A Survey on Multithreading Alternatives for Soft Error Fault Tolerance

ISIL OZ, Izmir Institute of Technology, Turkey
SANEM ARSLAN, Marmara University, Turkey

Smaller transistor sizes and reduction in voltage levels in modern microprocessors induce higher soft error rates. This trend makes reliability a primary design constraint for computer systems. Redundant multithreading (RMT) makes use of parallelism in modern systems by employing thread-level time redundancy for fault detection and recovery. RMT can detect faults by running identical copies of the program as separate threads in parallel execution units with identical inputs and comparing their outputs. In this article, we present a survey of RMT implementations at different architectural levels with several design considerations. We explain the implementations in seminal papers and their extensions and discuss the design choices employed by the techniques. We review both hardware and software approaches by presenting the main characteristics and analyze the studies with different design choices regarding their strengths and weaknesses. We also present a classification to help potential users find a suitable method for their requirement and to guide researchers planning to work on this area by providing insights into the future trend.

CCS Concepts: • **Computer systems organization → Dependable and fault-tolerant systems and networks**; **Reliability**; **Redundancy**;

Additional Key Words and Phrases: Soft error, thread-level redundancy, redundant multithreading

ACM Reference format:
Isil Oz and Sanem Arslan. 2019. A Survey on Multithreading Alternatives for Soft Error Fault Tolerance. *ACM Comput. Surv.* 52, 2, Article 27 (March 2019), 38 pages.
https://doi.org/10.1145/3302255

1 INTRODUCTION

While CMOS scaling leads to valuable enhancements in performance gain and power consumption [5], with a large number of smaller transistors and reduced voltage levels, modern microprocessors tend to be more vulnerable devices to soft errors [44, 45]. Soft errors (or transient errors), which are temporary malfunctions in the operation of hardware elements due to environmental factors, are known to be a major threat to the reliability of modern processors [64]. Soft-error-induced failures in commercial products emphasize the importance of fault tolerance in those systems [1, 38].

To deal with the reliability problem in modern architectures, researchers in both academia and industry target providing efficient solutions to detect or correct soft errors by exploiting the

Authors' addresses: I. Oz, Izmir Institute of Technology, Computer Engineering Department, Gulbahce Campus, Urla, 35430, Izmir, Turkey; email: isiloz@iyte.edu.tr; S. Arslan, Marmara University, Computer Engineering Department, Goztepe Kampus, Kadıkoy 34722, Istanbul, Turkey; email: sanem.arslan@marmara.edu.tr.

0360-0300/2019/03-ART27 $15.00
https://doi.org/10.1145/3302255

inherent redundancy of modern systems and utilizing redundant resources such as extra cores or threads. In traditional dual or triple modular redundancy techniques [48], hardware components are replicated, and error detection or recovery is achieved by comparing outputs of replicas. However, such hardware-level redundancy techniques incur additional hardware cost and energy consumption overheads. Therefore, the redundant multithreading (RMT) mechanism employing thread-level time redundancy has drawn great attraction with lower cost compared to the hardware-level ones. RMT provides error detection by running the identical copies of the program as separate threads in parallel execution units with identical inputs and comparing their outputs. The redundant threads are executed either on the same simultaneous multithreading (SMT) [74] core or on different CMP cores [49] or parallel execution units in GPGPUs.

In this article, we focus on the discussion of RMT techniques by analyzing state-of-the-art studies in the literature. Additionally, the supplementary studies with extensions to the main works are included to present most of the overall work in this area. We also consider the application of RMT techniques to modern GPU architectures, power-efficient solutions of RMT, and performance-efficient solutions of RMT by using thread-to-core mapping techniques. Although the main focus is on soft errors, we also include the studies that target handling hard errors. Improving reliability with redundancy techniques may have overhead in terms of performance and power consumption. Therefore, we discuss the techniques by presenting their key contributions and findings based on performance, throughput, power consumption, and reliability.

Section 2 summarizes the background information related to soft errors and redundant multithreading as a fault tolerance technique against soft errors. While Section 3 explains our classification methodology for the work done on redundant multithreading in the literature, Section 4 presents RMT-based approaches by analyzing the methods and performing comparison. We conclude the article in Section 5 by providing research directions on redundant multithreading.

2 BACKGROUND AND TERMINOLOGY

2.1 Soft Errors

Faults can be classified as permanent, intermittent, and transient. While permanent faults result from persistent physical changes in a hardware component, intermittent faults occur occasionally and repeat themselves over time, and transient faults are temporary malfunctions of the system components. One or more components of the system deviate from normal operation in case of an active fault, and it is known as an error.

Soft errors, which are a manifestation of transient faults, result from a fault in a single bit in the computer systems as a result of alpha particles, cosmic rays, thermal neutrons, or other environmental causes [64, 81]. If the data in a memory location or a register is affected from a soft error, it becomes corrupted and produces incorrect results until it is updated. This sort of error, also called a single-event upset (SEU), appears randomly and may cause termination of program execution in addition to data corruption during the execution. The presence of such errors may affect the target system in different ways. As an example, the loss of correct functioning of a safety-critical system (e.g., programs controlling an aircraft or a nuclear power plant) can result in catastrophe even if the fault is temporary.

The possible results of a single-bit fault in a hardware structure are demonstrated in Figure 1. Silent data corruption (SDC), which is shown as outcome 4 in the figure, is assumed to be the most crucial one since it produces incorrect outputs in the system. The errors, named detected unrecoverable errors (DUEs), are distinguished by the system; however, they are not recovered by using any mechanism. A false DUE (shown as outcome 5 in the figure) is the case where a faulty bit read occurs; however, it does not influence the program output. A true DUE (shown as outcome 6

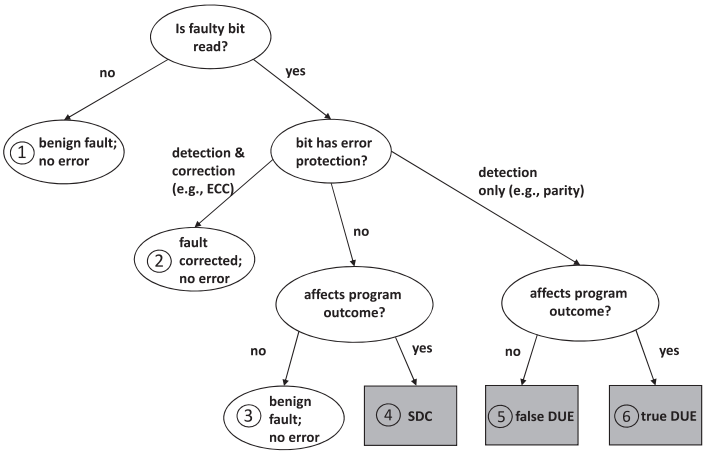

Fig. 1. Possible outcomes of a single-bit fault [45, 81].

in the figure) is the case that influences the program output as a consequence of a faulty bit. Fault tolerance strategies including both detection and recovery mechanisms are utilized with a specific goal to reduce error rates in order to eliminate outcomes 3 through 6 in the figure.

Processor frequencies may not be increased at arbitrary rates, so processor designers aim to place more processor cores in a chip to increase parallelism and throughput [49]. By putting a large number of processor cores on a single chip, they allow running of multiple threads concurrently to exhibit high performance. Reduction in transistor sizes with technology scaling and aggressive low-power optimizations to improve performance affect the susceptibility of modern architectures to soft errors [44]. This trend in transistor size increases the soft error rate (SER) in the chip multiprocessors [6]. More specifically, technology vendors estimate that there will be an 8% increase in the soft error rate per bit with each technology generation [25]. Therefore, soft errors are an unignorable design challenge in modern architectures and its significance is probably going to increment in every upcoming technology [6]. As a consequence, soft error is our main focus in this study by considering multithreading approaches either in single- or multiprocessing environments.

2.2 Soft Error Resilience

Fault tolerance is known as preventing a system from failures in the occurrence of faults [4]. Error detection, error containment, and error recovery are basic steps to provide reliability solution for soft errors. Error detection requires techniques that only recognize the occurrence of an error. In order to prevent the consumption of erroneous data by the system, error containment is needed, which restricts the propagation of error with isolation. After the detection and containment, the system can be restored to an error-free state, which is known as error recovery. Error recovery techniques might be rollback recovery, which means returning back to a previous safe state (e.g., checkpoint), or roll-forward recovery, which means continuing from an erroneous state by making corrections to be able to move forward (e.g., TMR [48]).

As a fault tolerance technique, redundancy, which is basically adding new functionalities to the system to enable correct execution in the presence of errors, is very popular. Redundancy techniques can be implemented in three ways: (1) *spatial redundancy,* which physically replicates a system component; (2) *temporal redundancy,* which re-executes an entire program or some parts of it; and (3) *information redundancy,* which adds new information to program data. In a redundant

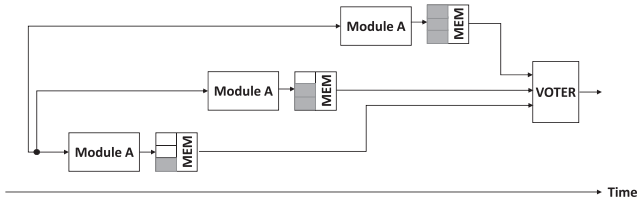

Fig. 2. Temporal redundancy example with one operation module executing multiple times [11].

system, normal system functions should be performed without the need for additional components, and these components are used to detect errors or recover the system.

A widely used spatial redundancy technique is N-modular redundancy (NMR), which utilizes an N replicated copy of processing elements to work on the same data [33]. Dual modular redundancy (DMR), which uses two replicated components to detect errors, and triple modular redundancy (TMR) [48], which utilizes three identical copies of processing elements to detect and correct errors with majority voting, are two popular examples of NMR techniques. In DMR, the outputs of the original and replicated components are compared to detect errors. On the other hand, faulty components cannot be identified in DMR; hence, it is suitable only for error detection. In TMR, the comparator of DMR is exchanged with a majority voter assuming that at least two same results are considered as accurate, and the system continues to the operation with this outcome. In such a case, the error is masked without affecting the correct functioning of the system. TMR is more powerful than DMR since it can both detect and correct errors; however, it is a costly approach in terms of utilized resources. In general, the number of replicas may change in the NMR approach and it can be increased depending on the available resources. On the other hand, adding redundant hardware components to the system causes high hardware cost in spatial redundancy techniques.

