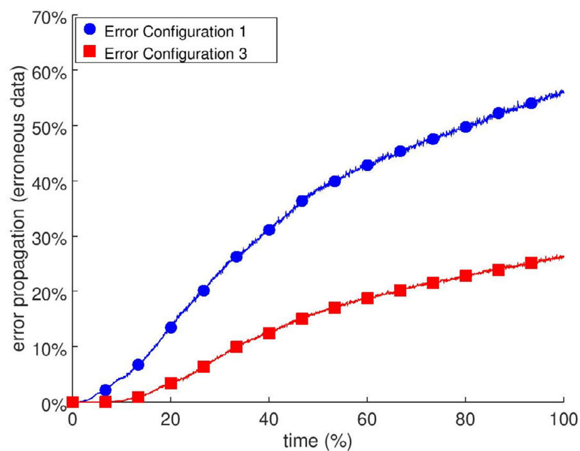


(a) Error propagation results for full data replication (protection configuration 2).

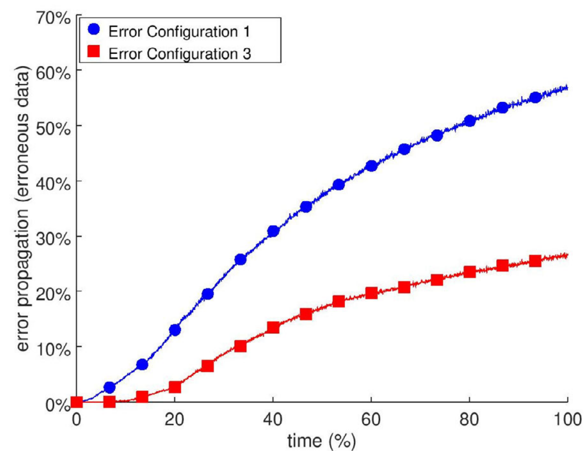


(b) Error propagation results for no protection (protection configuration 1)

Fig. 9 Error propagation results for block-wise matrix multiplication with different error configurations



(a) Error propagation results for full protection (protection configuration 2).



(b) Error propagation results for no protection (protection configuration 1).

Fig. 10 Error propagation results for heat diffusion with different error configurations

matrix multiplication, threads share the matrices. Consequently, if an error occurs in one of the elements, it is read by all threads. However, when we utilize the data replication, the threads are isolated from each other, and the error affects only one thread's results. Moreover, in Figure 9a, the error propagation is observed only at certain interval during the execution. In our fault injection experiments, we inject the error to selected threads. Since the block-wise matrix multiplication application does not require communication between threads, error propagation is observed only when error-injected threads are running. The error propagation remains constant during intervals where error-free threads are running.

In cases where the error is injected into more threads or more data, when the application uses shared variables, the error propagation among the threads becomes faster, and when a private copy is provided to the threads, these errors are isolated in the thread, and as a result, there is a decrease in the total error propagation. On the other hand, although giving a private copy to each thread reduces the error when the injected error is low, the protection does not provide a significant gain since the error propagation between the threads is already low. The costs of different protection settings are given in Table 5. While utilizing a costly protection setting is not efficient in cases where the reduction in error propagation is low, it can be an acceptable cost in cases where the reduction in error propagation is significant.

The second application used in the fault injection experiments is the one-dimensional heat diffusion application, and the corresponding experimental results are given in Fig. 10. In these experiments, two different error configurations are considered, and there is almost no difference in the error propagation curves between full protection (Fig. 10a) and no protection (Fig. 10b). The reason for this result can be explained by the underlying communication pattern of this application. In this application, the threads are tightly coupled. Since there is no communication between the threads in block-wise matrix multiplication, when local copies are assigned to the threads, the error that occurs in the thread could be isolated, whereas in the applications where data are heavily shared between the threads, the error in the thread is propagated to all threads through the shared variables. Another observation from these experiments is that, when the number of threads into which the error is injected is increased, a significant increase in the error propagation speed is observed.

