Received: 30 October 2017 | Revised: 27 September 2018 | Accepted: 26 October 2018
DOI: 10.1002/cpe.5085

WILEY

RESEARCH ARTICLE

A user-assisted thread-level vulnerability assessment tool

Isil Oz[1] | Haluk Rahmi Topcuoglu[2] | Oguz Tosun[3]

[1]Computer Engineering Department, Izmir Institute of Technology, Izmir, Turkey
[2]Computer Engineering Department, Marmara University, Istanbul, Turkey
[3]Computer Engineering Department, Bogazici University, Istanbul, Turkey

Correspondence
Isil Oz, Computer Engineering Department, Izmir Institute of Technology, Izmir, Turkey.
Email: isiloz@iyte.edu.tr

Summary

The system reliability becomes a critical concern in modern architectures with the scale down of circuits. To deal with soft errors, the replication of system resources has been used at both hardware and software levels. Since the redundancy causes performance degradation, it is required to explore partial redundancy techniques that replicate the most vulnerable parts of the code. The redundancy level of user applications depends on user preferences and may be different for the users with different requirements. In this work, we propose a user-assisted reliability assessment tool based on critical thread analysis for redundancy in parallel architectures. Our analysis evaluates the application threads of a parallel program by considering their criticality in the execution and selects the most critical thread or threads to be replicated. Moreover, we extend our analysis by exploring critical regions of individual threads and execute redundantly only those regions to reduce redundancy overhead. Our experimental evaluation indicates that the replication of the most critical thread improves the system reliability more (up to 10% for blackscholes application) than the replication of any other thread. The partial thread replication based on critical region analysis also reduces the vulnerability of the system by considering a fine-grained approach.

KEYWORDS

fault injection, fault tolerance, multicore architectures, reliability, thread vulnerability

1 | INTRODUCTION

Chip multiprocessors (CMPs), which have multiple cores in a single chip, have been accepted as the best way for higher performance.[1] While the performance increases by the use of multiple cores, the reliability becomes an important concern due to the higher vulnerability of multicore architectures. Continuously reducing transistor sizes and increasing the transistor frequencies lead chip components become more error prone, and the transient error rate increases. Soft errors result from a fault in a single-bit and their rates keep increasing with smaller transistors and more aggressive power modes.[2] Soft errors may cause data corruption during program execution as well as program execution termination. Output corrupting faults have different severity and it depends on the corruption magnitude and corruption location for an error to be important.[3]

Redundancy, as a fault tolerance technique, is the replication of hardware and/or software components of a system by targeting to increase reliability.[4-6] The program code is replicated at the instruction level,[4,7] and the results of the duplicate instructions are compared for error detection. On the other hand, there have been methods which execute a redundant copy of the whole program and compare the results at the end of the execution.[8]

The fault tolerance mechanisms have been proposed for parallel applications as well as serial programs.[9,10] The program code is replicated as in the single-threaded case, but the atomic operations are synchronized between master and slave threads to provide correct execution.

Since the redundancy causes performance degradation and resource consumption, the replication of whole program may not be efficient and practical. Therefore, partial redundancy techniques based on the selective replication of instructions in a program have been proposed for higher performance and acceptable reliability.[11-15]

In multicore systems, each additional core used for replication increases the system reliability. On the other hand, each additional core induces power consumption and affects energy efficiency badly. Moreover, potential performance improvement of additional processor core may be obstructed by consuming the core for the replication. To provide performance, reliability, and energy efficiency, it is essential to utilize a few cores for the optimum reliability requirement. If the system tolerates a given level of vulnerability, partial replication by using some of the cores may be a reasonable choice. The soft error vulnerability of a program represents how a program behaves in case of an error, by considering its vulnerable code segments.[16] Since a data might be affected by an error during its lifetime, ie, between Write-Read and Read-Read operations, the vulnerability is defined as the ratio of those intervals in a program. For parallel programs, the interaction between multiple threads also affects the program vulnerability and is considered for the evaluation.[17]

In a parallel application, there are multiple threads running concurrently each responsible for individual task. While some multithreaded applications exhibit data-parallel characteristics in which data is partitioned among threads, the others have multiple distinct tasks assigned for each thread. If the final output is composed of the partial results obtained by different threads, a possible error in one thread causes the final output corruption. However, partial data corruption may not have the same effect with the entire corruption for some applications.

Furthermore, the fault tolerance level of applications running on multicore systems may be selected based on user preferences. Different user requirements may determine how much vulnerability is tolerable for the parallel execution. While some users prefer to execute their applications in a system with a larger number of processors to provide higher reliability, others may tolerate vulnerability by considering higher hardware expenses. A user may also define specific replication limits to be tolerated in terms of both cost and performance. By considering vulnerability tolerance of the system and user-specific requirements, target application is analyzed in order to determine the critical parts to be replicated. In a parallel program, it is reasonable to apply partial redundancy by considering the replication of the critical thread(s) affecting the system vulnerability mostly. Additionally, thread-level redundancy may be extended by exploring critical regions of individual threads and execute redundantly only those regions to reduce redundancy overhead.

Based on above motivations, in this paper, we propose a novel user-assisted reliability assessment tool based on critical thread/region analysis for redundancy in parallel architectures. Thread Vulnerability Factor (TVF) has been proposed and evaluated in our previous work.[17] TVF measures the vulnerability of a thread (which is one of the threads of a multithreaded application) to soft errors by considering the codes of the threads that communicate with that thread as well as the code of the thread itself. In this work, we propose a thread-level vulnerability assessment methodology, which selects the critical thread/region analysis for redundancy in parallel programs, and use TVF for the evaluation of the multithreaded applications' reliability while comparing the vulnerability of various redundancy cases. We can summarize the main contributions of this work as follows:

- We present a *reliability assessment tool* for multithreaded applications which takes into account user preferences. Our tool evaluates the target application for the most vulnerable threads or thread regions, then it recommends the most efficient way for the replicated execution. We report vulnerability values of both redundant and non-redundant cases in terms of Thread Vulnerability Factor metric,[17] which quantifies the vulnerability of multithreaded applications running on multicore architectures.
- We propose a *critical thread identification algorithm* to evaluate the most critical thread in a parallel application for the redundancy. Our algorithm determines the critical thread or threads by analyzing the program for both threads' vulnerability and interactions between threads.
- We extend our analysis to eliminate the *thread regions* that do not contribute to the system vulnerability. We propose a more efficient and precise replication scheme by considering only the replication of critical regions.
- Our experimental evaluation shows that the replication of critical threads provides higher reliability (up to 10 %) than the replication of any other thread in the parallel application. The replication of critical regions from several threads also increases the reliability.

The remainder of this paper is organized as follows. Section 2 explains our fault model considered for critical thread analysis and application execution model. Section 3 presents our user-assisted reliability assessment tool by providing critical thread and critical region analysis. The effect of redundancy on the system vulnerability is examined in Section 4. The experimental setup used in our evaluations is given in Section 5 and results from our experimental analysis are presented in Section 6. Section 7 presents the related work, and it is followed by conclusions in Section 8.

2 | SYSTEM MODEL

2.1 | Fault model

We consider single bit-flip transient faults during the execution of a multithreaded application. Transient faults are transient transitions of single-bit values due to external factors such as particle strikes, electrical noise, and cosmic rays.[2]

A single-bit fault may impact the applications in various ways. In this work, we assume that the execution is correct if the application terminates successfully and produces correct results, and the execution is faulty if the computation results are different from the results without soft errors (Silent Data Corruption).

We further assume that replication does not have any cost or negative impact on system vulnerability. We only consider the vulnerability reduction (see Section 4) in case of any redundancy in the system.

2.2 | Application execution model

Our architectural model is a homogeneous multicore system with private L1 caches and one unified L2 cache. We assume that the system is dedicated to the execution of our target multithreaded application, that is, there is no other application on the system. The target application does not have any interaction with other processes and it is being executed without any interruption. Therefore, we can say that the criticality of threads does not vary in different runs significantly.

We assume that the user provides a representative input data set, and our analysis, which depends on vulnerable intervals and thread interactions, concludes with similar/predictable results for arbitrary input data.[18] Therefore, once profiling the application for critical thread identification with a representative data, the user will learn the most critical thread(s) for reliable execution of this application.

We further assume that we execute the application by mapping one thread onto one core. According to the user preferences, our analysis profiles application with the specified number of threads and decides on the criticality of the threads for the target execution.

3 | USER-ASSISTED RELIABILITY ASSESSMENT TOOL

Our reliability assessment tool (downloadable from http://mimoza.marmara.edu.tr/~isil.oz/research/criticalT.tar.gz) consists of two components as illustrated in Figure 1:

- **User:** The user provides input to the analysis phase by specifying the number of cores to be used for redundant execution of a given parallel application. The number of cores indicates the maximum number of cores that is required by the user for reliability improvement. Then, the user executes the target application with redundant execution of threads and/or thread regions by considering direction advised by the analysis phase. The user also provides input parameters for the application such as a representative input data and the number of threads for the target execution.
- **Analysis:** Our critical thread/region analysis consists of a simulation environment, based on Simics toolset.[19] The decision unit is a front-end component which gets reliability preferences of users (in the form of number of cores for replications) and it initiates the execution of the target application on the simulator. Our critical thread/region analyzer (which is a Simics module) works in parallel with the application, and collects information about the execution threads at runtime. It calculates vulnerability value and criticality degree value of each thread (explored in Section 3.1.3), and it performs region level analysis by considering the synchronization points to determine partial thread replications. At the end of the execution, analyzer decides the most critical threads and/or thread regions. It reports the criticality results to the decision unit of our tool. A complete application execution is the overhead of our tool, which is performed once to gather profiling data and perform criticality analysis. Finally, our decision unit combines user preferences and criticality results to make a suggestion on redundant execution of the application. Once the profiling data is obtained, the thread to be replicated is identified, and the redundant execution is performed by using the critical thread information. The performance of this redundant execution does not depend on our analysis, since our tool is not active during execution. Figure 1 clearly presents the *Redundant Execution* as a component outside *Analysis* part. The analysis part is performed for a sample representative parallel execution before the target redundant execution, and the target execution is performed by replicating the identified critical thread.