In temporal (or time) redundancy, the same operation is performed in an NMR approach; however, the N shows the number of times the operation performed in the same hardware. Figure 2 shows a temporal redundancy example with one module executing three consecutive times to eliminate the output corruption as a result of the soft error. The output of each instance is written to the intermediate memory, and the voter decides the final output by using majority rule [11]. Another well-known temporal redundancy technique is using checkpoints to perform rollback recovery. In this case, the state of the system is saved at specific times and the system is restored by reloading the last safe state in case of an error. Redundant multithreading provides thread-level time redundancy by executing replicated threads either on the same or different cores for error detection or recovery.

In information redundancy, additional information is added to the program data for error detection or correction. Error-detecting codes (EDCs) and error-correcting codes (ECCs) are two information redundancy techniques. As an EDC example, parity check uses a single-parity bit to count the number of 1s (or 0s) in binary data for error detection. As an example of the ECC technique, the Single Error Correction and Double Error Detection (SECDED) technique uses multiple bits to count the number of 1s (or 0s) in different parts of binary data to correct single-bit errors and detect double-bit errors [26].

Temporal or information redundancy techniques do not require any additional hardware component in most cases; therefore, their drawbacks are based on the additional usage of memory space or performance degradation caused from the re-execution rather than the hardware cost. Existing redundancy approaches might have overheads in terms of hardware cost, additional memory usage, performance degradation, and energy consumption. Redundant multithreading approaches

propose a flexible execution environment and aim to provide similar fault tolerance compared to spatial redundancy techniques with minimum hardware costs.

2.3 Calculating Soft Error Rate

In order to calculate the soft error rate, there are two major techniques: statistical fault injection and architectural vulnerability factor (AVF).

Fault injection, which is a traditional way of calculating the soft error rate, quantifies the reliability level of a system by inserting faults into the various system components at different times and evaluates the outcome [10]. In fault injection experiments, one should follow a brute-force approach and count all possible fault injection points by considering fault type, fault location, and injection time. Determining all combinations of these points is tedious and complex work, and the number of experiments might be enormously high. Therefore, an optimized way is to use a statistical approach to utilize a subset of all combinations. In such a case, a number of fault injection points, which is a representative sample of the whole search space, is selected and the soft error rate is given with a statistical confidence rate. With such an approach, the possible size of experiments decreases, but the accuracy level of the soft error rate decreases [80].

Architectural vulnerability factor (AVF) is a metric to evaluate vulnerability from the architectural perspective as the probability of an error in the program outcome in case of a fault in the hardware structure [47]. All of the transient errors may not affect the program outcome; some of them might be masked, and the program continues to execute without being affected by that error. The bits that influence the program output or execution are named Architecturally Correct Execution (ACE) bits, and these bits are used in the computation of the AVF metric. AVF is estimated as the average number of ACE bits over the total number of bits that the hardware structure has in a cycle [47]. The higher value of AVF for a hardware structure means it is more probable to have soft error in the corresponding structure.

2.4 Redundant Multithreading

With the introduction of SMT [74] and then the emergence of multicore processors [49], both providing execution of multiple independent threads simultaneously, RMT has become a widely used fault tolerance technique. RMT enables the execution of two copies of the program as separate threads in multiple execution units (either SMT processors [56], CMPs [46], or GPU cores [31]) by comparing their results for fault detection.

The key concepts for redundant multithreading are the components included in the redundant execution, the inputs to be replicated, and the outputs to be compared after redundant execution. Within these concepts, the Simultaneous and Redundantly Threaded (SRT) processor [56] introduces the *sphere of replication (SOR)* as the logical boundary of the redundant execution. The components in the SOR are included in the redundant execution, while the components outside must be protected by other resilience mechanisms. The values entering the SOR (inputs) must be replicated, and the values leaving the SOR must be compared. Figure 3(a) presents a general sphere of replication. Two execution streams run redundantly inside the sphere of replication, while the input to the original execution is replicated for two copies. After the execution of the redundant copies is completed, the results are compared outside the SOR to detect any fault during execution. As an example implementation, Figure 3(b) presents the SOR of the theoretical SRT processor architecture [56]. The SRT processor assumes that the CPU and registers are to be replicated, but the overall memory components including the on-chip L1 caches, shared L2 cache, and DRAM are outside the SOR, protected by ECC. In other words, the load/store instructions are not executed twice by relying on ECC-protected memory structures; only the computations performed in the

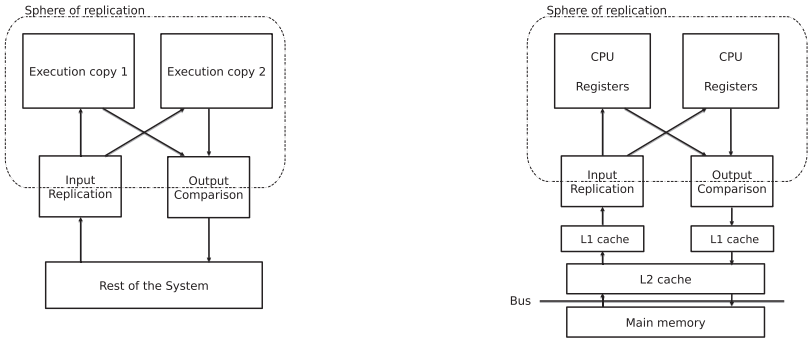

Fig. 3. (a) A General sphere of replication. (b) SRT processor's SOR [56].

datapath using the register file are executed by the redundant threads. Different design choices may implement different SOR boundaries, input replication, and output comparison methods.

If we consider a sphere of replication with the register file as in SRT, there are three output values leaving the sphere and four input values entering the sphere. For output comparison, the following three values need to be considered: *Stores.* The comparator must check the address and value of each committed store before it sends them out of the SOR. *Cached load addresses.* While cached memory load addresses (both data and instruction) exit the SOR, they do not influence the execution since other output comparison detects errors. *Uncached load addresses.* The comparator must check the addresses before the load commits. For input replication, the following four values need to be considered: *Instructions.* The instruction fetch from redundant threads returns the same instruction address with the assumption that the instruction space does not change. *Cached load data.* Since it is possible that other processors update data values and an out-of-order processor requests loads from different threads in a different order, cached load data needs to be handled by using buffer spaces. *Uncached load data.* The value is replicated for both redundant threads after the load data returns. *External interrupts.* Both threads must encounter the interrupts at the same time. It might be solved either by synchronizing the redundant threads by barriers and interrupting both threads or by interrupting the leading thread by recording the execution point and interrupting the trailing thread at that execution point.

3 CLASSIFICATION

We classify the related studies according to their target architecture and main focus. SMT [74] provides execution of multiple threads simultaneously by exploiting instruction-level parallelism. We first present the RMT-based studies on SMT processors in Section 4.1. Then we reserve a separate section for partial redundant multithreading methods implemented on SMT due to their major modifications on the system. With the introduction of multicore processors [49], CMP-based redundant multithreading methods are studied, and we explain the major studies in Section 4.3. After presenting hardware-based approaches, we also explain software-based multithreading methods in Section 4.4.

In the related sections, we first include the most influential studies with high citation and extensions in the literature. We visualize the citation relations among the related studies to emphasize the importance of the work by using the VOSviewer tool [75]. Figure 4 presents the citation graph (created by VOSviewer with data taken from Web of Science) for the related studies given in

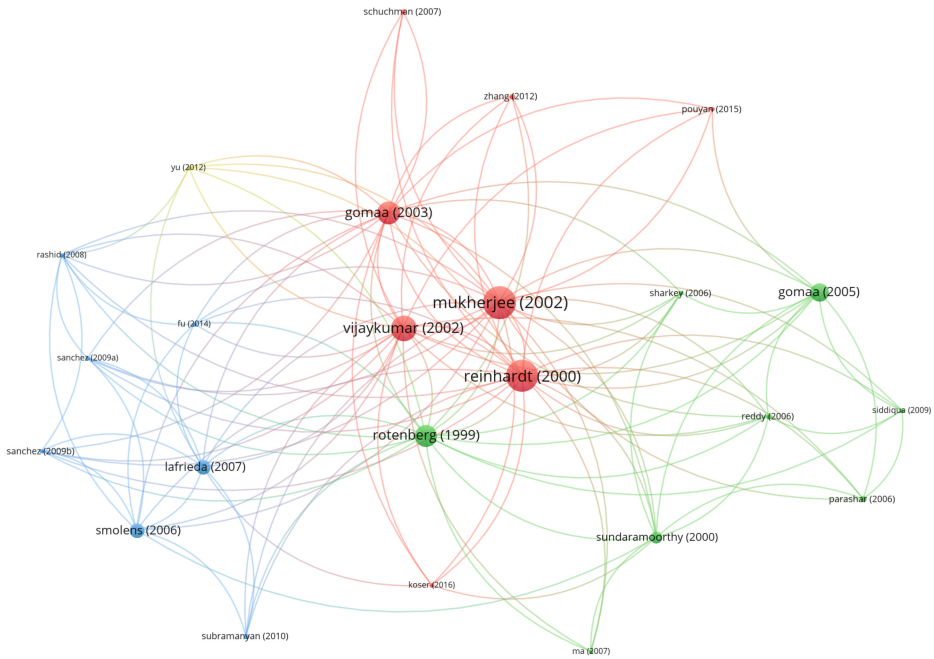

Fig. 4. Citation relations among the studies.

Sections 4.1, 4.2, and 4.3. The graph validates our approach by putting the Reinhardt (2000) [56], Vijaykumar (2002) [76], Mukherjee (2002) [46], and Gomaa (2003) [19] in the center with the highest number of citations. While Rotenberg (1999) [57], Smolens (2006) [66], and Lafrieda (2007) [36] have less weight on the graph, we also include them as the base papers in our analysis (Section 4.1 and Section 4.3) due to their significant impact relative to the remaining studies. We also include two other highly cited studies (Gomaa (2005) [20] and Sundaramoorthy (2000) [73]) in Section 4.2 since they propose partial redundant threading techniques.

After reviewing the main work in the area, we include the significant extensions to present the cumulative work and improvements. We also reserve separate sections for the RMT studies based on GPUs (Section 4.5), dealing with power issues (Section 4.6), and proposing thread mapping for efficient RMT (Section 4.7).