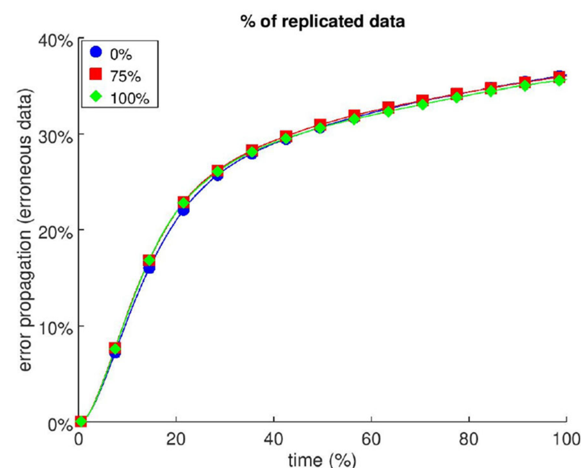


Fig. 11 Error propagation result for two-dimensional Jacobi for different "*% of replicated data*" values

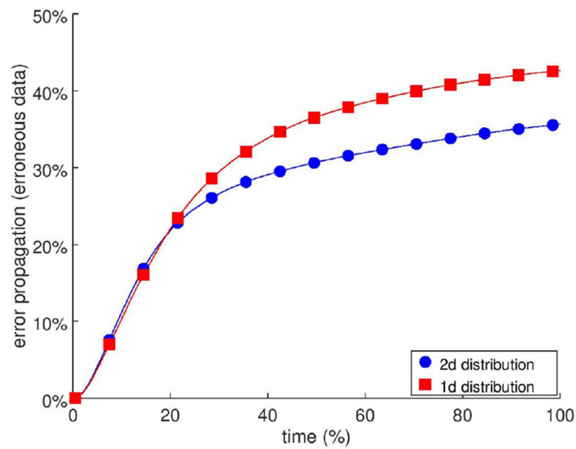


Fig. 12 Error propagation result for 1d distribution (row-wise) and 2d distribution (row and column-wise) in 2-D Jacobi

Figures 11 and 12 show the error propagation results for the two-dimensional Jacobi application for two-dimensional (both row and column-wise) and one-dimensional (row-wise) distribution respectively. As can be seen from Fig. 11, the error propagation behavior of the heat diffusion is also observed in the Jacobi application, where partial data replication is not successful in preventing error propagation. In both applications, threads are tightly coupled and data exchange is required between threads at each iteration. While desired reliability can be achieved with lower cost for the applications where threads mostly execute independently, costly data replication techniques should be utilized to achieve higher reliability for such applications. Our approach allows the designers to perform trade-off (*what-if*) experiments to understand the cost of the reliability. In addition to the error and protection parameters, error injection experiments with different implementations of the two-dimensional Jacobi iteration are conducted, and the resulting error propagation is analyzed in different implementations.

Figure 12 plots the error propagation in different Jacobi versions implemented with different data distribution methods (in the experiments, Error Configuration 1 and Protection Configuration 2 are utilized). As can be seen from the figure, the change in the Jacobi implementation affects the error propagation. Even though the same thread count, error parameters, and protection parameters are used in both the versions, an increase in error rate is observed when a row-wise distribution is used as the data distribution. One potential reason behind this result is the modulation in thread communication patterns when data distribution is changed. That is, when the communication patterns across threads changes, the error propagation between threads due to data communication also changes. These results also reveal that different parallel versions can be employed to slow down the error rate by reshaping the error propagation curve.

5 Conclusion

In this paper, we study the impact of selective data replication on error propagation, memory consumption, and performance of multithreaded applications executing on multicore environments. Data replication is implemented by providing a private copy to each thread. Thus, error propagation caused by the usage of the shared variables between threads can be prevented. To perform data replication, there are different dimensions to be considered, such as selecting the data structures and elements to replicate as well as identifying the threads to provide a private copy for. In our work, we propose several parameters to identify the degree/level of the protection.