As an example, assume that a user has three available cores for replication of an application. Our decision unit may advise the replication of three most critical threads if they are highly critical based on the analysis of our critical thread analyzer in our assessment tool or it may suggest

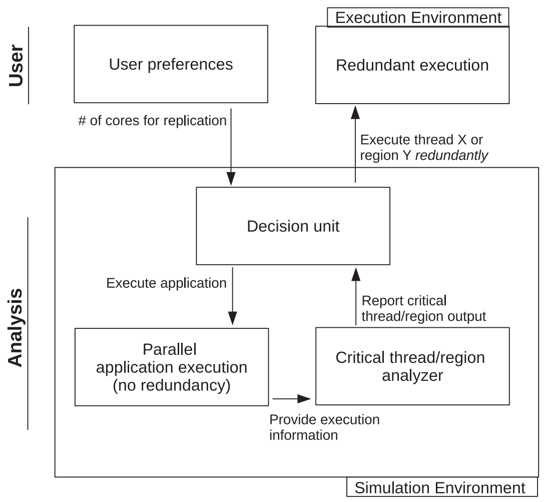
User
User preferences
Execution Environment
Redundant execution
# of cores for replication
Execute thread X or region Y redundantly
Decision unit
Analysis
Execute application
Report critical thread/region output
Parallel application execution (no redundancy)
Critical thread/region analyzer
Provide execution information
Simulation Environment

FIGURE 1 Flow of our reliability assessment tool

that two cores should be used for replication but third one may not be essential, which can be used for performance improvement or powered off for energy efficiency. The decision unit may also advise region level replication if it is more appropriate than thread replication.

3.1 | Critical thread replication

To decide the most critical thread for redundancy, we consider both local vulnerability and interactions between threads.

3.1.1 | Local vulnerability of a thread

Local vulnerability of a thread, represented with Local Vulnerability Factor (LVF)[17] in the literature, is the vulnerability induced on the thread's target code, where architectural resources (including register file, ALU unit, and memory) are evaluated by considering their vulnerable intervals in the code. The redundant execution of the thread that has the largest LVF value increases the system reliability by decreasing the vulnerability of the thread. Since the thread is more vulnerable to soft errors locally, it becomes more appropriate for redundant execution than the other alternatives, by having higher probability of soft error hits.

3.1.2 | Thread interactions

In general, a thread that affects the other threads via remote memory write operations is critical for the system vulnerability since an error in this thread probably causes a failure on the other dependent threads which read the erroneous data from the faulty thread.

Therefore, it seems to be efficient for redundancy analysis to discover thread that has the *most remote writes*, ie, the thread with the highest out-degree value. Additionally, *depth of a thread* and *number of writes to the same destination thread* are the other concerns that are utilized for retrieving thread interactions of our analyzer.

In a multithreaded application, the vulnerability of one thread induced by the other threads should also be considered. Since the threads communicate via shared-memory, an error affecting one thread may cause failure in another thread which reads the erroneous data written previously.

Figure 2 presents a *thread interaction graph* (TIG) for the threads in a multithreaded application, which illustrates the communication of the threads in a timeline. In this application, T_2 frequently writes data which is read by the other threads in the application. If an error hits T_2 and it calculates an erroneous data, the other threads may yield incorrect results using the wrong value. Since T_2 has the largest out-degree, its failure badly affects the system vulnerability by causing the reliability loss for all threads in the application.

There are several thread communication patterns in parallel applications. Figure 3 presents a thread behavior of an 8-thread execution, where T_1 writes data read by other two threads (T_2 and T_4). T_2 and T_4 read data directly from T_1, ie, T_1 writes a data value in a memory location, and then T_2 and T_4 access this location and read the value written by T_1. On the other hand, all other threads (T_3 via T_2; T_5, T_6, T_7 and T_8 via T_4)

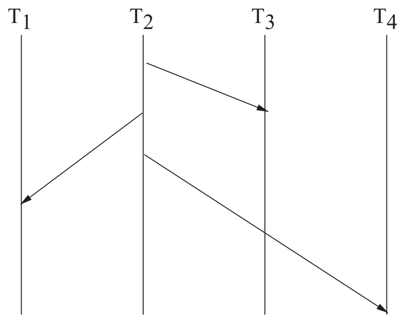

FIGURE 2 Thread interaction graph with four threads (threads in a timeline)

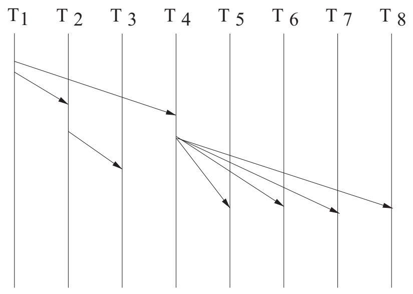

FIGURE 3 Thread behavior of an 8-thread application

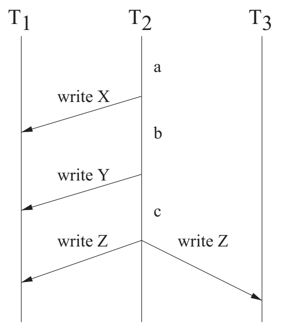

FIGURE 4 An example thread interaction graph which has multiple remote write operations of a single thread

in the execution may be affected by T_1 indirectly. Those threads may read the value written by T_2 and T_4, which calculate the values by using the values from T_1. Therefore, a failure in T_1 is critical for all the other threads in the application. On the other hand, T_4 has many outgoing edges which indicate that many threads may be affected directly if an error hits T_4. The effect of out-degree and the depth may depend on the thread interaction graph of the given application. During the execution of a parallel application, one thread may write data and it is read by a specific thread multiple times. Each remote write operation may affect the remote thread directly, since an error causing incorrect calculation also corrupts the remote thread. Figure 4 presents an example communication scenario in which T_2 writes data read by T_1 for three times (X, Y, and Z values). An error hit at the code segment marked with "a" possibly corrupts X value. The corruption of Y value is possible in case of an error which hits the code segment "b." Z value may be corrupted due to an error on the code segment "c," similarly. Moreover, Y and Z values are dependent on the code segment of T_1 which includes "a" and "a+b," respectively. Since we consider the code segment "a+b+c" for the last write operation (Z value), it is not necessary to count the previous remote write operations (X and Y values) for the critical thread analysis. Although these multiple write operations increase the dependency of T_1 on T_2, they do not enhance the importance of T_2 for redundancy analysis. The same Z value is also read by T_3, and T_2 becomes more critical since it affects two different threads. However, the remote write operations on T_1 and T_3 have the same effect on the critical thread analysis in our tool.

3.1.3 | Critical thread identification algorithm

Our thread-level vulnerability assessment tool considers both local behavior of each individual thread and thread interactions to determine the critical thread for redundancy in a parallel application. The critical thread analyzer (given in Figure 1, implemented as in Algorithm 1) dynamically tracks memory load/store operations during program execution. We calculate LVF value of each thread during execution and we also keep track of thread interactions.

The *criticality degree* of each node in a thread interaction graph is calculated by using two components, which are *direct criticality degree* and *indirect criticality degree*. The former one represents the criticality induced from the write/read relationship between two threads (eg, the interaction between T_1 and T_2 in Figure 3). If one thread alters a memory location and another thread reads that data, the writer thread directly affects the reader thread and becomes more critical due to its effect on the reader thread. This interaction leads to an increase on direct criticality degree value. Since the vulnerability is represented by LVF, we increase value of direct criticality degree of the writer thread by LVF amount of the thread. On the other hand, the *indirect criticality degree* represents the criticality propagated from previous write/read relationships of the writer thread (eg, the interaction between T_1 and T_5 in Figure 3). All threads having affected the writer thread has an indirect effect on the reader thread. This causes a slight increase in indirect criticality degree value of the threads due to the weighted sum of direct and indirect criticality degree values.

We track memory operations in our target parallel application running on a multicore architecture and store them on a *hashmap*, where each entry of the hashmap contains a memory location and the number of the thread that writes to the given location. Whenever a store operation occurs, a new entry, for representing thread that performs the store operation, is added to the hashmap. Algorithm 1 presents our criticality degree calculation procedure. The direct and indirect criticality degree values are stored in *direct_degree* and *indirect_degree* matrices, respectively. If the memory location in the load operation has been altered by another thread previously, the writer thread criticality degree value is incremented due to its effect on the reader thread. In Algorithm 1, the $[tid_{out}][tid_{in}]$ entry of *direct_degree* matrix, which stores total vulnerability effect of the writer threads on the reader threads, is updated. Additionally, the *indirect_degree* matrix is modified by analyzing all threads. For both direct and indirect criticality computations, there are three distinct structures of each resource including *ALU*, *register*, and *memory*. To reduce the effect of the indirect threads, the vulnerability values are multiplied by a *weight* term which is a predefined value in the range [0..1].

Algorithm 1 Algorithm for calculating direct and indirect criticality degree values of threads

```
procedure $on\_load\_operation(tid, location, hashmap)$
    $tid_{in} \leftarrow tid$
    $tid_{out} \leftarrow$ search $(hashmap, location)$
    if $tid_{in} <> tid_{out}$ then
        $direct\_degree[tid_{out}][tid_{in}] += LVF(T_{tid_{out}})$        ▷ Update direct criticality degree values
        for each thread $tid_X$ do
            $indirect\_degree[tid_X][tid_{in}] += weight *$
            $(direct\_degree[tid_X][tid_{out}] + indirect\_degree[tid_X][tid_{out}])$        ▷ Update indirect criticality degree values
        end for
    end if
end procedure
```

After running the multithreaded application and collecting statistics on the behavior of threads, the most critical thread or threads for redundancy is determined based on the Algorithm 2. The algorithm considers criticality degree matrices of vulnerability factors based on a predefined threshold value, ε. If maximum number of remote memory write operations is larger than ε, then degree metrics are utilized to determine the critical thread. Otherwise, LVF values are used since local behavior of threads determines the critical thread in the application. We assign the threshold value according to the pre-experimentation results by comparing values from different applications. The majority of the three resources, which are ALU, memory, and registers, determines the most critical thread of the given application. As an example, if T_1 is selected by both ALU and memory, and T_2 is selected by register unit as the critical thread, then our algorithm returns the T_1 as the most critical thread.