Table 1 shows the general classification of the studies that we have analyzed in this study. We list the research works in several dimensions including the structures in SOR, their key approach or feature, fault types, error recovery support, processor architecture, evaluation platform, and comparison metrics. We place some of the studies in two different rows based on their key feature. As an example, [69] is included in both the RMT application on CMPs and power-efficient RMT. With the emergence of multicore platforms, RMT techniques are widely applied on multicore environments as is seen from the table. The target fault type of the works is transient faults in most cases; however, some of them target permanent faults additionally. Most of the studies prefer to use a simulation environment as an evaluation platform since it is more flexible and easy to reconfigure based on the design specifications. In general, the researchers compare their work based on performance using execution time or instruction per cycle (IPC) metrics. They do not prefer to measure reliability in most studies since the soft errors can be detected or recovered with redundant execution.

Table 1. A Classification of Research Works

Classification		References
Sphere of replication (SOR)	Pipeline	[19, 21, 22, 36, 46, 53, 56, 57, 60, 61, 66, 76] [3, 17, 18, 20, 39, 50, 59, 62, 63, 73, 82] [51, 55, 65]
	Registers	[19, 21, 22, 36, 46, 53, 56, 57, 60, 65, 66, 76] [3, 17, 18, 20, 39, 50, 59, 61–63, 65, 73, 82]
	L1Cache	[17, 18, 36, 51, 53, 55, 66, 73, 82]
Key approach/ feature	RMT via SMT	[3, 39, 56, 57, 59, 62, 63, 76]
	Partial RMT	[20, 50–52, 55, 65, 73]
	RMT on CMP	[18, 19, 21, 22, 34, 36, 46, 53, 58, 60, 61, 66] [17, 18, 23, 40, 71, 72, 82] [7, 8, 15, 32, 35, 42, 52, 54, 68–70]
	Compiler-level software-based RMT	[24, 43, 67, 78, 84]
	Application-level software-based RMT	[9, 28–30]
	OS-level software-based RMT	[13, 14]
	GPU-based RMT	[12, 24, 31, 77]
	Power-efficient RMT	[41, 42, 54, 68–70]
	Thread mapping for RMT	[7, 8, 15, 32, 35, 52]
Fault types	Transient	[16, 19, 36, 46, 53, 56–58, 60, 61, 65, 66, 76] [7, 8, 23, 32, 35, 40, 42, 52, 54, 68–72, 82] [3, 9, 15, 20, 30, 39, 50, 51, 55, 59, 62, 63, 73] [12–14, 17, 18, 24, 31, 77]
	Permanent	[12, 16, 17, 31, 36, 46, 58, 62, 65, 69–72, 77, 82]
Error recovery	Yes	[19, 21, 34, 36, 40, 53, 57, 58, 60, 61, 66, 82] [3, 7, 9, 13, 14, 30, 32, 35, 55, 63, 65, 69, 73] [23, 41, 42, 54, 70–72]
	No	[8, 15–18, 20, 39, 46, 50–52, 56, 59, 65, 68]
Processor architecture	Single core	[3, 16, 20, 39, 50, 51, 55–57, 62, 63, 65, 76]
	Multicore	[18, 19, 21, 22, 34, 36, 46, 53, 58, 60, 61, 66] [7, 8, 15, 23, 32, 35, 40, 42, 52, 54, 59, 68–72] [9, 17, 30, 82]
	GPU	[12, 24, 31, 77]
Evaluation platform	Simulator	[19, 21, 22, 36, 46, 53, 56, 57, 60, 65, 66, 76] [7, 8, 15, 23, 32, 39–42, 54, 59, 62, 68–72] [17, 18, 20, 31, 50, 51, 55, 65, 67, 73, 78, 82] [35, 52, 58, 61]
	Real hardware	[9, 12, 16, 24, 30, 34, 43, 77, 84]
Comparison/tradeoff	Performance	[19, 21, 22, 34, 36, 46, 53, 56, 57, 60, 61, 65, 76] [3, 7, 20, 32, 35, 42, 50–52, 54, 62, 63, 68–72] [9, 13, 14, 30, 39, 43, 55, 66, 67, 73, 78, 84] [12, 17, 18, 23, 24, 31, 58, 59, 77, 82]
	Throughput	[15, 40, 42, 52, 53, 71, 72]
	Reliability	[7, 8, 21, 22, 35, 40, 52, 60, 61, 65]
	Power/energy	[23, 32, 40–42, 54, 68–70, 77]
Academic papers/ industry papers	Academic papers	[3, 7–9, 12–24, 28, 31, 32, 34–36, 39–42] [43, 46, 50–63, 65–73, 76, 77, 82, 84]
	Industry papers	[12, 24, 28, 43, 46, 55, 56, 77, 78]

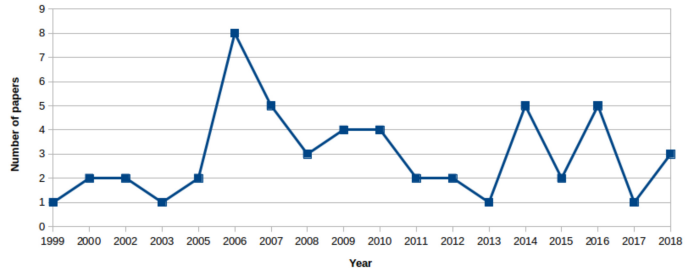

Fig. 5. The number of papers published per year.

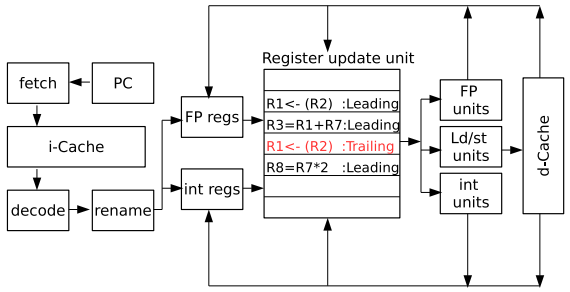

Fig. 6. Simultaneous and Redundantly Threaded (SRT) processor [56].

We survey redundant multithreading techniques proposed during the last 20 years. To present the distribution of the studies over the years, we present the number of papers published per year in Figure 5.

4 SURVEY OF METHODS

This section reviews redundant multithreading studies to present the general approaches and alternative methods to overcome the implementation or mission-specific issues.

4.1 Redundant Multithreading via Simultaneous Multithreading

SMT [74] allows multiple instruction issues from multiple independent threads by exploiting superscalar processors with hardware multithreading. It increases performance by utilizing instruction-level parallelism and provides opportunities for time-redundant fault tolerance. As an example, we put the Simultaneous and Redundantly Threaded processor exploiting SMT for fault detection in Figure 6. The fetch stage starts instructions from multiple threads to a decode queue, where they are sent to the *register update unit (RUU)* with their source operands. The instructions from redundant threads share the same RUU, which issues multiple instructions per cycle to the computation and memory units. The instruction results for each thread are committed to the register file in program order. The studies in the literature evaluate different spheres of replications or propose performance improvements to handle the performance overhead due to thread communications. This section presents SMT-based techniques by providing their main contributions.

AR-SMT. Active-stream/Redundant-stream Simultaneous Multithreading (AR-SMT) [57] proposes a time redundancy fault-tolerant approach based on SMT microarchitecture. It exploits SMT for time redundancy and performs the computations twice by executing two execution streams (A-stream and R-stream) on SMT resources, then compares the results to detect transient errors. The original program thread, called active stream (A-stream), executes and commits the instructions, and their results are also stored in the *Delay Buffer*. The redundant instruction stream

(R-stream) processes the same instructions, and the computation results are compared to those in the Delay Buffer. If they are not the same, a fault is detected. While an error in the R-stream is detected immediately, an error in the A-stream is detected with some delay. AR-SMT also includes operating system support to maintain A-stream/R-stream execution. In order to replicate the operations with some delay in the R-stream, the I/O support allocates a separate memory image and handles address translations. It also synchronizes the A-stream and R-stream for exceptions and context switches.

SRT. The SRT processor [56] provides transient fault detection by executing multiple copies of the program in SMT resources simultaneously. The study is the first that introduces the concept of the sphere of replication (see Section 2.4). It proposes output comparison and input replication design alternatives, as well as evaluating performance improvements in an SRT processor.

The redundant instructions in an SRT processor can execute in different cycles, in different order due to the dynamic instruction scheduling. Therefore, lockstepping (instruction-by-instruction comparison) may not work, and more advanced techniques are required for input replication and output comparison. For input replication, the study proposes two structures: *Active Load Address Buffer (ALAB)* and *Load Value Queue (LVQ)*. The ALAB delays the cache block replacement until the trailing thread commits to guarantee the correct input replication of the load operations having the cached data. In the LVQ, only the leading thread issues the cached load operations and stores the load's address and the value in the queue. The trailing thread gets the data from the queue instead of the cache by waiting for the leading thread's corresponding operation. For the output comparison, SRT explores two spheres of replication: checking the memory accesses only or checking the memory accesses and the register updates. For the memory store operations, a shared *store buffer* is used to synchronize the redundant copies and verify the store values. After the verification, the value is issued to the data cache. Similarly, for the register writeback operations, a *register check buffer* is used for the comparison of the redundant computations.

SRT also tries to improve the performance by two mechanisms. First, the *slack fetch* inserts a constant slack between the leading and trailing threads. The trailing thread probes both the branch predictor and data caches of the leading thread to get some idea about the branch outcomes and cache misses. Second, the *branch outcome queue (BOQ)* mechanism implements a hardware queue to send the committed branch outcomes of the leading thread to the trailing thread. Thus, the trailing thread never mispredicts the branches by using the results of the leading thread as the branch predictor.

SRTR. The Simultaneous and Redundantly Threaded processor with Recovery (SRTR) [76] extends the SRT processor with fault recovery. The instructions of each thread are stored in a private *active list (AL)* besides the shared issue queue. When an instruction leaves the issue queue after completion, it still stays in the AL. The corresponding instructions from the leading and trailing threads occupy the same positions in their ALs to be checked easily.