This work also investigates the impact of data replication in different faulty environments. More specifically, several error parameters such as error rate, bit span, and erroneous thread number are defined to create fault injection experiments. The results of our fault injecting experiments indicate that thread communication has a significant impact on the effectiveness of the protection provided by data replication. Among the applications used in our experiments, data replication reduces the error propagation in applications where threads do not communicate much, while there is no significant change in error propagation in applications where threads have data dependencies among them. In addition, our experiments reveal that, in highly faulty environments the error propagation can be very high. Consequently, in such environments, data replication can being quite effective in reducing error propagation significantly, if one is willing the pay the cost of execution time increase and memory footprint expansion. The acceptable cost in execution time and memory consumption overhead can change based on the user-specified requirements and acceptable error rate and also system constraints. Our proposed approach allows the users to perform trade-off experiments to understand the cost of the reliability and decide the acceptable cost and reliability.

In this study, besides slowing down the error propagation by providing data replication at different levels, the impact of the changes in the implementation on the error propagation is also examined. Experiments performed by changing the data distribution reveal that being a bit cagey about which implementation to use can slow down error propagation significantly. Slowing down the error propagation with different implementation and code optimizations offers an approach that can be exploited to ensure reliability at a low cost.

A possible extension of this work is to slow down the error propagation by parallelizing the algorithm in various ways, which might provide error resiliency with a low cost. Our study reveals that different parallel implementations of the algorithm (by changing the data decomposition and the thread count) can provide different degrees of reliability. Hybrid methods can be designed by combining various protected versions with online error detection methods. The hybrid methods are utilized to provide reliability aware computation where the protection techniques can be tuned during runtime according to the application and system requirements.

Funding This research was supported by The Scientific and Technological Research Council of Turkey (TUBITAK) with a research grant (Project Number: 118E715).

Data availability The datasets generated and/or analysed during the current study are available from the corresponding author on reasonable request.