Algorithm 2 Algorithm for determining critical thread of an application

```
procedure $determine(n, \epsilon)$
    $MCT \leftarrow$ the most critical thread
    $RC \leftarrow \max(remote\ count[tid_n])$
    if $RC > \epsilon$ then
        $CT\_ALU \leftarrow \max_n(degree_{ALU}[tid_n])$
        $CT\_mem \leftarrow \max_n(degree_{memory}[tid_n])$
        $CT\_reg \leftarrow \max_n(degree_{register}[tid_n])$
    else
        $CT\_ALU \leftarrow \max_n(LVF_{ALU}[tid_n])$
        $CT\_mem \leftarrow \max_n(LVF_{memory}[tid_n])$
        $CT\_reg \leftarrow \max_n(LVF_{register}[tid_n])$
    end if
    $MCT \leftarrow majority(CT\_ALU, CT\_mem, CT\_reg)$
end procedure
```

3.1.4 | An example execution

Figure 5 presents an example thread interaction graph for an 8-thread execution. For this execution scenario, T_2 reads a value which is calculated by T_1 previously. Then, T_2 writes a value which will be read by T_3. T_4 produces data, and the threads other than T_1 and T_2 read these data values

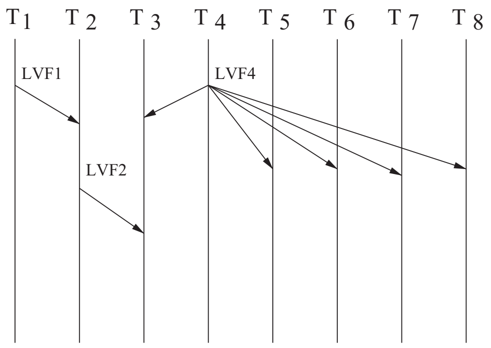

FIGURE 5 A thread interaction graph example with 8 threads

TABLE 1 Matrix for direct and indirect criticality degree values for critical thread analysis

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
T_1	-	LVF_1	$w * LVF_1$	-	-	-	-	-
T_2	-	-	LVF_2	-	-	-	-	-
T_3	-	-	-	-	-	-	-	-
T_4	-	-	LVF_4	-	LVF_4	LVF_4	LVF_4	$LVF4$
T_5	-	-	-	-	-	-	-	-
T_6	-	-	-	-	-	-	-	-
T_7	-	-	-	-	-	-	-	-
T_8	-	-	-	-	-	-	-	-

along their execution. T_1 is the thread with the largest depth which writes to T_2 directly, and T_3 indirectly. LVF_1, LVF_2, and LVF_4 values denote the local vulnerability factor values in the course of remote write operation for threads T_1, T_2, and T_4, respectively.

We may store remote write/read relationships in a matrix structure where the rows represent the *writer* threads and the columns represent the *reader* threads (see Table 1). When a thread, T_1, remotely writes a value which is read by another thread, T_2, the entry in the first row and the second column is filled with the vulnerability value that T_1 has at the time of write operation, ie, $LVF_1 = LVF(T_1)$. This entry indicates that T_1 affects the vulnerability of T_2 by the specified value. The other remote write operation (ie, from T_2 to T_3) is more interesting since there is an indirect communication between T_1 and T_3. Firstly, the cell of the second row and the third column in Table 1 is filled with LVF_2 values for direct communication between T_2 and T_3. This operation also leads to a new entry at the cell of the first row and the third column, due to an indirect communication between T_1 and T_3. Since the effect is not direct and the possibility of that T_1 impacts T_3 is not large as the effect on T_2, we may reduce the vulnerability value by a constant rate (w). The other remote write operations by T_4 to other threads are also added to the related locations of the matrix. At the end of the execution, each row of the table presents the criticality degree of the corresponding thread.

3.2 | Critical region replication

Since the redundancy introduces synchronization overhead between replicated threads, it may be more preferable not to replicate the complete code of a thread if the reliability gain is not significant; one or more critical regions of the thread can be replicated, instead. For critical region replication, we track communication points at thread codes and determine the code region that contributes most to the criticality of the thread. The TIG of an application given in Figure 6 presents remote write operations of threads in an execution. For this figure, the most critical thread for redundancy is T_2 due to its plenty of remote write operations. However, full replication of T_2 may not be necessary. Since most of the remote write operations occur until the *interaction 5*, the replication of the thread code up to this point may be adequate for fault tolerance.

To determine the critical code region of a given thread, we perform criticality degree calculations at the end of each synchronization point during program execution. After gathering thread-level criticality degree values at distinct points, we compare the values of consecutive points. If the values are not significantly different, we may conclude that the replication of the last region is not crucial and suggest the partial replication of the thread by excluding the ineffective region. As an example, if the criticality degree values between the last two write operations (*interaction 5* and *interaction 6* in Figure 6) are not much different, we may say that the redundancy of the code region after *interaction 5* does not provide significant gain.

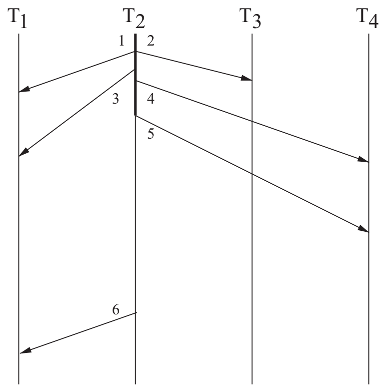

FIGURE 6 A TIG example to represent synchronization of thread regions

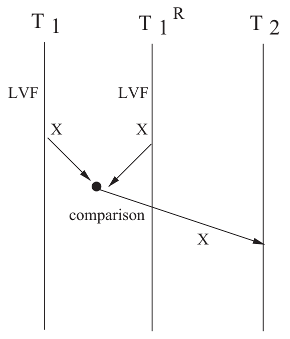

FIGURE 7 A TIG example for thread replication case

4 | VULNERABILITY OF REDUNDANT COMPUTATIONS

To illustrate the effect of the selective thread redundancy on the system reliability, we adapt Thread Vulnerability Factor (TVF),[17,20] which is a new metric for quantifying the relative vulnerability of multithreaded applications. The TVF term has both remote and local parts, where Remote Vulnerability Factor (RVF) constitutes the effect of the remote threads on the target thread and Local Vulnerability Factor (LVF) includes the vulnerability of the target thread. Since TVF metric does not consider thread redundancy, we extend it to evaluate the thread vulnerability for redundant executions. First of all, we should evaluate how TVF is affected by redundant computations and how to calculate TVF if there are redundant threads. Since the LVF term represents the vulnerability of the thread induced by the code itself, its value decreases if there is a redundant copy of the thread. While the redundancy increases the reliability of the redundant thread, it increases the reliability of the other threads as well. The remote term of the thread vulnerability (RVF) represents the vulnerability impact of the threads that interact with the target thread, and it is calculated by considering the vulnerability of remote threads. Specifically, RVF for T_2 in Figure 5 can be calculated as follows:

$$\begin{aligned} RVF(T_2) &= TVF(T_1) \\ &= [w_L \times LVF(T_1)] + [w_R \times RVF(T_1)]. \end{aligned}$$

Assume that T_1^R is the replica of T_1 in Figure 7. Then, the vulnerability (LVF) of the remote thread T_1 decreases; therefore, LVF term should be reduced in the above calculation. Although the reduction amount on vulnerability is not concise, we may consider an approximation to reveal the replication effect. If we multiply the local term by itself, its value decreases and the effect of the redundancy appears in the vulnerability of the dependent thread T_2. RVF for T_2 in the redundant execution of T_1 becomes:

$$\begin{aligned} RVF(T_2) &= TVF(T_1) \\ &= [w_L \times (LVF(T_1) \times LVF(T_1))] + [w_R \times RVF(T_1)]. \end{aligned}$$

Since LVF term takes value in range [0..1], the local vulnerability of T_1 (ie, the partial vulnerability of the remote vulnerability of T_2) is reduced. These reductions on the vulnerability of threads that read data from the redundant thread result in the reduction on the overall system vulnerability which is calculated by augmenting TVF of each thread in the application. For the partial thread replication case, we calculate remote values with the same consideration. We reduce remote values only for the replicated code regions, while the calculation is the same for non-redundant parts.

RVF value of one thread is calculated by adding TVF values of threads communicating with the target thread. Since LVF values have been counted for RVF calculations, only RVF values are considered as the reliability metric by ignoring very small LVF values that are not counted for RVF calculations.

4.1 | A case study for redundant computations

To evaluate critical thread identification and the vulnerability computation of a redundant system, we execute a synthetic application with the thread interaction graph given in Figure 5. In our application, the remote write operations consist of simple array element calculations. The amount of data for each communication is fixed, that is, the number of elements written and read for each interaction is equal. Since the synchronization part in the application is not related to our analysis, we exclude the barrier code which provides atomic operations in the parallel application. We gather vulnerability data for three architectural resources including register, memory, and ALU. The reduction factor for indirect remote writes is taken as 0.8 in our experiments.

The result of the critical thread analysis for this application is given in Table 2, which includes only threads' criticality degrees. We also provide the number of remote write operations (given in remote count row) to point out the out-degree value, without concerning vulnerability concept. Based on Table 2, the most critical thread for redundancy is T_4, which has the largest criticality degree values for all resources. T_4 has also the largest number of remote write operations as seen in the last row.

TABLE 2 Criticality degree values for critical thread analysis of the synthetic application

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
ALU	689.104	399.537	0.000	1839.236	0.000	0.000	0.000	0.000
register	775.859	430.016	0.000	2235.174	0.000	0.000	0.000	0.000
memory	1033.348	474.736	0.000	2457.000	0.000	0.000	0.000	0.000
remote count	1800	1001	0	5001	0	0	0	0

TABLE 3 RVF values of redundant executions of synthetic application

		T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	*Total*
No redundancy	ALU	0.000	0.191	0.239	0.000	0.184	0.184	0.184	0.184	1.166
	Register	0.000	0.215	0.272	0.000	0.223	0.224	0.224	0.224	1.382
	Memory	0.000	0.287	0.313	0.000	0.245	0.246	0.246	0.247	1.584
Redundant = 4	ALU	0.000	0.191	0.182	0.000	0.068	0.068	0.068	0.068	0.645
	Register	0.000	0.216	0.211	0.000	0.100	0.100	0.101	0.101	0.829
	Memory	0.000	0.291	0.252	0.000	0.121	0.122	0.122	0.123	1.031
Redundant = 1	ALU	0.000	0.074	0.210	0.000	0.184	0.184	0.184	0.184	1.020
	Register	0.000	0.094	0.242	0.000	0.223	0.224	0.224	0.224	1.231
	Memory	0.000	0.172	0.285	0.000	0.246	0.246	0.247	0.247	1.443

After we determine the most critical thread for redundancy in the application, we gather TVF values for the redundant case. Table 3 represents the vulnerability values of each thread in the synthetic application. We evaluate the redundant cases for T_4, which is the largest criticality degree values for our critical thread analysis, and T_1, which is the largest depth (ie, its depth is equal to 2) in the dependency structure. Although our analysis recommends us to replicate T_4, we also evaluate vulnerability values for T_1 redundancy in order to compare the results of different cases. Table 3 demonstrates that T_3, which has the most remote read operations (due to direct read from T_2 and T_4, and indirect read from T_1), has the largest remote vulnerability values for all resources. Since T_1 and T_4 have no dependent threads, their remote vulnerability factor values equal to zero. The vulnerability values for the other threads are similar to each other due to the same number of remote read operations.