In order to provide recovery by checking the faults, the results of the trailing thread need to be checked before the leading thread commits the instructions. The slack between the redundant threads must be short, so that the leading thread does not have to wait for the trailing thread for comparison. On the other hand, the slack must be long enough so that the trailing thread can use branch outcomes of the leading thread as the branch prediction. To have a short slack, the trailing thread in SRTR uses the branch predictions of the leading thread instead of the branch outcomes. The leading thread stores the predicted PC value in the *prediction queue (predQ)*, which is similar to BOQ in the SRT. If the leading thread detects a branch misprediction, it performs one of the two actions according to the status of the trailing thread. It clears the incorrect prediction in the prediction queue if the trailing thread has not used the value yet. It squashes the trailing thread's mispredicted instructions in its AL if the trailing thread has already used the predQ entry.

SRTR modifies the LVQ by operating speculative cached loads. It stores pointers to the LVQ in a table, called the *shadow active list (SAL)*, to keep it as efficient as possible. When the load enters the AL, the leading thread creates an LVQ entry and puts a pointer in the SAL. After the issue of the load, the leading thread gets the LVQ pointer from the SAL and stores the load address and the value in the LVQ entry. Then, the trailing thread gets the LVQ pointer from the SAL and compares its own load address with the one in the LVQ. If there is a match, the trailing thread gets the value from the LVQ. Otherwise, it signals a rollback due to the erroneous load address. In case of a rollback, the LVQ pointer in the SAL is placed back before the load operation. Similar to SRT, *store buffer (StB)* is used to compare the address and the value of the store operation.

SRTR uses a sphere of replication with a register file. When the register values are compared, they have been already written to the register file. To avoid bandwidth pressure on the register file, SRTR maintains a structure called the *register value queue (RVQ)*. The leading thread writes the results in the RVQ with a pointer in the SAL, and the trailing thread gets the pointer from the SAL to read the value from the RVQ and to perform comparison. Those operations are completed without creating bandwidth pressure on the physical register file. SRTR also proposes *dependence-based checking elision (DBCE)* to decrease the bandwidth demand on the RVQ. Since the faults propagate through dependent instructions, checking only the last instruction in a dependence chain is enough to discover the faulty case. DBCE exploits register dependencies to store only those instructions in the RVQ to be checked by the trailing thread.

If the leading and trailing thread agree on the instructions' results (stored in RVQ, StB, and LVQ structures), *commit vector (CV)* entries are set to the *checked-ok* state (previously *not-checked-yet* state). A leading thread commits only if the corresponding leading and trailing thread entries are in this state. If the redundant threads do not agree, the entries are set to the *failed-check* state. The leading thread squashes all later instructions in the AL, and both threads are restarted from the latest safe state.

Abu-Ghazeleh et al. [3] propose lifetime-based checking elision (LBCE), which eliminates short-lived values for the verification. An output value is defined as short lived if the destination register has been renamed again before the value is committed.

Sharkey et al. [63] propose a recovery scheme that totally eliminates RVQ in SRTR. In the *RVQ-Free (RVQ_F)* scheme, when the main thread commits an instruction, it stalls and waits for the verification thread to catch up. Then the register states of both threads are compared for fault detection. If there is no mismatch, a checkpoint of the register file is created.

Comparison. Table 2 presents the main SMT-based RMT approaches including AR-SMT, SRT, and SRTR. While the SOR of AR-SMT and SRT includes CPU and registers, SRTR also covers L1 caches. Each architecture includes additional buffer areas to store the redundant results. Their performance results do not yield much difference due to the similar architecture implementations.

Other RMT Extensions with SMT. While AR-SMT, SRT, and SRTR affect the literature substantially (see Figure 4), there have been other SMT-based redundancy studies based on them.

BlackJack [62] presents an SRT extension to support permanent fault detection on an SMT core. The leading and trailing threads in SRT exploit the temporal redundancy to enable soft error detection by executing two identical copies at different times. However, the spatial redundancy is essential to detect hard errors by executing the redundant threads in different hardware components. Since the redundant threads in SRT are almost identical and the redundant executions use the same hardware, spatial diversity is not possible. BlackJack proposes instruction shuffling so that the redundant threads follow different instruction streams. Instead of a naive approach, which forces trailing thread instructions to use different pipeline resources from the leading thread, BlackJack proposes a novel scheme, named *safe-shuffle*. In order to ensure spatial diversity, the redundant threads need to be executed on completely different pipeline stages. Therefore, shuffling needs to

Table 2. Comparison of RMT Techniques via Simultaneous Multithreading

	AR-SMT [57]	**SRT** [56]	**SRTR** [76]
Sphere of replication	CPU including processor pipeline and register files	CPU including processor pipeline and register files	CPU including processor pipeline and register files
Hardware overhead	Delay buffer	Check Store Buffer, Load Value Queue, Branch Outcome Queue	Register Value Queue
Environment	Simple-scalar OoO simulator	Sim-outorder simulator from Simplescalar toolset	Simple-scalar OoO simulator
Processor architecture	SMT-based trace processor/ single core	Out-of-order, speculative SMT processor	Out-of-order SMT processor
Error recovery	Yes	No	Yes, with a little exception
Fault types	Transient faults	Transient faults	Transient faults
Fault model	Single	Single	Single
Fault coverage	Assumption of broad coverage of transient faults and restricted coverage of permanent faults	Not evaluated	Not evaluated
Performance	10% to 30% compared to the single version of the program	SMT-Dual 32% slower, others improve differently	1%, 7% of SRT
Benchmark	SPEC CPU95	SPEC CPU95	SPEC CPU95

be done before the fetch of the trailing thread (not in the middle of the pipelined execution excluding the fetch stage). However, instruction shuffling requires dependence information among the instructions, and the dependencies cannot be known without fetching the instructions in program order. Due to the slack between the leading and trailing threads, the leading thread can be used to obtain dependencies before the trailing thread starts execution. Safe-shuffle uses the leading thread for capturing the dependencies among the instructions, then performs shuffling of the trailing thread instructions before it is fetched. With this implementation, BlackJack covers 97% of hard errors on average with 33% slowdown compared to non-fault-tolerant single-thread execution.

DTE [39] presents a dual-thread execution architecture for SMT processors to provide efficient fault tolerance. The front thread executes instructions as in the normal execution, except long-latency L2-cache misses. L2-cache misses and the dependent instructions are invalidated, and the front thread runs with a virtually ideal L2. The back thread re-executes the instructions by fetching them from the front thread's result queue and also executes twice the invalidated instructions by using the cache misses as prefetching. While the trailing thread contains fewer instructions than the leading thread in SRT, DTE implements the lightweight front thread. DTE proposes efficient fetch policies to improve the performance of SRTR by dealing with resource-sharing issues in SMT architectures. Since the front and back threads share an out-of-order execution core and cache structure, it is critical to allocate resources efficiently between the front and the back threads for better performance. DTE explores different fetch policies that decide the scheduling of the threads and examines their effects on performance. The experimental study reveals that DTE, with full coverage, improves performance by 15.5% on average over single-thread execution and outperforms SRTR for each benchmark application.

4.2 Partial Redundant Multithreading

While the full redundancy exploited by SMT-based approaches provides full coverage from soft errors, the systems with different design choices may prefer low-performance overhead by

tolerating some errors. In order to evaluate performance-reliability tradeoffs, several studies propose partial redundant threading techniques for SMT.

Slipstream. The Slipstream processor [73] presents a processor architecture for the AR-SMT approach by implementing the A-stream and R-stream in two architectural contexts. It mainly focuses on the performance benefit of the approach but also explores the potential fault tolerance improvement. The study introduces new hardware components to support slipstreaming, which represents an execution of a pair supporting each other for both performance and fault tolerance. The *instruction-removal detector (IR detector)* finds and eliminates unnecessary instructions to create a shorter program for higher performance. It monitors the R-stream and constructs a partial dataflow graph for a set of retired instructions (a trace). When a potential instruction to be removed (such as unreferenced writes, nonmodifying writes, and branch instructions) is encountered, it is selected. Then back-propagation in the graph is performed to obtain dependent instructions to remove. The IR detector sends the potentially skipped instructions by the A-stream to the *instruction-removal predictor (IR predictor)*. The IR predictor, which is a modified branch predictor, collects information for different traces and tries to create a confidence for instructions to be removed by counting the selection of the same trace. If the same trace (a set of instructions) is selected for instruction removal by the IR detector more than a threshold value, the IR predictor removes those instructions from the A-stream. A *delay buffer* provides communication between the A-stream and R-stream to compare the architectural states. When the instructions of the A-stream are removed incorrectly (that should not be removed), IR misprediction occurs. When the R-stream or IR detector detects IR mispredictions, the architectural state of the A-stream is restored so that it is the same with the R-stream. The *recovery controller* handles the addresses of the potentially corrupted memory locations in the A-stream to recover the A-stream's memory context from the R-stream's memory context in case of an IR misprediction. The study analyzes a set of scenarios to understand fault tolerance of Slipstream processing provided by redundant execution streams and claims that it potentially increases the fault tolerance of the execution.

Opportunistic. Opportunistic Transient-Fault Detection [20] proposes a partial redundancy scheme by presenting two opportunistic approaches to minimize the performance degradation for soft error coverage. Partial explicit redundancy (PER) exploits the processor's idle resources for redundancy and activates the redundant execution only when the redundant thread does not block the main thread. Since the performance degradation of the redundant execution results from the competition between the main thread and redundant thread, which are executing simultaneously in an SRT processor, PER does not replicate the execution during high-ILP phases of the main thread. It executes in Single Execution Mode (SEM) with the main thread only unless the main thread underutilizes the resources. Whenever the main thread enters a low-ILP phase or there is an L2 miss, PER explicitly switches the execution to Redundancy Execution Mode (REM), similar to SRT. Unlike SRT, PER checks all instructions including the stores due to the potential fault propagation through the duplicated and nonduplicated instructions. Since the duplication of all instructions in a dependence chain is not probable in PER and the duplication of only store operations provides low error coverage, PER duplicates all instructions in REM. Another issue with PER is that the redundant thread needs to resume before an REM and to get the current state from the main thread. PER maintains the *register delay queue (RDQ)* to store the main thread's current state including register values, branch outcomes, load values and addresses, and store values and addresses. Its size is kept as the slack between the main thread and redundant thread to ensure that the redundant thread matches the resume point. The final issue with PER is to decide the switch time between SEM and REM. PER observes the threads to understand whether *execute-IPC of the main thread + execute-IPC of the redundant thread* is less than the *processor issue width*, which shows the resource usage state. This is helpful if PER executes in REM, but the IPC of the

redundant thread cannot be obtained due to its absence during this period. PER uses the approximation of commit-IPC of the main thread as the IPC of the redundant thread since the redundant thread does not employ any misprediction or cache misses due to its direct access to the main thread information. As a result, PER checks that the condition *execute-IPC of the main thread + commit-IPC of the main thread* is less than the *processor issue width* and switches to REM if the condition holds and executes in SEM otherwise. Implicit redundancy through reuse (IRTR) utilizes dynamic instruction reuse, where the same instruction is executed several times with the same input values. Instruction reuse stores the results of those instructions in the *reuse buffer (RB)* and uses them instead of re-executing instructions. IRTR maintains RB to eliminate the redundant execution of the instructions by providing some coverage with no explicit redundancy and performance loss. The study explores the benefit of both PER and IRTR and presents a tradeoff between the performance and the soft error rate (SER) by reporting IPC and AVF values. The simulation results yield that the combined PER and IRTR mechanism reduces the soft error rate by 56% with only 2% performance loss.