References

1. Baumann, R.C.: Radiation-induced soft errors in advanced semiconductor technologies. IEEE Transactions on Device and Materials Reliability. **5**(3), 305–316 (2005)
2. Mukherjee, S.S., Kontz, M., Reinhardt, S.K.: Detailed design and evaluation of redundant multi-threading alternatives. In: Proceedings 29th Annual International Symposium on Computer Architecture, pp. 99–110 (2002)
3. Rotta, R., Ferreira, R.S., Nolte, J.: Real-time dynamic hardware reconfiguration for processors with redundant functional units. In: 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC), pp. 154–155 (2020)
4. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: SWIFT: Software implemented fault tolerance. In: International Symposium on Code Generation and Optimization, pp. 243–254 (2005)
5. Mahmoud, A., Hari, S.KS., Sullivan, M.B., Tsai, T., Keckler, S.W.: Optimizing software-directed instruction replication for gpu error detection. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 842–853 (2018)
6. Vallero, A., Savino, A., Chatzidimitriou, A., Kaliorakis, M., Kooli, M., Riera, M., et al.: SyRA: Early system reliability analysis for cross-layer soft errors resilience in memory arrays of microprocessor systems. IEEE Transact. Comput. **68**(5), 765–783 (2018)
7. Cheng, E., Mirkhani, S., Szafaryn, L.G., Cher, C.Y., Cho, H., Skadron, K., et al.: CLEAR: cross-layer exploration for architecting resilience-combining hardware and software techniques to tolerate soft errors in processor cores. In: Proceedings of the 53rd Annual Design Automation Conference, pp. 1–6 (2016)
8. Borodin, D., Juurlink, B.H.: Protective redundancy overhead reduction using instruction vulnerability factor. In: Proceedings of the 7th ACM International Conference on Computing Frontiers, pp. 319–326 (2010)
9. Benso, A., Di Carlo, S., Di Natale, G., Prinetto, P., Tagliaferri, L.: Data criticality estimation in software applications. In: International Test Conference, pp. 802–810 (2003)
10. Mukherjee, S.S., Weaver, C.T., Emer, J., Reinhardt, S.K., Austin, T.: Measuring architectural vulnerability factors. IEEE Micro. **23**(6), 70–75 (2003)
11. Zhang, W.: Computing cache vulnerability to transient errors and its implication. In: 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05), pp. 427–435 (2005)
12. Yan, J., Zhang, W.: Compiler-guided register reliability improvement against soft errors. In: Proceedings of the 5th ACM International Conference on Embedded Software, pp. 203–209 (2005)
13. Sridharan V, Kaeli DR. Eliminating microarchitectural dependency from architectural vulnerability. In: 2009 IEEE 15th International Symposium on High Performance Computer Architecture; 2009. p. 117–128
14. Oz, I., Topcuoglu, H.R., Kandemir, M., Tosun, O.: Thread vulnerability in parallel applications. J. Parallel Distribut Comput. **72**(10), 1171–1185 (2012)
15. Utrera, G., Gil, M., Martorell, X.: Analyzing data-error propagation effects in high-performance computing. In: 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 418–421 (2016)
16. Guo, L., Li, D.: Moard: Modeling application resilience to transient faults on data objects. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 878–889 (2019)
17. Guo, L., Li, D., Laguna, I., Schulz, M.: Fliptracker: Understanding natural error resilience in hpc applications. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 94–107 (2018)
18. Li, Z., Menon, H., Maljovec, D., Livnat, Y., Liu, S., Mohror, K., et al.: Spotsdc: Revealing the silent data corruption propagation in high-performance computing systems. IEEE Transac. Visual. Comput. Graphics. **27**(10), 3938–3952 (2020)
19. Gu, J., Zheng, W., Zhuang, Y., Zhang, Q.: Vulnerability analysis of instructions for SDC-causing error detection. IEEE Access. **7**, 168885–168898 (2019)
20. Li, G., Pattabiraman, K., Hari, S.K.S., Sullivan, M., Tsai, T.: Modeling soft-error propagation in programs. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 27–38 (2018)
21. Li, G., Pattabiraman, K.: Modeling input-dependent error propagation in programs. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 279–290 (2018)
22. Anwer, A.R., Li, G., Pattabiraman, K., Sullivan, M., Tsai, T., Hari, S.K.S.: Gpu-trident: efficient modeling of error propagation in gpu programs. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–15 (2020)
23. Restrepo-Calle, F., Martínez-Álvarez, A., Cuenca-Asensi, S., Jimeno-Morenilla, A.: Selective swift-r. J. Electron. Testing. **29**(6), 825–838 (2013)
24. Chielle, E., Kastensmidt, F.L., Cuenca-Asensi, S.: Overhead reduction in data-flow software-based fault tolerance techniques. In: FPGAs and Parallel Architectures for Aerospace Applications. Springer, pp. 279–291 (2016)
25. Rebaudengo, M., Reorda, M.S., Torchiano, M., Violante, M.: Soft-error detection through software fault-tolerance techniques. In: Proceedings 1999 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (EFT'99), pp. 210–218 (1999)
26. Chielle, E., Kastensmidt, F.L., Cuenca-Asensi, S.: Tuning software-based fault-tolerance techniques for power optimization. In: 2014 24th International Workshop on Power and Timing