We calculate the vulnerability values for redundant cases as explained previously by reducing the vulnerability effect of the redundant thread. While the vulnerability of all threads except T_2 is decreased for the redundancy of T_4, only T_1 and T_2 have smaller vulnerability values for the redundancy of T_1. Thus, total system vulnerability, which is equal to the sum of thread vulnerabilities, decreases by larger amount for T_4 redundant case.

5 | EXPERIMENTAL SETUP

5.1 | Simulation platform

To evaluate our critical thread analysis for the multithreaded applications, we use the Simics toolset.[19] We build an 8-core multicore architecture with private L1 caches, and one unified L2 cache.

To track memory operations of our parallel applications, we use *trace* module of Simics, which provides both instruction and data trace of the program.

5.2 | Fault injection framework

In this subsection, we present details of our fault injection framework, which is for validation of our partial fault tolerance scheme. Fault injection is a dependability validation technique based on the controlled experiments, which introduce faults into the system.[21,22] We conduct a simulation-based fault injection by assuming that the failures are uniformly distributed. A single bit flip on a register of one processor core is introduced during the execution of the target application. In our framework, we select *one bit position* (among 32 bits), *one register* (among 8 registers), *one processor core* (among 8 cores), and *one instruction* (among number of instructions for the target application) for the injection point. In our experiments, we take two sets of bit positions (one bit among the most significant 16 bits or one bit among the least significant 16 bits) and two sets of registers (one register among data registers or one register among address registers). The instruction number that is executed at the injection time is randomly assigned. Therefore, we design 1600 different scenarios (ie, 2 sets of bit positions x 2 register types x 8 cores x 50 replications) for each application considered.

We build an automated tool for fault injection on Simics environment.[23] Figure 8 illustrates our fault injection framework, which includes several tools to implement the phases of the experiments.

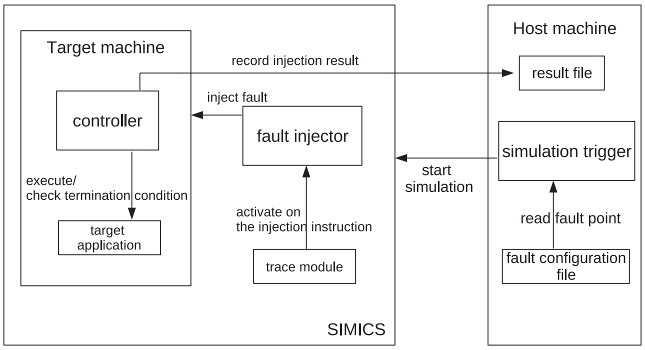

FIGURE 8 Our fault injection framework for validating thread vulnerability assessment tool[23]

Firstly, we randomly create uniformly distributed fault injection points by specifying fault injection instruction, core number, register number, and register bit position and store the parameters of each experiment in a configuration file. Then, our **simulation trigger module** starts the execution of the fault injection simulations by providing parameters of the fault data. Our **fault injector module** enables the **trace module**, which tracks the target application. When the simulation reaches the fault injection point (the execution of the specified instruction on the specified core), trace module activates the related procedure of the fault injector. At this point, the fault injector flips the specified bit and lets the simulation end up. After the injection of the specified fault, the **controller module** waits for the termination of the target application and gathers the result of the experiment by evaluating the termination condition. If the program terminates with an error code (segmentation fault, floating point exception etc.), the experiment result is defined as *program error*. Similarly, the program may never terminate if the fault causes an infinite loop error or similar stuck failure. The result of this kind of execution is also defined as *program error*. If the program terminates normally (with zero exit status), there are two different scenarios: *correct execution* and *output error*. To understand the result of the execution, the controller checks the output file created by the program by comparing it to the golden output file. If there is no difference between the output files, then the result is a *correct execution*. Otherwise, the result has an *output error* and the details are logged into the fault injection result file. When a new result is logged in the file, the simulation trigger finds out that the experiment finished, and continues with the following experiment. The result file is formed online by executing the experiments and processed offline by an external analysis tool to conclude overall fault injection scenario.

5.3 | Benchmark applications

In our experimental analysis, we perform critical thread evaluation for a variety of parallel programming patterns and thread behavior. We select four main classes of patterns.[24] We evaluate our analysis on parallel benchmark applications from PARSEC[25] and SPLASH-2[26] suites by choosing applications that exhibit different patterns.

- **Task parallel:** The thread tasks are balanced and there is no data dependencies between tasks (*blackscholes*, *swaptions* from PARSEC).
- **Divide and conquer:** The task is divided into subtasks, where the solutions to the subtasks are then combined to give a solution to the original task (*radix* from SPLASH-2).
- **Geometric decomposition:** The problem is decomposed into smaller chunks operated in parallel, and the solution is composed of updates to local chunks, and boundaries of chunks which induces data sharing between neighboring threads (*barnes*, *FFT*, *LU* from SPLASH-2).
- **Pipeline:** There is data flow between coarse grained tasks and it is executed on pipeline stages (*canneal* from PARSEC).

We use medium-sized input provided with PARSEC benchmark and default problem size with SPLASH-2 benchmark.

6 | EXPERIMENTAL RESULTS

We execute our benchmark applications on an 8-core multicore architecture by mapping one thread onto one core. Our experimental analysis consists of two phases including *critical thread evaluation* and *critical region evaluation*. We consider the replication of the execution codes (both thread and region) in our redundancy experiments, with the assumption of improving the system reliability. In this work, we do not deal with redundancy levels (duplicate, triplicate) that may have distinct effects on the reliability.

6.1 | Evaluating critical thread replication

For evaluating our critical thread assessment tool, it requires critical thread analysis, replication, and validation phases, which are summarized at the following subsections.

6.1.1 | Critical thread analysis

An application is executed and memory operations are tracked to construct dependency structures for critical thread analysis. At the end of the execution, we figure out the most critical thread(s) for replication, and collect statistics to calculate the vulnerability (RVF) of the execution.

Figure 9 presents values of criticality degree terms of each thread for ALU, register, and memory resources, for 7 applications selected from benchmark suites. When the results are examined, the criticality degree values are evidently larger for one thread (T_1) in *blackscholes*, *canneal*, and *swaptions*. Since *blackscholes* and *swaptions* have task-parallel characteristics, their threads have similar tasks. Therefore the threads other than the first thread have similar criticality degree values which are not too large due to the lack of communication. Only the first thread, which distributes the input data to other threads, has large criticality degree values for each resource. *Canneal* application, which exhibits pipeline pattern, has large communication between its threads. However, the threads have similar criticality degree values due to the homogeneous work. Again the first thread having the input data has the largest criticality degree value for all the resources. For *blackscholes*, *canneal* and *swaptions*, the first thread should be selected for replication if we want to increase the reliability with minimum number of replications. Our analyzer suggests exactly one thread replication to the user, and it advises not to use any other processor core for reliability improvement.

Since *FFT* threads exchange data along their executions, the criticality degree values are similar to each other. It is probable that the replication of any thread results in the same amount of reliability gain. We cannot select the thread for redundancy with our critical thread analysis. We may suggest to the user to use as much as possible cores for replication to improve system reliability.

On the other hand, the difference between the criticality degree values of application threads for *LU*, *radix*, and *barnes* is more apparent. Since the application threads have diverse characteristics, the critical thread analysis becomes more essential for these applications. While *LU* threads have more similar criticality degree values, *radix* and *barnes* threads exhibit diverse values. If we have resources for only one thread replication, we select T_3 for *LU* and T_8 for *radix* as well as *barnes* in order to decrease the vulnerability in a most efficient way.

An inference on the least critical thread can be stated by using our critical thread analysis. Since each replication causes additional resource and performance cost, it is also critical to decide the thread which may not be replicated. *Radix* application figures show that T_3 and T_4 have almost no effect on the other threads. If we do not execute these threads redundantly, the reliability does not change significantly due to the lack of remote effects of these threads.

6.1.2 | Critical thread replication

After discovering the most critical thread(s) for redundancy, we re-execute our applications by a redundant copy of the most critical thread (if any) and calculate the vulnerability values by considering the effect of the redundant copies. In this work, we do not deal with synchronization of redundant copies and do not consider performance issues. We assume that redundant threads decrease the vulnerability and partial redundancy reduces the vulnerability with smaller performance degradation. To validate and demonstrate the efficiency of our analysis, we also include experiments of no redundancy and the replication of other threads in the application. We compare the vulnerability values for different replication cases. Our executions include at most one thread replication for each case.

Figure 10 and Figure 11 present the vulnerability values for redundant cases of PARSEC and SPLASH-2 applications, respectively. We include the normal execution case with no redundancy and two additional scenarios with one thread replication for each application. The figures demonstrate the remote vulnerability factor values for each thread as well as the arithmetic mean (shown in A-m bars).

Since T_1 is the most critical thread for *blackscholes*, *canneal*, and *swaptions* applications, we include the first thread replication which demonstrates the vulnerability decrease in case of the most critical thread replication. We also execute a randomly selected thread (T_6) redundantly as a second scenario and calculate the vulnerability values. Figure 10 demonstrates that while T_1 replication increases the reliability significantly, there is almost no effect of T_6 replication in the vulnerability values. We also note that the replication of any other thread would not effect the vulnerability values significantly due to their small contribution to the vulnerability.

Since none of *FFT* threads exhibits distinct characteristics for critical thread analysis, we select two threads (T_5 and T_6) randomly for redundancy in order to evaluate the effect of any thread's replication on the vulnerability. Figure 11A demonstrates that the vulnerability values decrease for redundant cases, where the difference is not significant in average. We expect the same results for any thread replication case.

While *LU* has more diverse communication behavior between its threads, the average reliability improvement for T_3 (the most critical thread) replication is not much different compared to T_5 (the least critical thread) replication. T_3 is the most critical thread, but the difference between the criticality degree values are not much significant. Therefore, the effect of the most critical thread replication is not clear for *LU* application. While the distinct vulnerability values for individual threads differ, the mean values are similar.