SlicK. Slice-based Locality Exploitation [50] presents a partial redundant threading mechanism by using slice-level execution and exploiting the value and control-flow locality. It executes the leading thread as in SRT but tries to eliminate the trailing thread instructions by predicting the outputs of them. Only the backward slices (instructions along dependency chain) of the unpredicted outputs are executed redundantly. The system identifies the store and branch instructions as the trigger instructions, which present a trigger point for verification. If a trigger instruction is verified, the instruction and its backward slice are not executed redundantly and are flushed from the trailing thread. Those trigger instructions are called *Flush Triggers (FTs)*. On the other hand, if a trigger instruction is not verified, its backward slice is extracted, and the instruction and its backward slice are executed redundantly by the trailing thread. Those trigger instructions are called *Execute Triggers (ETs)*. The verification of triggers is handled by a simple last-value (LV) or Finite Context Method (FCM) predictors for the store instructions, and a pattern-based filter-supported branch predictor for the branch instructions. SlicK also implements an efficient backward-slice extractor, *Slice Extraction Matrix (SliceEM)*, to store the instructions on the backward slices of ETs and FTs and specify them as flushable or redundant. The SliceEM frequently updates and keeps track of dependency chains of the instructions in its buffer. Thus, it is possible to get a backward slice of an instruction at any time during execution without any reverse traversal. Since its implementation relies on a small set of structures and bitwise operations, it works with a tolerable low latency for frequent triggers and slice extraction requirement. The SlicK builds a full system by integrating the predictors and SliceEM into an SRT. The study presents IPC and AVF values for the comparison to the baseline SRT and performs 10% better than SRT with around 2% vulnerability.

Reddy et al. [55] analyze partial redundant threading approaches. The study presents the partial redundant threading (PRT) spectrum by examining Opportunistic [20], ReStore [79], and Slipstream [73] as the examples using partial duplication, confident predictions, and both, respectively. It especially analyzes Slipstream by implementing predictions including branch prediction, silent write prediction, dead write prediction, and silent store prediction. As a result, branches with a high confidence prediction, instructions that predictably write the same value in a register or in a memory location as the previous write, and instructions that are predictably dead are removed from the redundant execution. The experiments conducted in the study reveal that Slipstream implementation can detect the faults in 99.9% of the instructions. The authors also propose a recovery implementation for Slipstream processors. The previous recovery implementation assumes that the duplicated instruction is faulty when it detects a fault and restarts the execution from the duplicated instruction. However, the faulty instruction may be the nonduplicated instruction producing a faulty value for the duplicated instruction that detects the fault. Therefore, the

nonduplicated instruction is evaluated as nonfaulty, and the recovered state may still include the corrupted data produced by the nonduplicated instruction. On the other hand, in the proposed recovery implementation, the duplicated instruction detects the fault, but the execution is restarted from the oldest instruction in the reorder buffer (ROB) instead of the duplicated instruction. Since it is probable that the nonduplicated instruction that produced the faulty value is in the ROB, the recovered state becomes error-free. The experimental evaluation shows that 99% of the instructions are recovered by the best combination of the proposed methods. The study also proposes a novel analysis framework for the comparison of PRT approaches by considering each instruction as potentially faulty and provides an alternate way for a complete fault injection experiment including all instructions as faulty. The analysis covers both duplicated and nonduplicated instructions and simply tries to understand whether an instruction is checked directly or indirectly by the fault tolerance approach.

Pouyan et al. [51] present a partial thread redundancy protection scheme based on the online AVF estimations on SMT processors. They compute AVF for the issue queue, reorder buffer, load/store queue, and register file in an SMT architecture during execution. By using online AVF estimations, they consider two execution modes. While one mode includes a redundant thread to check the instructions, the other mode executes single thread to eliminate performance degradation for reliable intervals. The redundancy mode is switched to either the redundant or single execution. If the AVF is greater than a predefined threshold value for an interval, the redundant thread re-executes the instructions on that interval and checks the store instructions' values with the values of the main thread. Otherwise, the store values computed by the main thread are sent to cache or main memory.

Siddiqua and Gurumurthi [65] analyze the effect of SRT on lifetime reliability. They observe that executing a redundant copy of the same program increases activity on the chip, which in turn reduces the lifetime reliability of RMT techniques. They apply Dynamic Voltage Scaling (DVS), partial RMT, and a hybrid of both techniques to overcome this problem. By using the DVS technique, the voltage level of the processor can be reduced; therefore, the temperature of the chip can be decreased. They determine a temperature threshold for on-chip structures and change the DVS setting based on the difference between the current temperature and the threshold value. The authors also use the partial RMT approach in which the redundant thread is disabled at certain times to improve performance. In this case, when the temperature of on-chip structures is below the threshold, a full redundant execution approach is used; otherwise, they switch to the partial RMT approach. As a hybrid approach, they combine the DVS and partial RMT approaches by setting a temperature threshold. When the temperature is above the threshold, they apply partial RMT first; however, if the temperature does not fall below the threshold for multiple times, then they change the DVS setting by scaling the temperature value. The hybrid scheme improves the lifetime reliability significantly with a comparable performance loss relative to SRT.

4.3 Redundant Multithreading on Chip Multiprocessors

With the emergence of multicore architectures, applying redundant multithreading techniques to CMPs becomes very popular. In those techniques, the redundant threads run on different processor cores to balance the load and achieve high resource utilization. On the other hand, the communication cost between the redundant threads is the main bottleneck of those approaches. Chip-level Redundant Multithreading [46] and Chip-level Redundantly Threaded multiprocessor with Recovery [19] are the primary studies that apply RMT on CMP architectures to provide transient fault detection and recovery, respectively. Figure 7 shows a representative schematic view of the CRTR configuration. As can be seen from the figure, two copies of a given program run on two different processors and the helper structures such as BOQ, StB, RVQ, and LVQ provide information flow

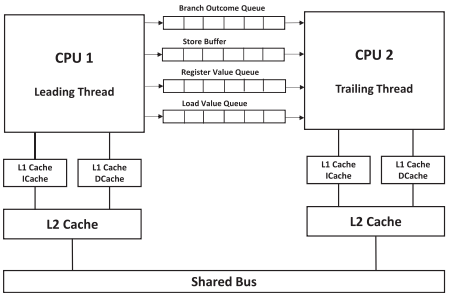

Fig. 7. Chip-level Redundantly Threaded multiprocessor with Recovery (CRTR) [19].

from the leading thread to the trailing one. The proposed studies may differ in sphere of replication, how they reduce the interprocessor communication, or how they handle the synchronization of shared resources. This section summarizes those studies by presenting their key points.

CRT. Chip-level Redundant Multithreading (CRT) [46] is an adaptation of SRT [56] to a CMP environment in which two redundant threads are executed on separate processor cores. The sphere of replication includes the processor pipeline and register files but excludes L1 data and instruction caches. For the input replication, the load value queue is used as in SRT. The leading thread writes the address and the data values of load instructions to the LVQ and the trailing thread loads these values from that queue in an out-of-order way. For the output comparison, there is a store queue to where the leading thread writes the store address and data values, and the trailing thread uses the store comparator to verify its own address and data values with the ones in the store queue. Only one copy of store instruction is released to the cache memory after the verification. Additionally, the *line prediction queue*, a modified version of the branch outcome queue, sends the correct line predictions from the leading thread to the trailing thread. Since the load value queue, store comparator, and line prediction queue structures are utilized by the leading and trailing threads executing on different processors, CRT needs wide datapaths between the processors.

By using space redundancy, CRT covers permanent faults within the transient faults as in the lockstepping approach. Furthermore, CRT has performance advantages over the lockstepping approach by decoupling the execution of redundant threads with the existence of helper structures. The leading thread of an application can be coupled with a trailing thread of another application on the same processor. Therefore, efficient resource utilization can be achieved to enhance the system throughput. The experimental evaluation of the study shows that CRT outperforms lockstepping by 13% on average for multithreaded programs.

Gong et al. [21] propose dual-core redundancy with context saving (DCR-C), which extends the CRT approach to reduce the bandwidth demand for interprocessor communication. Their approach reduces the bandwidth demand by comparing only the store instructions before the commits as in CRT. They save the processor context (such as the register file, PC, and state registers) for recovery, which is performed on cache misses caused from the committed store instructions to hide the saving latency. In case of a transient fault, the redundant cores roll back to the last context saving point for recovery. In order to provide consistent load values in case of recovery, a queue, named CLV, saves the *commit load values* between the successive context-saving points. The performance overhead of DCR-C is 2% compared to CRT, including fault recovery based on the authors' findings.

CRTR. Chip-level Redundantly Threaded multiprocessor with Recovery (CRTR) [19] is an extension of CRT and SRTR [76] that presents a recovery mechanism for RMT in CMPs. Since the leading and the trailing threads are executed on different processors as in CRT, CRTR executes them with a long slack by using an asymmetric commit to hide the performance degradation

caused from the interprocessor communication. In CRTR, the leading thread commits the register updates before checking, while the trailing thread does after checking. Therefore, the state of the trailing thread is used for recovery. CRTR allows the commits for memory updates after checking as in CRT and a consistent memory state can be provided. In case of a fault, an exception is raised and the state of the trailing thread is copied to the leading thread for recovery.

There is a *check buffer (CB)*, which has individual queues for load addresses and values, store addresses and values, and branch outcomes and register values to forward them from the leading thread to the trailing thread. CRTR can recover transient faults except for the faults affecting the register file, which is assumed to be protected with ECCs as in SRTR.