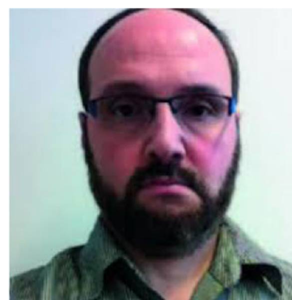
Modeling, Optimization and Simulation (PATMOS), pp. 1–7 (2014)
27. Chielle, E., Du, B., Kastensmidt, F.L., Cuenca-Asensi, S., Sterpone, L., Reorda, M.S.: Hybrid soft error mitigation techniques for COTS processor-based systems. In: 2016 17th Latin-American Test Symposium (LATS), pp. 99–104 (2016)
28. Tsai, T.Y., Huang, J.L.: Source code transformation for software-based on-line error detection. In: 2017 IEEE Conference on Dependable and Secure Computing, pp. 305–309 (2017)
29. Liu, J., Kurt, M.C., Agrawal, G.: A practical approach for handling soft errors in iterative applications. In: 2015 IEEE International Conference on Cluster Computing, pp. 158–161 (2015)
30. Thati, V.B., Vankeirsbilck, J., Pissoort, D., Boydens, J.: Hybrid technique for soft error detection in dependable embedded software: a first experiment. In: 2019 IEEE XXVIII International Scientific Conference Electronics (ET), pp. 1–4 (2019)
31. Serrano-Cases, A., Restrepo-Calle, F., Cuenca-Asensi, S., Martínez-Álvarez, A.: Softerror mitigation for multi-core processors based on thread replication. In: 2019 IEEE Latin American Test Symposium (LATS), pp. 1–5 (2019)
32. Quinn, H., Baker, Z., Fairbanks, T., Tripp, J.L., Duran, G.: Robust duplication with comparison methods in microcontrollers. IEEE Transact. Nuclear Sci. **64**(1), 338–345 (2016)
33. Xu, J., Tan, Q., Shen, R.: A novel optimum data duplication approach for soft error detection. In: 2008 15th Asia-Pacific Software Engineering Conference, pp. 161–168 (2008)
34. Sangchoolie, B., Pattabiraman, K., Karlsson, J.: One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors. In: 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 97–108 (2017)
35. Lu, Q., Farahani, M., Wei, J., Thomas, A., Pattabiraman, K.: Llfi: An intermediate code-level fault injection tool for hardware faults. In: 2015 IEEE International Conference on Software Quality, Reliability and Security, pp. 11–16 (2015)
36. Ferreira, R.R., Da Rolt, J., Nazar, G.L., Moreira, A.F., Carro, L.: Adaptive low-power architecture for high-performance and reliable embedded computing. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 538–549 (2014)
37. Hu, J., Wang, S., Ziavras, S.G.: On the exploitation of narrow-width values for improving register file reliability. IEEE Transact. Very Large Scale Integrat. (VLSI) Sys. **17**(7), 953–963 (2009)
38. Subasi, O., Arias, J., Unsal, O., Labarta, J., Cristal, A.: Nanocheckpoints: A task-based asynchronous dataflow framework for efficient and scalable checkpoint/restart. In: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 99–102 (2015)
39. Ashraf, R.A., Gioiosa, R., Kestor, G., DeMara, R.F.: Exploring the effect of compiler optimizations on the reliability of HPC applications. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1274–1283 (2017)
40. Mattson, T.G., Sanders, B., Massingill, B.: Patterns for parallel programming. Pearson Education, London (2004)
41. Parasyris, K., Tziantzoulis, G., Antonopoulos, C.D., Bellas, N.: GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 622–629 (2014)
42. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., et al.: The gem5 simulator. SIGARCH Comput Archit News. **39**(2), 1–7 (2011)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Zuhal Ozturk is a PhD student in Computer Engineering Department at Marmara University, Turkey. She obtained his BS degree and MS degree from the same department in 2014 and 2017. Her research interests include fault tolerant computing, reliability in multicore architectures and parallel computing.

Haluk Rahmi Topcuoglu is a Professor in Computer Engineering Department at Marmara University, Turkey. He obtained his Ph.D. degree from Syracuse University, Syracuse, NY in 1999. His research interests mainly include software-based hardware reliability, task scheduling and mapping for multicore architectures, scheduling in cloud/fog computing and dynamic optimization problems. He is a member of the IEEE, the IEEE Computer Society and the ACM.



Mahmut Taylan Kandemir (IEEE Fellow) is a professor in the Computer Science and Engineering Department at the Pennsylvania State University, and co-leads the Microsystems Design Lab. He obtained his PhD from Syracuse University, Syracuse, New York, in 1999. His current research interests include compilers, computer architectures, high-performance storage systems, and drug discovery.