The most interesting results that demonstrate the effect of critical thread analysis belong to *radix* and *barnes* applications. T_8 is the most critical thread for both applications. The replication of the most critical thread decreases the vulnerability values for each resource significantly. We include the replication of the least critical threads in order to emphasize the difference between the partial replication cases. Although the replication of thread T_4 for *radix* and thread T_6 for *barnes* causes vulnerability decrease for some threads, the overall reliability improvement is smaller than the case of replicating the most critical thread. These results demonstrate that if we replicate the thread other than the most critical one, the decreases at vulnerability values would not be so high. Therefore, the thread with the highest criticality degree value should be selected, if there are limited resources for full redundancy.

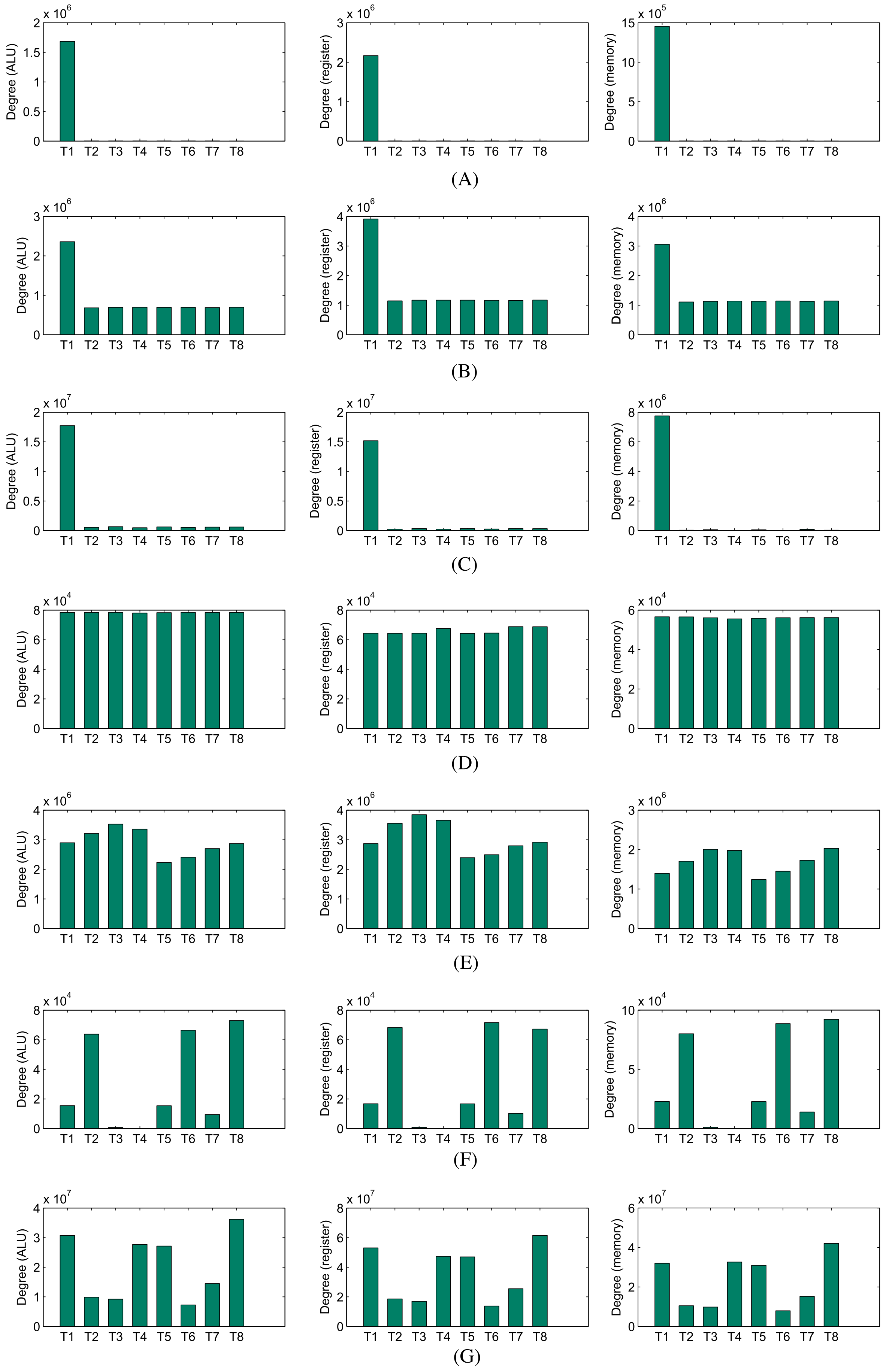

FIGURE 9 Criticality degree values for benchmark applications. A, blackscholes; B, canneal; C, swaptions; D, FFT; E, LU; F, radix; G, barnes

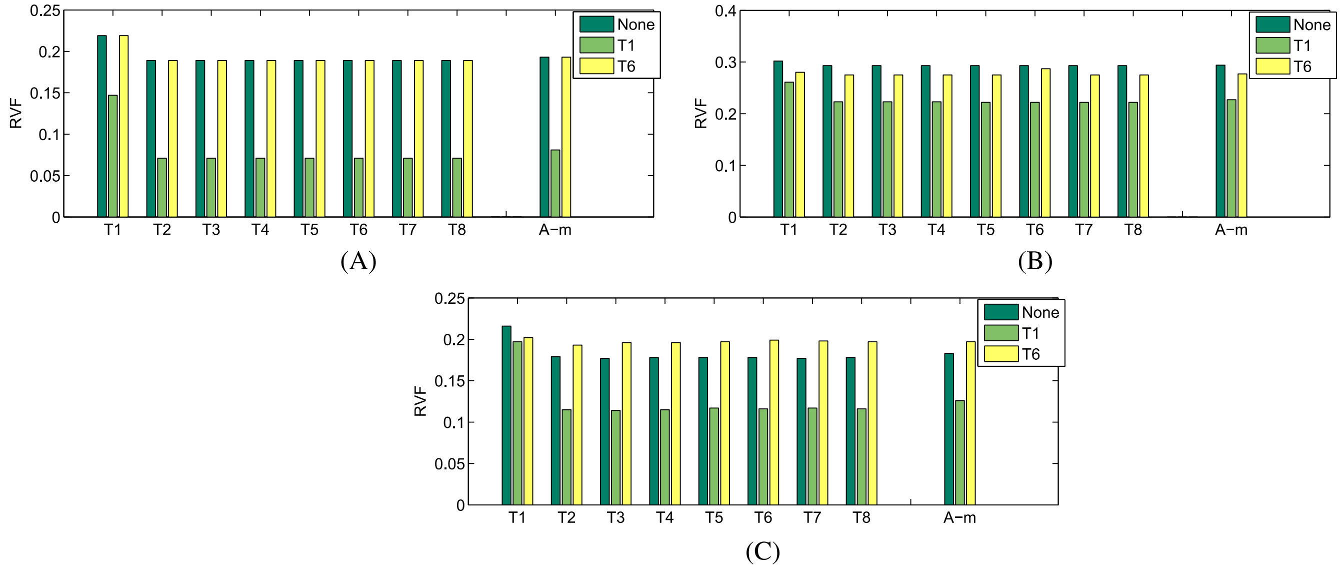

FIGURE 10 The vulnerability values for redundant cases of PARSEC applications. A, blackscholes; B, canneal; C, swaptions

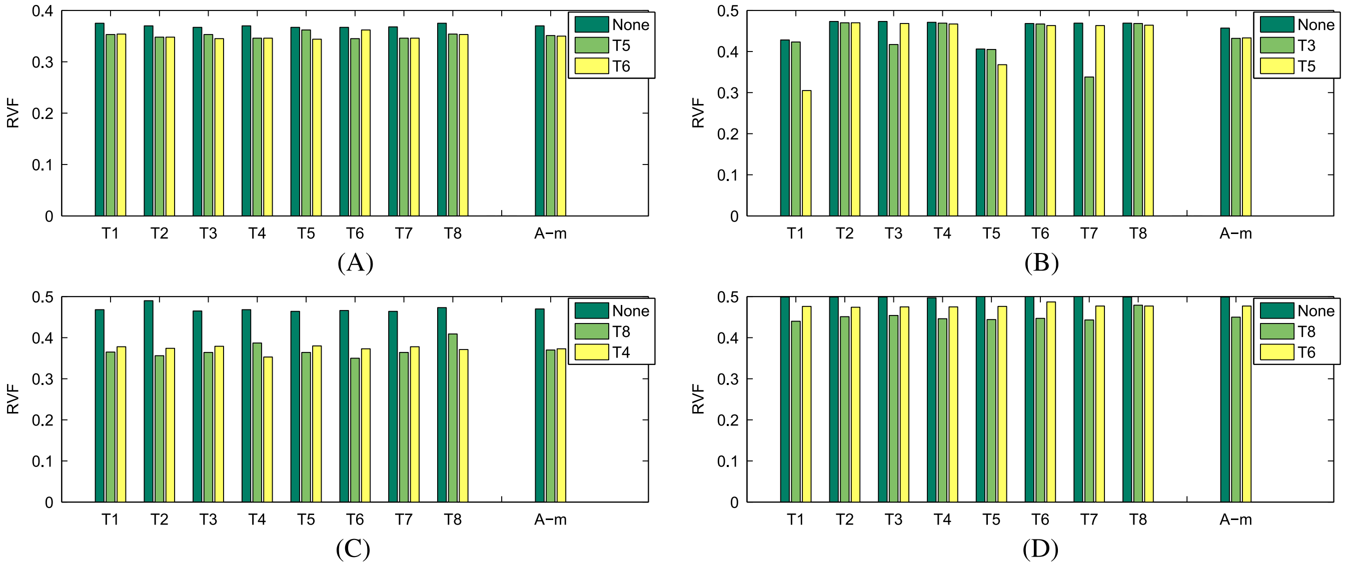

FIGURE 11 The vulnerability values for redundant cases of SPLASH-2 applications. A, FFT; B, LU; C, radix; D, barnes

6.1.3 | Validation of critical thread replication

We execute selected benchmark applications on our fault-injection framework to validate our critical thread-based fault tolerance scheme. SDC (Silent Data Corruption) errors are subtle form of errors considered in our study, which include both self-thread errors and fault propagation errors in a parallel program execution. SDC rate is utilized as a metric to compare results, which is the fraction of the injected faults that results in unacceptable outputs.[27] We do not classify the data corruptions as acceptable or unacceptable; we assume all data corruptions are unacceptable.

To analyze the output errors and detect data corruption, the application output should be deterministic and easy to compare. Therefore, we select a subset of applications that have exact results from the Parsec and Splash-2 benchmarks for our fault injection experiments including *blackscholes*, *LU*, *FFT*, and *radix*. Figure 12 represents the SDC rates of applications for different redundant cases. We include no redundancy case as well as the replication of the most critical and the least critical threads evaluated at Section 6.1.2. While *blackscholes* has significantly lower SDC rates for the most critical thread replication (T_1), *LU* and *FFT* redundant cases have relatively similar results. It is also observed that the replication of T_8 and T_6 which have the largest criticality degree values (see Figure 9) for *radix* application reduces the SDC rates more than the replication of T_3 and T_4 which have the smallest criticality degree values.