The interprocessor bandwidth is increased with decreasing bandwidth demand in CRTR. To increase bandwidth, interprocessor paths are pipelined by partitioning wires into a set of segments. Bandwidth demand is decreased by checking only committed values and not speculative values and utilizing a *death- and dependence-based checking elision (DDBCE)* scheme. The *dependence-based checking elision (DBCE)* scheme of SRTR may have some problems in masking instructions. A masking instruction may produce a correct result, although its input operands are faulty. The DBCE scheme is extended with DDBCE, which traces the register deaths in masking instructions, and a masking instruction is chained if its source operand dies after the instruction. The register values are assumed to be dead for the input operands of the other instructions later than the masking instruction in the chain; therefore, it will be ensured that the faulty data will not propagate to the remaining instructions. Based on the experimental results, CRTR has imperceptible performance overhead compared to CRT, including fault recovery.

In order to reduce the interprocessor communication demand, Gong et al. [22] propose dual-core redundancy (DCR) and triple-core redundancy (TCR) techniques. In DCR, they simply extend CRT by comparing only the store instructions before commit and provide a recovery mechanism by utilizing the context-saving approach as in [21]. In TCR, they provide an extension of CRTR by executing three redundant threads (leading, middle, and trailing threads) on three different cores. Additionally, they compare only the store instructions to reduce the bandwidth demand. They do not commit the store instructions of the leading thread until they are verified by the middle thread. In case of matching, the results are forwarded to the memory system; otherwise, they use a majority voting mechanism by including the trailing thread's results. The helper structures such as LVQ, StB, and BOQ are also used in TCR to communicate with the redundant threads, but the RVQ structure in CRTR is eliminated to reduce bandwidth demand. Based on their experimental results, DCR can recover most of the transient faults with similar performance overhead compared to CRT, and TCR reduces the bandwidth demand with similar performance overhead in comparison with CRTR.

Reunion. Complexity-Effective Multicore Redundancy [66] presents a redundancy model that provides relaxed input replication for CMP-based RMT approaches. The previous RMT techniques [19, 56, 76] use the LVQ to store the values from the memory load operations and to enable the trailing thread to consume the values from the queue instead of the memory system. This approach forces strict input replication and complicates the situation for the out-of-order execution. However, relaxed input replication allows the redundant threads to execute independently by issuing load requests directly to the caches. On the other hand, the fault tolerance is not guaranteed as in the strict input replication case.

The Reunion execution model defines two logical processor cores named vocal and mute cores. While the vocal core updates the values to the whole memory system by considering the coherence protocol and the consistency model, the mute core only performs update operations on its private cache by providing no update in the other memory structures. Reunion maintains the cache coherence in the vocal cores only and allows incoherent execution for the mute cores. After successful

comparison of the redundant output, a new safe state is defined to be stored as a fault-free state. Whenever an output comparison mismatch occurs, the recovery mechanism restores the state to the latest safe state defined by the vocal core. Only the undetected soft errors can cause an unsafe state. The authors claim that their approach works well due to the infrequent undetected soft errors. The study implements the proposed execution model on a baseline CMP architecture. The first specialization is about the shared cache controller that manages the local and mute cores' memory accesses. The shared cache controller ignores the mute cache requests due to their incoherent behavior. It provides coherence between vocal and mute cores for only synchronizing requests. The processor pipeline also has changes to support the Reunion execution model. First, the safe state is stored in the register file, store buffer, and memory. Second, output comparison is implemented by an additional in-order stage to compare the fingerprints (the hash of instruction results) of the redundant executions from the core pairs.

Reunion incurs a 5% and 6% average performance penalty for commercial and scientific workloads, respectively, compared to the baseline nonredundant execution. The experimental study also includes detailed results related to the checking overhead, the frequency of the incoherence events, and the serialization overhead.

DCC. Dynamic Core Coupling (DCC) [36] is a flexible version of the DMR technique for CMP architectures in which one core can verify the execution of any other core dynamically at runtime without static core binding. DCC has an advantage over DMR such that it does not require any dedicated communication channels. The communication between the redundant threads is achieved by using the system bus of a shared memory multicore system. The performance degradation caused from the shared bus traffic is amortized by supporting long checkpoint intervals. Two types of cores that do not have to be leading or trailing cores are used as the master and slave cores. Both cores can load data from the shared memory to their private caches redundantly, but only the master core is allowed to write the dirty cache lines back to the shared memory after verifying it with the slave core. Therefore, the cache buffer of each core has an unverified bit for each line to indicate whether that line is verified by the redundant core.

Both hard and soft errors can be detected and recovered with DCC. For fault recovery, it presents on-demand TMR; otherwise, all cores are used as DMR pairs. In case of an interrupt, cache buffer overflow, I/O request, or context switch, a checkpoint request is released by either the leading or trailing cores. Then, the architectural state of the core is compressed and broadcast to other cores via the system bus. Each core compares its own state with the received one and a checkpoint is taken in case of a state match. In case of a mismatch, they apply backward error recovery (BER) and the system rolls back to the last taken checkpoint. If multiple BERs cannot recover from a fault, then a third processor is selected to enable TMR. These three processors, namely, one master and two slaves, execute the next checkpoint interval and use the majority voting approach to obtain correct results. After isolation of a faulty core, the remaining two cores continue execution as DMR pairs. For parallel applications, DCC presents a master-slave memory access window model to provide a consistent shared memory state. Either master or slave threads can open a read window in case of a load instruction, and this window is closed when both threads commit the load instruction. In a similar way, a master or slave thread can open a write window for a store instruction and that window is closed when both threads commit the store instruction. The system does not allow one to open a write window if there is already a read or write window for the same address in a different node (a master-slave pair). On the other hand, opening windows for different addresses, multiple read windows for the same address, or private data read/write operations are allowed to be performed simultaneously. To implement this approach in the hardware level, they put an age table on each cache controller, which indicates the number of load/store instructions from the last checkpoint time to trace open read/write windows. Based on their experimental evaluation,

Table 3. Comparison of RMT Techniques for Multicore Architectures

	CRT [46]	CRTR [19]	Reunion [66]	DCC [36]
Sphere of replication	CPU including processor pipeline and register files	CPU including processor pipeline and register files	Datapath from fetch to retirement (control logic excluding)	Processor including private caches
Hardware overhead	Additional core and buffers (LVQ, LPQ, SC)	Additional core and buffers (LVQ, StB, RVQ, BOQ)	Additional core Buffer to compare fingerprints	Additional core
Environment	Simulator/Asim	Simple-scalar out-of-order simulator	Flexus on Simics simulator	SESC simulator
Processor architecture	Multicore/ eight-wide SMT Alpha 21464 processor	OoO SMT processor/ 2-CPU & 4-CPU CMPs	4-core CMP for nonredundant, 8-core CMP for redundant	SMT CMP with 2, 4, 8, and 16 processors
Error recovery	No	Yes	Yes	Yes
Fault types	Transient & permanent faults	Transient faults	Transient faults	Transient & permanent faults
Fault model	Single	Single	Single	Both single and multiple faults
Fault coverage	Assumption of full coverage of transient faults and high coverage of permanent faults	Assumption of full coverage from single transient faults except from register file and the memory controller unit	Not evaluated	Not evaluated
Performance	Outperforms lockstepping by 13% on average	Negligible performance loss compared to CRT	5%, 6% for commercial and scientific workloads, respectively, compared to a non-fault-tolerant system	3%-20% overhead over a CMP without fault tolerance depending on the checkpoint interval, also outperforms DMR
Benchmark	SPEC CPU95	SPEC2000	Parallel commercial and scientific workloads	Single applications: SPEC2000 Parallel applications: scientific data mining applications such as Splash-2 and Spec OpenMP

the overhead of DCC is 3% for sequential applications and 5% for parallel applications compared to a non-fault-tolerant CMP. Additionally, they compare their age-table-based scheme over the relaxed input replication paradigm proposed in Reunion to provide input coherence between the threads. Their conclusion is that the relaxed input replication does not provide optimistic results under DCC's large checkpoint intervals and it has poor performance especially for the applications having high read/write sharing.

Comparison. Although the proposed studies might use a different sphere of replication, environment, hardware model, or benchmarks, we show the results of the main studies (i.e., CRT, CRTR, Reunion, and DCC) by presenting their hardware overhead, fault coverage, and performance costs in Table 3. The main advantages of using RMT techniques in a multicore environment are to balance the load among the processors, to achieve high resource utilization, and to provide permanent fault coverage since the leading and the trailing threads of an application do not share a single processor. The main disadvantage of those studies is that they require intensive

interprocessor communication with high bandwidth requirements. When we compare CRT with CRTR, the latter has imperceptible performance overhead compared to the former by providing error recovery. CRT and CRTR do not utilize parallel applications in their benchmarks, while Reunion and DCC use parallel applications by presenting effective solutions to the synchronization of the parallel threads. Reunion proposes relaxed input replication by an optimistic low error probability assumption. DCC presents fault recovery by enabling on-demand TMR in their solution with an additional 3% to 20% performance cost compared to a non-fault-tolerant baseline configuration. It supports long checkpoint intervals to hide the interprocessor communication cost, but this might be seen as a problem for some of the parallel applications [34]. DCC utilizes bus-based multicore architectures rather than supporting modern network-based architectures. Furthermore, DCC includes private caches within the processor in its sphere of replication; on the other hand, CRT and CRTR assume that these structures are protected via ECC. Reunion further assumes a circuit-level protection for pipeline control logic.

Other RMT Extensions on CMPs. Apart from the studies mentioned above, there are several studies that extend the RMT approaches on CMP architectures.

Highly Decoupled Thread-Level Redundancy (HDTLR) [53] presents thread-level redundancy (TLR) for parallel programs in a CMP environment such that the redundant threads are highly decoupled from each other in time. The redundant threads, named *computing wavefront* (leading thread) and *verification wavefront* (trailing thread), are executed asynchronously as different hardware contexts on different cores. Their architectural states are compared with each other at certain time periods (named *epochs*) to detect transient faults. In their approach, two wavefronts can read and write their own logical memory independently and the coherence mechanisms are also handled separately. Since the redundant threads are highly decoupled in time, the number of unverified instructions is high, which requires efficient buffering mechanisms. Only the validated data is allowed to be written to the shared memory, and unverified data is buffered in private memories. The store instructions are committed without waiting and written to the *post commit buffer (PCB)* until they are verified. The PCB is simply an extension of an L1 cache with several sections. In case of a store instruction, the value is written to both the L1 cache and the PCB. When the corresponding section of the PCB is filled and verified, it will be written back to the L2 cache. The L1 cache no longer performs write-back operations; instead, the PCB does.

The recovery mechanism in HDTLR is based on the checkpoints. At each epoch, the register content and the valid lines of the PCB of both wavefronts are compared. In case of a mismatch, HDTLR restores a previous checkpoint for recovery. To reduce the bandwidth demand, it compresses the state by using a checksum.

The nondeterminism in parallel applications may result in different outcomes based on the execution timing of both wavefronts. To handle such situations, HDTLR simply tracks the races. It partitions the instructions in subepochs such that there is no data race within a subepoch. To enforce the execution order of subepochs, it presents three approaches: *strict enforcement,* in which a processor has to wait on the execution of subepochs of the other verification processors; *blind speculation,* in which order violations are assumed not to happen optimistically; *selective enforcement*, which is the hybrid of previous approaches, offers to perform strict enforcement for only the tight races happening in close-by temporal proximity. These three techniques have different overheads in terms of performance. While the strict enforcement may keep waiting on the verification wavefront unnecessarily due to a false ordering, the assumption made in blind speculation may not be so realistic. If there is an output inconsistency between the wavefronts in the blind speculation approach, HDTLR uses a rollback mechanism for the verification wavefront and enforces the strict ordering in the second turn. In case of selective enforcement, HDTLR selectively enforces the ordering for a race condition that occurs between two instructions close in time.

Based on the experimental findings, their PCB approach offers 2.5% performance overhead compared to the nonredundant execution, and the memory access tracking model does not affect the performance of the system significantly.

The redundant multithreading technique is applied to a dataflow-scheduled multithreaded processor, named DRMT, in which the redundant threads are executed on different cores of a CMP [18]. The sphere of replication includes the processor pipeline, the thread scheduler, the register file, and the L1 cache. DRMT uses the relaxed input replication approach proposed in Reunion and utilizes the asynchronous output comparison mechanism to decrease the interprocessor communication overhead. It uses a *comparison buffer*, which keeps the unverified store instructions, shared by a core pair. Therefore, there is no need to transmit data among the cores for the output comparison. The approach performs only fault detection with a maximum performance overhead of 60% when the number of threads is 4× relative to the number of processors.

Fu et al. [17] propose an on-demand thread-level redundancy technique for concurrent programs in a multicore environment. The redundant threads run on an adjacent fixed core pair. The processor pipeline, register file, and L1 cache of each core are included in the sphere of replication. A relaxed input replication technique is utilized for the input replication, and only the store instructions' results are compared for the output comparison. Their cross-layer approach needs support from the programming model, compiler, ISA, and micro-architecture. In the proposed on-demand thread-level redundancy technique, the redundancy level of a program might be specified by the user. A master thread can create the child threads of both the master and redundant threads and handles the synchronization of a child thread with its redundant parent. The model can detect both hard and soft errors for concurrent programs with a maximum performance overhead of 100% for multicore benchmarks compared to a non-fault-tolerant baseline.

Yu et al. [82] present a thread-level redundancy approach similar to HDTLR. The sphere of replication includes the processor pipeline, the register files, and the private caches apart from the L2 cache. An additional cache, named *unverified value cache (UV cache)*, keeps the results of the unverified store instructions to prevent a shared L2 cache update before verification (similar to the PCB approach of HDTLR). The proposed fault-tolerant system utilizes a network-based architecture rather than a shared-bus system. The nondeterminism of the parallel applications is handled by enabling L1 cache updates for the unverified store instructions; however, it is not allowed to update L2 cache for the unverified instructions. The thread-level redundancy approach with checkpoint-based recovery mechanism has 3% performance overhead compared to a nonredundant system for shared-memory parallel applications.

Madan and Balasubramonian [40] propose an eager register release technique applied on several RMT techniques such as SRTR and CRTR. Providing redundant copies of threads in RMT techniques has negative effects on the performance because of the shared resources such as a register file. For this reason, they propose utilizing eager register release in the leading thread so the trailing thread can take advantage of using the available registers. A physical register used by the leading thread can be released after the value is rewritten or used by all consumers in the pipeline. They apply hardware modifications to keep track of the various register states. Therefore, a register file with a smaller size can be utilized with improvements in power consumption, register file access time, and reliability. Based on the experimental results, the size of the register file can be reduced by 37.5% with similar throughput compared to the baseline approaches (CRTR and SRTR) with a minor decrease in fault coverage.

Greskamp and Torrellas [23] propose a leader-checker microarchitecture, named Paceline, to improve CMP performance by overclocking (executing at a higher clock rate than the nominal rate) the leader core. On the other hand, the checker core runs at a nominal frequency rate in the same CMP. The checker thread has already executed faster than the leader due to information

flow from the leader core. In order to detect faults, the leader and checker cores compare their architectural states periodically. In order to balance the heat and power consumption on CMP, two cores change their roles periodically (leader core may exchange between two cores). The proposed microarchitecture improves performance by 9% and 21% for different benchmarks, increasing the frequency of the leader core by 30% compared to a baseline configuration without overclocking and the helper structures.

Subramanyan et al. [71] observe that the proposed fault-tolerant CMPs suffer from the throughput loss due to redundant use of the cores. Therefore, they present a multiplexed redundant execution (MRE) technique in which more than one trailing thread is executed on a single core and each leading thread is executed on a different core based on one-to-one mapping. Their coarse-grained multithreading model enables the execution of multiple trailing threads on the same core. Execution assistance forwards load values and branch outcomes from the leading thread to the trailing threads to speed up the execution of the trailing thread. The leading thread executes a program partitioned into chunks and puts the results of them into a *run request queue (RRQ)*. These chunks are verified by the corresponding trailing thread executed on the trailing core. Since there are multiple trailing threads on the same core, usage of RRQ and executing the program as chunks enable fairness among them. The fault detection mechanism is based on exchanging fingerprints of redundant cores periodically and the recovery mechanism is based on restoring to a previous checkpoint. The model enhances the throughput by 23% over a baseline CRT for multiprogrammed workloads.

In another study of the same research group, an adaptive execution assistance approach [72] is applied over the multiplexed redundant execution technique [71]. In this approach, the instruction results fed from the leading core to the trailing one may vary depending on the characteristics of the program executed. In the hardware-based adaptive execution assistance approach, they present adaptive branch forwarding and adaptive critical value forwarding techniques. In the former one, there are different branch forwarding levels that change from sending all branch outcomes to sending only mispredicted ones. The starting level is determined as sending only the mispredicted ones, but the level is incremented or decremented at the trailing core side based on the program phase executed. In the latter one, the instruction results sent from the leading core to the trailing one are kept in the *instruction result queue (IRQ)*. When the queue becomes full, the leading core cannot put the new results in and stalls. In such a case, it starts to send only the results of instructions on the critical path (identified based on a heuristic) and the instruction at the head of the ROB identified as critical. Based on their results, the throughput of their approach with both fault detection and recovery is higher than CRT and MRE.

4.4 Software-Based Redundant Multithreading

Software-based redundant multithreading approaches deal with soft errors in either the compiler, operating system, or application levels by considering the effect of the errors on the software running on the system.

4.4.1 Compiler Level. Compiler-level redundant multithreading techniques (also referred to as software RMT in the related literature) transform the application code into a redundant execution with two communicating threads. The leading thread performs all instructions in the original code with additional communication code, while the trailing thread replicates the computations and compares the results with those received from the leading thread. Figure 8 presents an example transformation for a simple code segment. By the assumption that memory is protected by an error detection mechanism and the sphere of replication only includes computations, the memory load/store operations are performed by only the leading thread, while the ALU operations (additions here) are replicated in the trailing thread. The leading thread sends the computed load

Original code	Software RMT-protected code	
	Leading thread	Trailing thread
r1 = r1 + 4	r1 = r1 + 4	r1 = r1 + 4
	send r1	**receive r1'**
load r2 <- [r1]	load r2 <- [r1]	if (r1!=r1') Error
	send r2	**receive r2**
r3 = r2 + r3	r3 = r2 + r3	r3 = r2 + r3
r4 = r2 + 4	r4 = r2 + 4	r4 = r2 + 4
	send r4	**receive r4'**
	send r3	if (r4!=r4') Error
store r3 -> [r4]	store r3 -> [r4]	**receive r3'**
		if (r3!=r3') Error

Fig. 8. The transformation of the code in software RMT.

address (r1) to the trailing thread, and the trailing thread checks the received value with its own computed one. If they do not match, an error is raised by the trailing thread. Then the leading thread performs the load operation and sends the loaded value (r2) to the trailing thread. The trailing thread directly uses the received value in the subsequent computations since the load operations are not in the SOR. After two threads perform addition operations redundantly, the leading thread sends the values (r3 and r4) to the trailing thread. The trailing thread again compares the received values, which are the operands of the next store operation, with the computed ones. If there is no difference, the leading thread performs the store operation successfully. Since software RMT techniques rely on code duplication and the thread communication overhead becomes very high due to the synchronization requirement between the threads, the related studies propose optimizations to deal with the performance degradation.

SRMT. Similar to the hardware-based RMT approaches, SRMT [78] replicates the original program code into two threads: the leading thread contains the original program instructions and additional operations for communication with the trailing thread, and the trailing thread includes the replication of the computations and compares its results with the leading thread's results. SRMT consists of compiler and runtime support for the target programs.

The SRMT compiler support defines the operations inside the SOR by considering the values duplicated for redundancy and the values checked for error detection. It presents optimizations for fail-stop situations and binary function calls. Since the system calls for I/O operations and shared memory access operations are not included in the SOR, these operations are not replicated (executed twice). Instead, the return values obtained by the leading thread are sent to the trailing thread, and the trailing thread directly uses these values. Similarly, the addresses of the shared memory load/store operations, the values of memory store operations, and the parameters for system calls are sent to the trailing thread by the leading thread, and the trailing thread compares the values by its own computed values. For the fail-stop operations, which represent the volatile and shared variable accesses and incur waiting time for the no-error acknowledgment from the trailing thread to the leading thread, the SRMT compiler behaves optimistically. It relies on the assumption of the infrequent occurrence of those operations and relaxes for some of the memory operations by rescheduling the instructions to eliminate the waiting time; the leading thread may cause an exception even if the trailing thread detects an error. The compiler also provides support for call-to and call-back from binary functions, excluded from the replication, by both the leading and trailing threads.