In another experiment, SDC rates are computed by varying the number of redundant threads (see Figure 13). We start with no redundant case and include one thread replication by considering the criticality degree value. For example, we assume the replication of T_8, the replication

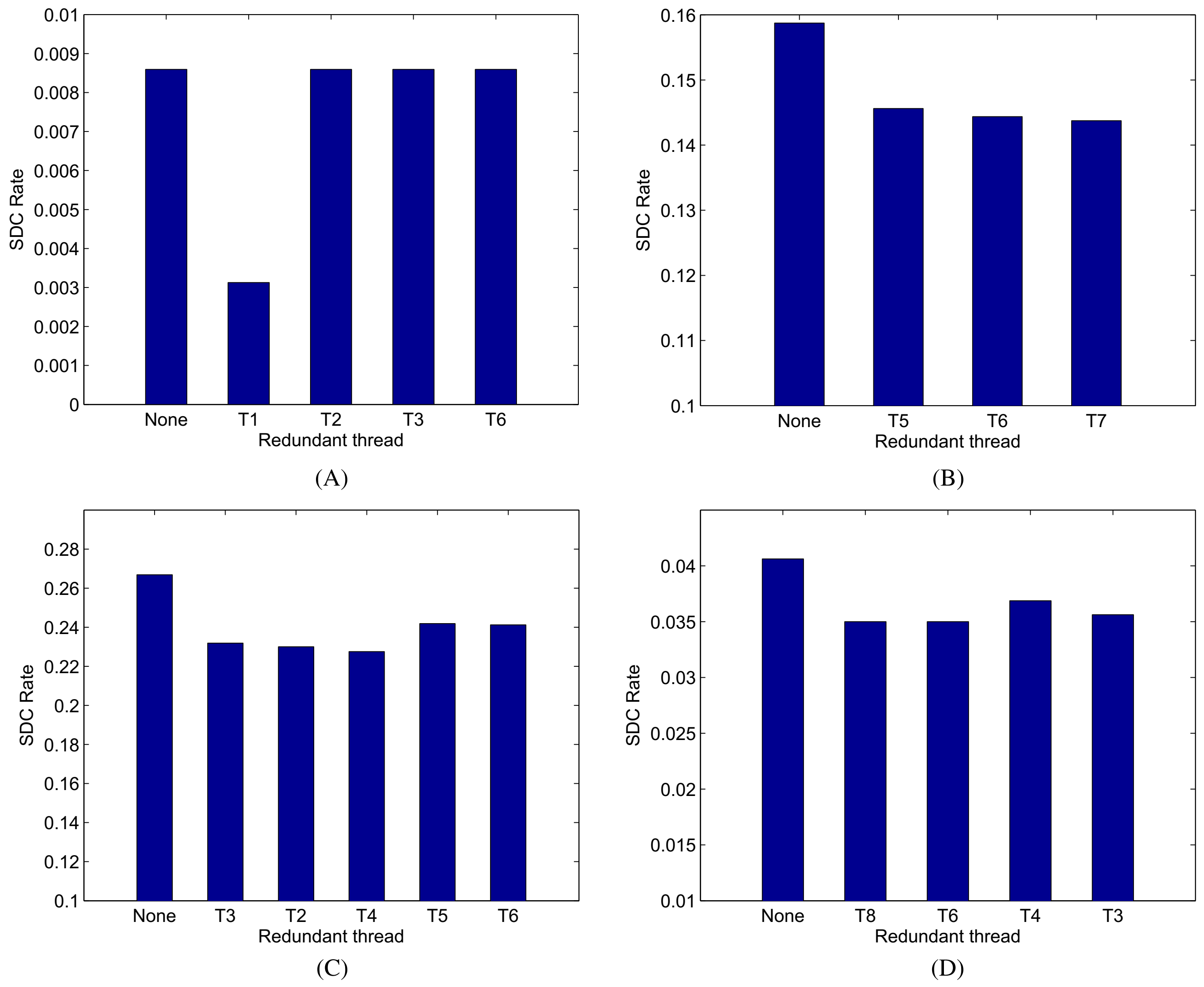

FIGURE 12 SDC rates of applications for redundant cases. A, Blackscholes; B, FFT; C, LU; D, Radix

of both T_8 and T_6, the replication of T_8, T_6, and T_2 for *radix* application for the case of one, two, and, three thread replications, respectively. We construct the complete graph by including one more thread replication and end with full redundancy which results in zero SDC rate. We also conduct experiments for the replication of different number of threads and calculate the vulnerability values by considering the redundancy effect on vulnerability values. The similar results for our RVF values are represented in Figure 13. These results validate our partial replication scheme based on critical thread evaluation.

6.2 | Evaluating critical region replication

We extend our critical thread analysis by considering synchronization points of the applications. We consider execution steps as thread codes divided by these points in our critical region evaluation. The applications that have the behavior of geometric decomposition pattern are more appropriate for critical region analysis due to much communication between threads and many synchronization points. Therefore, we select *LU* for evaluating critical region replications. *LU* has 19 synchronization points in SPLASH-2 implementation. Since the first three intervals do not have any criticality degree values, we include only the last 16 regions for our analysis. We calculate criticality degree values at the end of each 16 execution steps by considering register, ALU, and memory resources. Since register resource has the largest values, we use the results of the that resource in our analysis. We construct graphs for distinct threads in Figure 14 to illustrate the criticality degree values at the end of each execution step represented by the synchronization points in the code.

As presented in our previous critical thread analysis, the most critical threads for redundancy in *LU* application were T_2, T_3, and T_4. While the criticality degree values of these threads do not differ significantly, we may select T_3 for critical thread replication if we have only one available core. However, critical region analysis allows us to replicate different regions from different threads. The most critical regions of the most critical

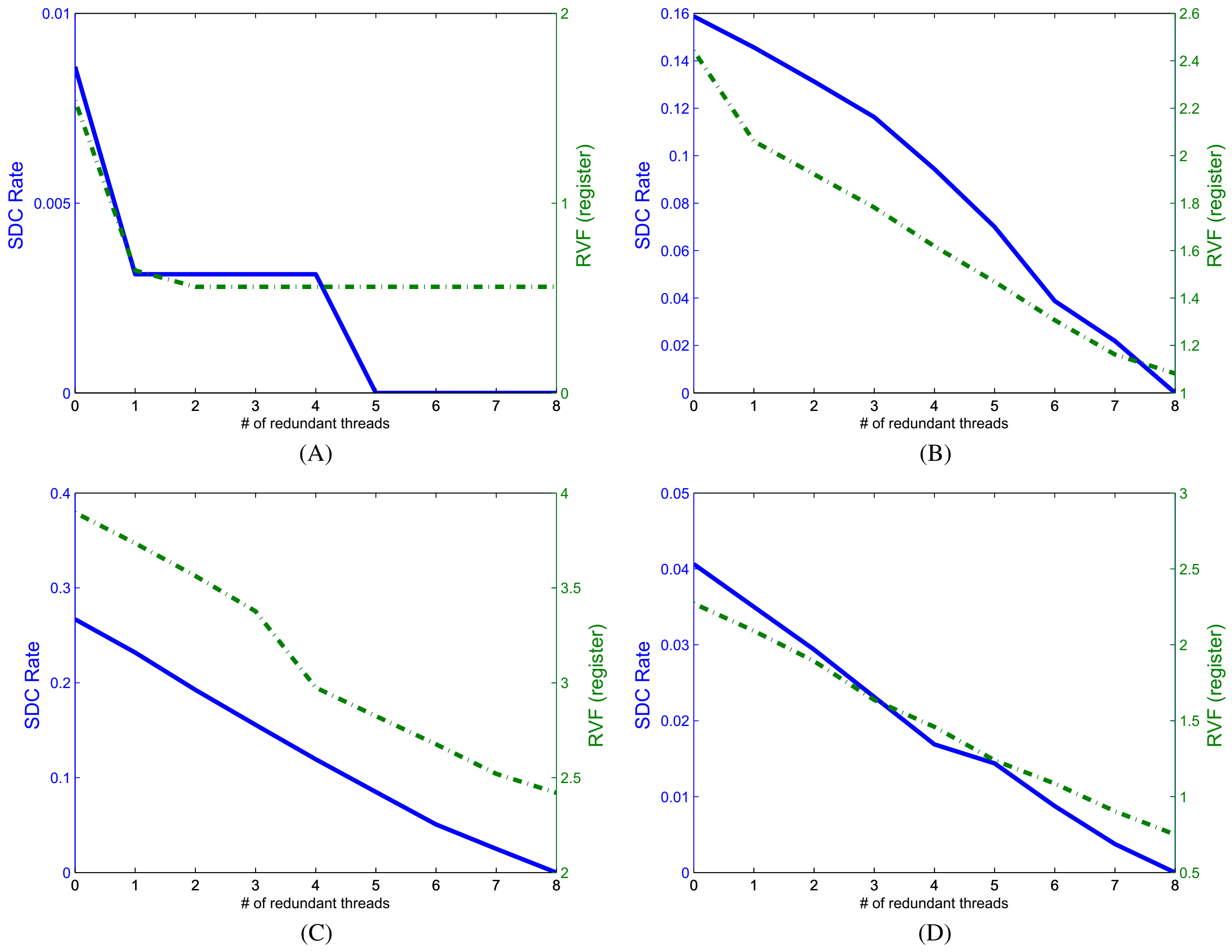

FIGURE 13 SDC rates and RVF values of applications for different number of redundant threads. A, blackscholes; B, FFT; C, LU; D, radix

threads (T_2, T_3, and T_4) are the execution intervals between 10-11, 5-6, and 6-8 (among 16 intervals), respectively. If the user has only one core to use for redundancy, we may advise the selective redundancy for different time intervals. While T_2 is executed redundantly between 10-11 execution steps, T_4 is replicated at 6-8 execution steps. The interval between 5-6 execution step is more critical for T_3, and the redundancy level for other steps may be determined by similar observations. If the synchronization overhead for redundant threads is not tolerable for performance, no redundancy may be applied for some intervals due to slight increase in vulnerability.

To illustrate the effect of critical region replication on the vulnerability of *LU*, we execute the application for different redundancy cases and calculate vulnerability values based on our critical region replication technique. We gather values for both one critical thread replication and partial replication of selective threads for *LU* application. Figure 15 represents RVF values for four cases including no replication, full replication of thread T_2, and two partial thread replication cases (stated as *Partial v1* and *Partial v2*). For the case of *Partial v1*, we replicate the most effective three regions from three most critical threads obtained from our critical region analyzer, which are the replication of T_2 between 10-11 execution steps, the replication of T_3 between 5-6 execution steps, and the replication of T_4 between 6-8 execution steps. We do not execute any redundant code in any other regions. We include the redundancy of relatively less effective regions for *Partial v2* case. Since we assume that we have one available core for the redundant execution, all regions from different threads have been replicated to increase the reliability of the application. We select the replicated thread by considering its contribution at the region and compare the values for the eight threads. For *LU* application, we determine following replications: the replication of T_2 between 1-5 and 10-11 execution steps, the replication of T_3 between 5-6 execution steps, and the replication of T_4 between 6-10 and 11-16 execution steps. As illustrated in Figure 15, *Partial v2* case, which considers both the critical region redundancy from the critical threads and the utilization of the available core for the redundancy, has the smallest vulnerability values. While *Partial v1* case does not present the best case for the vulnerability, it may be a choice to avoid from the performance loss that is induced by the synchronization overhead of the replicated threads.

Another benchmark application for our analysis is *FFT*, which we do not conclude any critical thread identification for full thread replication. The execution steps also have similar behavior for *FFT* threads (see Figure 16). While the first two intervals do not contribute the criticality

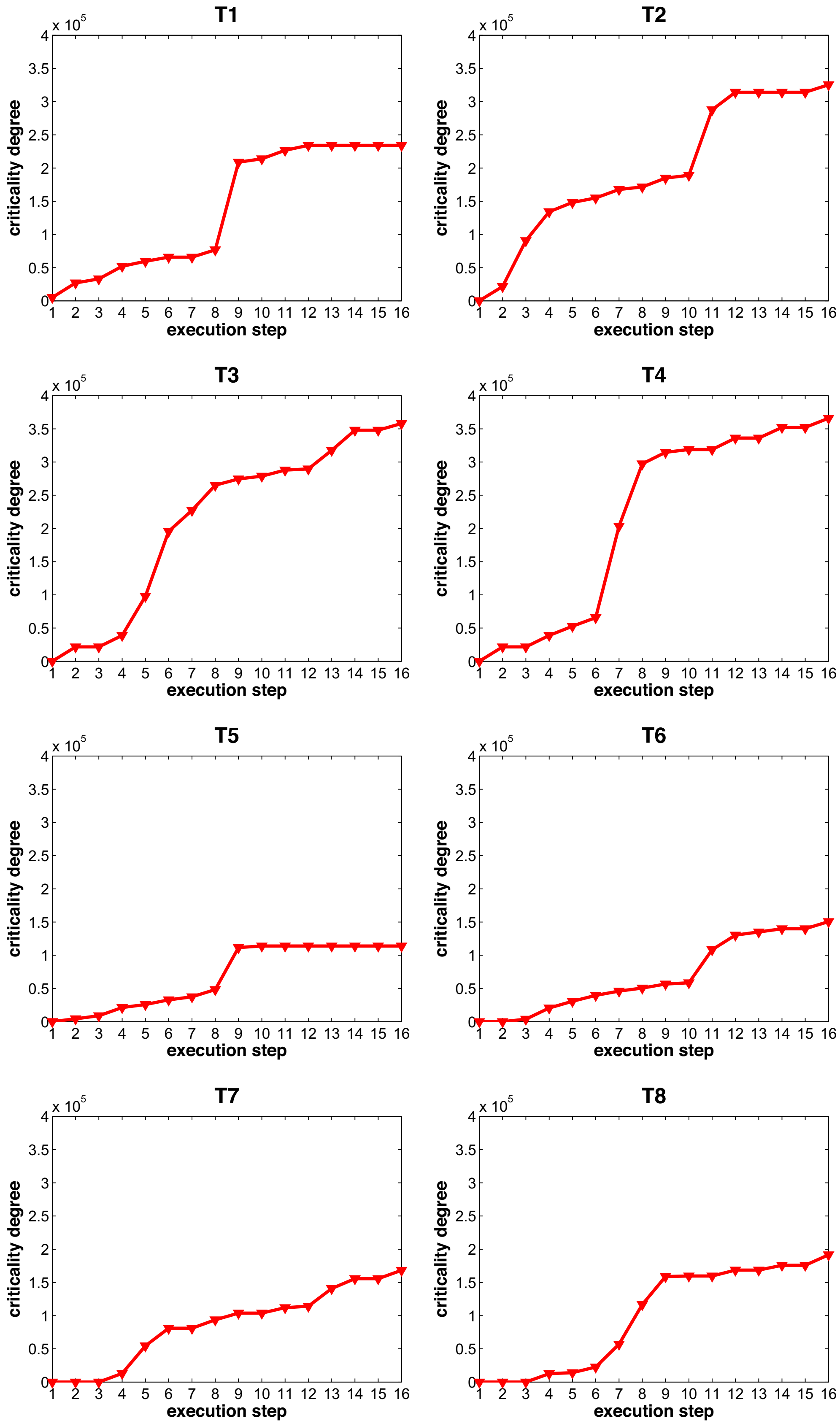

FIGURE 14 Criticality degree values of LU execution steps for distinct threads

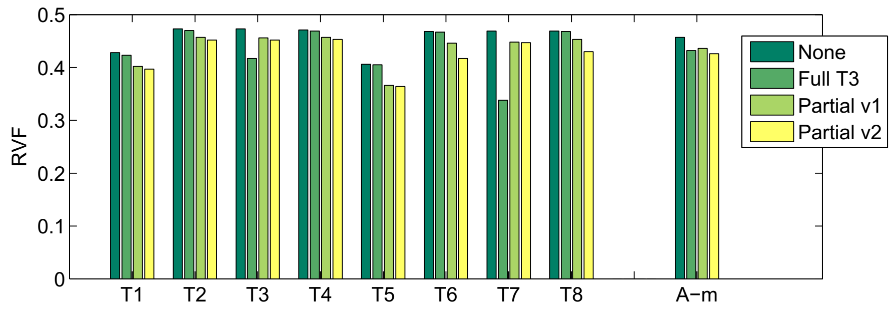

FIGURE 15 Vulnerability values for partially redundant cases of LU application

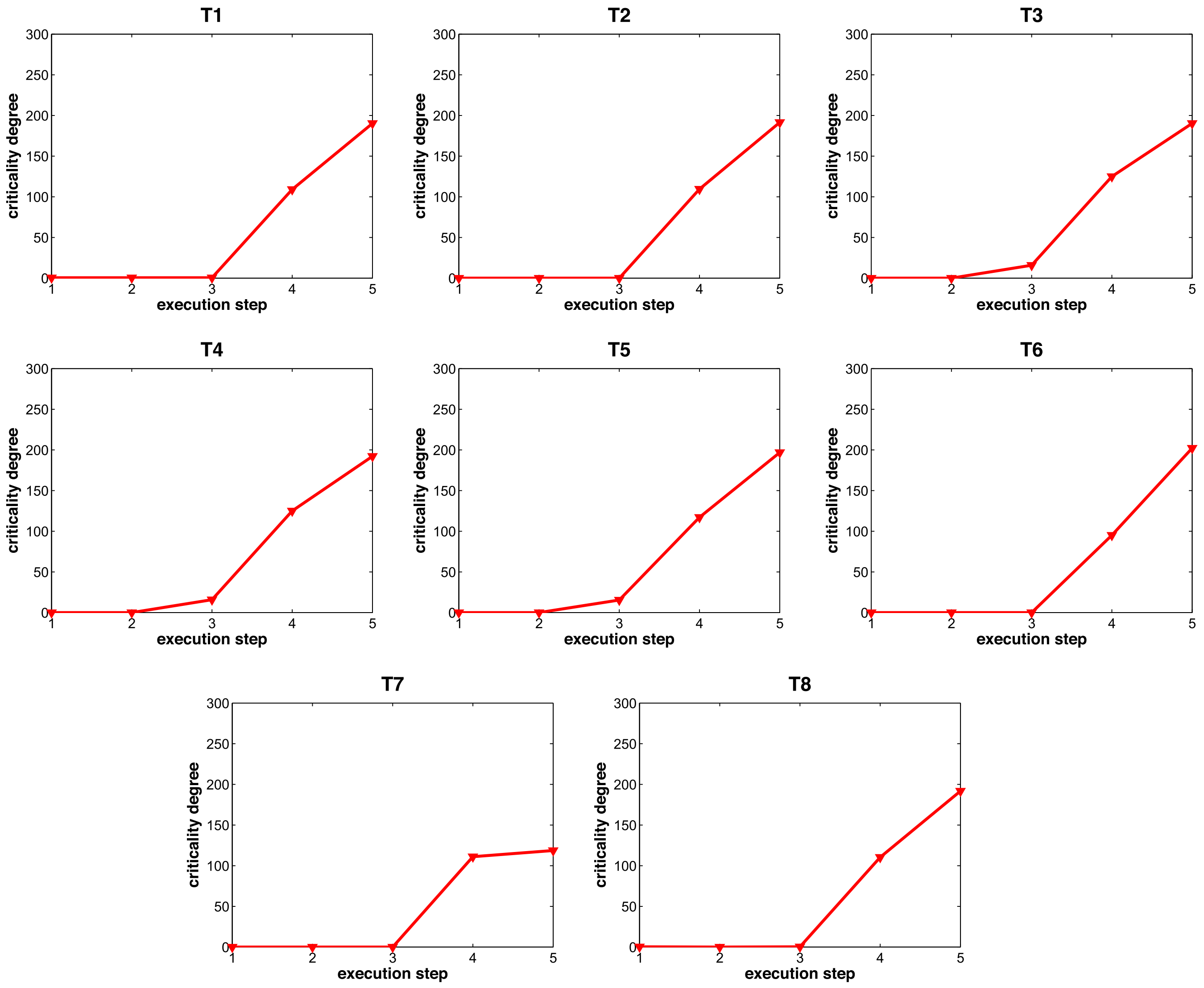

FIGURE 16 Criticality degree values of FFT execution steps for distinct threads

significantly, the last two intervals (especially interval 3-4) are effective for all threads. We may advise to replicate only the code region between 3-4 if the synchronization overhead is very crucial.

7 | RELATED WORK

While the replication of hardware components is a method for reliability,[28-31] there have been software redundancy methods which execute the program code redundantly.[4,7,8]

Since the redundancy methods incur performance overhead in the system, partial redundancy schemes have been proposed to reduce performance loss of full redundancy scheme. Partial explicit redundancy (PER) distinguishes execution phases as Single Execution Mode (SEM) and Redundant Execution Mode (REM).[11] While SEM only executes the main thread, in REM the redundant thread executes with the main thread by considering IPC characteristics. It provides both high soft error-coverage and low performance degradation due to partial redundancy scheme. SlicK proposes a partial redundant threading mechanism based on SRT processor.[32] It uses a set of predictors to estimate the output of the master thread without re-execution. Since the instructions, whose output has been predicted, do not need to be executed redundantly, the performance degradation of full redundancy scheme is reduced. Soundararajan et al[12] propose a selective redundancy mechanism which selects a set of instructions for redundancy to provide maximum performance and minimum vulnerability based on a greedy heuristic dealing with the constraints. Jacques-Silva et al[33] propose a partial redundancy scheme for stream processing applications. Their scheme is based on application quality analysis by considering an application-specific output score function.

Instruction-level redundancy schemes duplicate the instructions in a program at the compile time and compare the results of the replicated instructions at runtime.[4,7] To reduce the cost of the full redundancy, there have been partial redundancy techniques which analyzes how the instructions affect the final application output.[13,34] Instruction-level fault tolerance configurability (ILCOFT) technique[34] provides different protection levels for different instructions in an application by specifying the critical instructions which affect the output at most. ILCOFT-enabled system uses Instruction Vulnerability Factor (IVF) metric to determine the protection level of each instruction. Kumar and Aggarwal[35] propose an instruction-level partial redundancy method to reduce both performance loss and energy consumption. They define self-checking instructions, which do not need replication for fault tolerance, and reduce instruction redundancy level by implementing self-checking in redundant multi-threading scheme. Li et al[36] also propose a selective instruction replication technique by identifying vulnerable instructions at compile time, and replicate a subset of instructions to protect the execution by satisfying worst-case execution time constraints.

There have been redundancy techniques for fault tolerance of shared-memory multithreaded applications as well as single-threaded applications.[9,10] Sanchez et al[9] propose a new design for simultaneous and redundantly threaded method to reduce performance degradation of atomic operations. Atomic operations in parallel applications are handled by synchronizing master and slave threads in the redundant execution. An efficient redundancy scheme to deal with communication latency and nondeterministic ordering of communication events is proposed by mining available redundancy in the program execution.[10]

8 | CONCLUSION AND FUTURE WORK

In this paper, we propose a user-assisted reliability assessment tool based on critical thread analysis for redundancy in parallel architectures. Our analysis evaluates the application threads of a parallel program by considering their criticality in the execution and it selects the most critical thread and critical region, which affect the other threads via remote memory write operations. We demonstrate the efficiency of our tool by providing vulnerability values for executions with different redundancy levels. Our experimental evaluation indicates that the replication of the most critical thread improves the system reliability more than the replication of any other thread. The partial thread replication based on critical region analysis also reduces the vulnerability of the system by considering a fine-grained approach.

We build a proof-of-concept fault injection framework and use single event upset (SEU) in our vulnerability analysis. On the other hand, multi-bit upset (MBU) analysis can be easily adapted in a future work.

ORCID

Isil Oz https://orcid.org/0000-0002-8310-1143

REFERENCES

1. Olukotun K, Nayfeh BA, Hammond L, Wilson K, Chang K. The case for a single-chip multiprocessor. In: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII); 1996; Cambridge, MA.
2. Shivakumar P, Kistler M, Keckler SW, Burger D, Alvisi L. Modeling the effect of technology trends on the soft error rate of combinational logic. In: Proceedings International Conference on Dependable Systems and Networks; 2002; Washington, DC.
3. Shye A, Moseley T, Reddi VJ, Blomstedt J, Connors DA. Using process-level redundancy to exploit multiple cores for transient fault tolerance. Paper presented at: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07); 2007; Edinburgh, UK.
4. Reis GA, Chang J, Vachharajani N, Rangan R, August DI. SWIFT: software implemented fault tolerance. Paper presented at: International Symposium on Code Generation and Optimization; 2005; New York, NY.
5. Reis GA, Chang J, Vachharajani N, Rangan R, August DI, Mukherjee SS. Design and evaluation of hybrid fault-detection systems. Paper presented at: 32nd International Symposium on Computer Architecture (ISCA'05); 2005; Madison, WI.
6. Reis GA, Chang J, Vachharajani N, Rangan R, August DI, Mukherjee SS. Software-controlled fault tolerance. *ACM Trans Archit Code Optim*. 2005;2(4):366-396.
7. Oh N, Shirvani PP, McCluskey EJ. Error detection by duplicated instructions in super-scalar processors. *IEEE Trans Reliab*. 2002;51(1):63-75.
8. Shye A, Blomstedt J, Moseley T, Reddi VJ, Connors DA. PLR: a software approach to transient fault tolerance for multicore architectures. *IEEE Trans Dependable Secure Comput*. 2009;6(2):135-148.

9. Sanchez D, Aragon JL, Garcia JM. Extending SRT for parallel applications in tiled-CMP architectures. Paper presented at: 2009 IEEE International Symposium on Parallel & Distributed Processing; 2009; Rome, Italy.
10. Hyman R Jr, Bhattacharya K, Ranganathan N. Redundancy mining for soft error detection in multicore processors. *IEEE Trans Comput*. 2011;60(8):1114-1125.
11. Gomaa MA, Vijaykumar TN. Opportunistic transient-fault detection. Paper presented at: 32nd International Symposium on Computer Architecture (ISCA'05); 2005; Madison, WI.
12. Soundararajan NK, Parashar A, Sivasubramaniam A. Mechanisms for bounding vulnerabilities of processor structures. In: Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA); 2007; San Diego, CA.
13. Borodin D, Juurlink BHH. Protective redundancy overhead reduction using instruction vulnerability factor. In: Proceedings of the 7th ACM International Conference on Computing Frontiers; 2010; Bertinoro, Italy.
14. Feng S, Gupta S, Ansari A, Mahlke S. Shoestring: probabilistic soft error reliability on the cheap. In: Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems; 2010; Pittsburgh, PA.
15. Vera X, Abella J, Carretero J, González A. Selective replication lightweight technique for soft errors. *ACM Trans Comput Syst*. 2009;27(4):40-70.
16. Sridharan V, Kaeli DR. Eliminating microarchitectural dependency from architectural vulnerability. Paper presented at: 2009 IEEE 15th International Symposium on High Performance Computer Architecture; 2009; Raleigh, NC.
17. Oz I, Topcuoglu HR, Kandemir M, Tosun O. Thread vulnerability in parallel applications. *J Parallel Distrib Comput*. 2012;72(10):1171-1185.
18. Sridharan V, Kaeli DR. The effect of input data on program vulnerability. Paper presented at: Workshop on System Effects of Logic Soft Errors (SELSE-5); 2009; Stanford, CA.
19. Magnusson PS, Christensson M, Eskilson J, et al. Simics: a full system simulation platform. *Computer*. 2002;35(2):50-58.
20. Oz I, Topcuoglu HR, Kandemir M, Tosun O. Reliability-aware core partitioning in chip multiprocessors. *J Syst Archit*. 2012;58(3-4):160-176.
21. Clark JA, Pradhan DK. Fault injection: a method for validating computer-system dependability. *Computer*. 1995;28(6):47-56.
22. Arlat J, Crouzet Y, Karlsson J, Folkesson P, Fuchs E, Leber GH. Comparison of physical and software-implemented fault injection techniques. *IEEE Trans Comput*. 2003;52(9):1115-1133.
23. Oz I, Topcuoglu HR, Kandemir M, Tosun O. Examining thread vulnerability analysis using fault-injection. Paper presented at: 2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC); 2013; Istanbul, Turkey.
24. Mattson TG, Sanders BA, Massingill BL. *Patterns for Parallel Programming*. Boston, MA: Pearson Education; 2004.
25. Bienia C, Kumar S, Singh JP, Li K. The PARSEC benchmark suite: characterization and architectural implications. Paper presented at: 2008 International Conference on Parallel Architectures and Compilation Techniques (PACT); 2008; Toronto, Canada.
26. Woo SC, Ohara M, Torrie E, Singh JP, Gupta A. The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture; 1995; Santa Margherita Ligure, Italy.
27. Hari S, Kumar S, Li M-L, Ramachandran P, Choi B, Adve SV. mSWAT: low-cost hardware fault detection and diagnosis for multicore systems. Paper presented at: 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO); 2009; New York, NY.
28. Austin TM. DIVA: a reliable substrate for deep submicron microarchitecture design. In: Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture; 1999; Haifa, Israel.
29. Reinhardt SK, Mukherjee SS. Transient fault detection via simultaneous multithreading. In: Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA); 2000; Vancouver, Canada.
30. Avirneni NDP, Somani AK. Low overhead soft error mitigation techniques for high-performance and aggressive designs. *IEEE Trans Comput*. 2012;61(4):488-501.
31. Gupta S, Feng S, Ansari A, Mahlke S. StageNet: a reconfigurable fabric for constructing dependable CMPs. *IEEE Trans Comput*. 2011;60(1):5-19.
32. Parashar A, Gurumurthi S, Sivasubramaniam A. SlicK: slice-based locality exploitation for efficient redundant multithreading. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS); 2006; San Jose, CA.
33. Jacques-Silva G, Gedik B, Andrade H, Wu K-L, Iyer RK. Fault injection-based assessment of partial fault tolerance in stream processing applications. In: Proceedings of the 5th ACM International Conference on Distributed Event-Based System (DEBS); 2011; New York, NY.
34. Borodin D, Juurlink BB, Hamdioui S, Vassiliadis S. Instruction-level fault tolerance configurability. *J Signal Process Syst*. 2009;57:89-105.
35. Kumar S, Aggarwal A. Self-checking instructions: reducing instruction redundancy for concurrent error detection. Paper presented at: 15th International Conference on Parallel Architecture and Compilation Techniques (PACT 2006); 2006; Seattle, WA.
36. Li J, Xue J, Xie X, Wan Q, Tan Q, Tan L. Epipe: a low-cost fault-tolerance technique considering WCET constraints. *J Syst Archit*. 2013;59(10):1383-1393.

How to cite this article: Oz I, Topcuoglu HR, Tosun O. A user-assisted thread-level vulnerability assessment tool. *Concurrency Computat Pract Exper*. 2019;31:e5085. https://doi.org/10.1002/cpe.5085