The SRMT runtime support provides an optimization for the communication overhead between the leading and trailing threads. It uses a software queue that implements *delayed buffering (DB)* and *lazy synchronization (LS)* optimizations to make the communication granularity coarser and reduce the synchronization overhead. The DB technique buffers the data to be sent to the trailing

thread and only sends when the buffer reaches enough size (a predefined limit). The LS technique keeps local copies of the shared variables in the queue implementation and updates the synchronization variables in a lazy way to reduce the communication.

DAFT. Decoupled Acyclic Fault Tolerance (DAFT) [84], which is implemented as a compiler transformation in the LLVM compiler framework [37], also replicates the program computation in a redundant trailing thread and inserts fault detection instructions into program code by performing them off the critical path. The SOR includes the processor and does not cover the memory system by the assumption that memory is protected by error correction codes. The memory operations are treated as in the SRMT; the loaded values are forwarded to the trailing thread, while the stores are executed in the leading thread, and the addresses and the values are checked in the trailing thread. Similar to SRMT, DAFT also tries to optimize the waiting time of the leading thread for no-error condition check by assuming infrequent occurrence of faulty operations. Instead of rescheduling the instructions, DAFT removes the communication dependence between the send signal (error check) of the trailing thread and the wait signal of the leading thread speculatively. To detect the misspeculation, which occurs in case of a transient error, DAFT presents three mechanisms. *In-Thread Operand Duplication* performs the error checking of the volatile load/store operations' operands by duplicating them in the leading thread instead of thread pairs, which eliminates the interthread communication. All of the other operations are compared in the trailing thread with the values produced by the leading thread (*Redundant Value Checking*), and the trailing thread reports a transient fault in case of a mismatch. The *Custom Signal Handler* allows one to distinguish the normal program exceptions from a transient fault. When an exception occurs in the execution, the DAFT signal handler waits for the trailing thread to complete. Unless the value-checking code detects a transient error before a timeout, the exception is reported as a normal exception to be handled by the system signal handler; otherwise, a transient fault is reported and the program is stopped.

DAFT proposes optimizations for minimal communication cost between the redundant threads and smaller number of branches. Branch removal eliminates the branch instructions including nonredundant code in the trailing thread. Interthread communication lifting moves the error checks to remove some interthread communications. The software queue implementation uses a streaming store and prefetching for better performance by accelerating communication.

COMET. The Communication-optimized multithreaded error detection technique (COMET) [43] performs a detailed analysis on the communication overheads of the existing software redundant multithreading techniques and proposes optimizations to reduce the communication time. It evaluates the performance of the queue operations for the communication between the leading and the trailing threads and uses an optimized version of a multisection lock-free single-producer/single-consumer queue to remove synchronization instructions of enqueue/dequeue functions performed by the leading and trailing threads. When the threads reach a synchronization point (defined as red zone), an exception is raised and the custom exception handler performs the synchronization by keeping the threads waiting. In order to allow the handler to have full access to the register file and eliminate the register allocation process for the queue index pointer, which points to the next section in the queue, COMET uses a fixed register for the store and load instructions of the enqueue and dequeue operations. COMET optimization also includes enqueue function inlining, embedding the increment of the index within the memory instruction for the consecutive enqueue (or dequeue) operations (called as address offset fusion), packing the data and the address values to be checked into a single value to reduce queue access operations.

EXPERT. Effective and flexible error protection by redundant multithreading (EXPERT) [67] provides microprocessor-wide error detection by including the memory operations inside the SOR. While the previous software-RMT approaches (presented above) suffer from the vulnerable

Table 4. Experimental Characteristics of Compiler-Level RMT Approaches

	SRMT [78]	DAFT [84]	COMET [43]	EXPERT [67]
Compiler	ICC 9.0	LLVM	GCC 4.9	LLVM
Benchmark suite	SPEC CPU2000	SPEC CPU2000	NAS Parallel	MiBench
Environment	Internal CMP simulator	6-core Intel Xeon	Quad-core Corei5	Gem5
Fault coverage	99.79%	99.93%	~SRMT	65xSRMT
Performance overhead	~2.86x	~1.38x	~2.85x	~5x

input replication and output comparison processes, EXPERT solves this problem by duplicating the memory read/write operation values. It assigns a checker thread for the main thread and coordinates them for the memory operations. In the checker thread, it replicates both computational instructions and memory load operations. In order to provide input data coherency for the loaded values between the main and checker thread, EXPERT inserts synchronization points before store operations in the main thread to prevent it from updating a memory location to be read by the checker thread redundantly. It uses a relaxed approach to keep the number of synchronization points at a minimum and puts a majority voting operation for the loads from the volatile memory locations (which can be updated from the outside the application) instead of inserting synchronization code. Both the leading and the checker threads perform three volatile load instruction issues redundantly and check the results by a two-of-three majority voting. The checker thread also checks the store operations by loading the value written by the main thread and comparing it against its own computed value to be stored. The optimization mechanism for communication overhead proposed by EXPERT evaluates memory operation packing to eliminate synchronization points before memory operations. It considers all of the independent successive memory operations as one operation by packing them together and synchronizes the main and checker thread once for the whole pack.

Comparison. Although the experimental study conducted for the compiler-level RMT techniques is based on different settings for different approaches, we put together the main characteristics to guide the future work. We list the experimental setup properties and performance of the techniques in Table 4. While SRMT and EXPERT are tested in simulation environments, DAFT and COMET are applied in a real multicore system. COMET, the only one that addresses the issues for multithreaded applications, uses the NAS parallel benchmark suite in the experimental study. Even though the results are based on different settings (the programs or the architecture), we list the fault coverage and performance overhead to give insight about the potential benefit and cost of each approach. While the related studies provide additional comparative performance results, we only refer to the performance overhead relative to the case without any fault tolerance.

4.4.2 Application Level. A few studies propose application-level RMT techniques providing multithreading as an interface to the programmer.

RedThreads. An Interface for Application-Level Fault Detection/Correction Through Adaptive Redundant Multithreading (RedThreads) [30] recently presented an RMT-based error detection/correction API configurable by the programmer. It provides a set of directives and library routines specified for C/C++ programs. The directives include the number of the redundant thread copies to indicate the level of redundancy and data scoping clauses to define both input variables to be replicated and the output variables to be compared. The programmer may selectively decide the code regions for reliable computation by providing the detection/correction level and the selective variables.

The RedThreads compiler support extends Rolex [27] to enable redundant multithreaded execution. The RedThreads runtime system manages the redundant thread execution specified by

runtime library routines, which are only visible to the compiler. The execution mode may be manipulated adaptively by specifying the redundancy level. For the specified execution phases, the user can focus on the system performance by eliminating redundancy; for the other times reliability can be preferred by enabling redundant execution. The RedThreads runtime system also provides opportunistic fault detection by monitoring the fault events in the system and taking the detection/correction decisions based on the fault frequencies and probabilities. RedThreads also provides runtime optimizations by considering lazy fault detection and thread clustering. The lazy fault detection scheme relaxes the immediate synchronization of the redundant threads. The runtime system buffers the computed values, and a lightweight thread performs the output value comparison while the redundant threads continue their subsequent execution. The thread clustering strategy assigns different priorities to the redundant threads. The runtime system assigns higher priority to one of the redundant threads (the primary thread) for each code region and lower priority to the duplicate threads. Moreover, all the duplicate threads are clustered and scheduled in a single core. The primary threads can be assigned to the remaining cores. In this way, there would be lower interference between the primary and the duplicate threads. Combining these two optimizations, the primary thread can continue its execution without waiting for the others.

The same authors previously proposed an application-level adaptive redundant multithreading based on a simple language level directive [29]. They also introduce the *flexible* spheres of replication concept. When the programmer identifies a code region to be executed redundantly, the compiler translates the code block into flexible spheres of replication. The runtime system manages flexible spheres of replication by turning off the redundant execution according to the fault tolerance state of the system. It collects information about the system state from system logs related to failure events and computes Time Between Events (TBE) and Time Since Last Event (TSLE) for system resources including the processor cores and memory modules. When the TSLE exceeds the TBE, the redundant execution is switched off by considering that the currently executing code region is not vulnerable to errors. This flexibility works on the computation and input redundancy level; namely, if TSLE is higher than TBE for processor cores, the runtime system stops the redundant computation being performed by the computation units; if TSLE is higher than TBE for memory modules (DRAM or cache), the runtime system disables the replication of the inputs stored in memory. The study also explores thread mapping policies by considering the execution of the redundant threads on the same core or on the separate cores. In a follow-up paper [28], the same authors proposed a lazy fault detection where the redundant results are not compared immediately. As the redundant threads compute their results, they write their output values into a buffer space. Then another lightweight thread performs the value comparison.

Chen and Chen [9] propose a programming model based on thread-level redundant execution and majority voting for fault detection and recovery from errors. The leader thread creates several follower threads to execute the protected code region redundantly and checks the results by majority voting. The execution continues according to the majority voting. Moreover, a watchdog thread checks the leader and follower threads. If a follower is not alive, it recreates to avoid an unresponsive system state. The study presents a programming model based on the POSIX thread library by providing additional functions for the proposed fault tolerance approach. It implements *r_pthread_create()* to handle the creation of the follower and watchdog thread, and *checkpoint_work1()* to perform majority voting in the application level as functions using the standard *pthreads* functions. The programmers can call the functions to improve fault tolerance in their programs without worrying about the implementation details.

4.4.3 OS Level. A couple of studies propose operating system support for RMT execution.

Romain. Operating System Support for Redundant Multithreading (Romain) [14] presents an operating system service based on software-implemented RMT for error detection and recovery. It

relies on ECC-protected memory hardware included in commercial off-the-shelf (COTS) systems and deals with the soft errors affecting the functional units.

Romain is implemented inside a system architecture, ASTEROID, which includes a critical core allowing the programs to correctly execute in the presence of errors. The system introduces the *reliable computing base (RCB)* term for the critical core to represent the software essential for transparent replication of applications to provide fault tolerance. In order to minimize the time for RCB execution and the necessary code, the authors try to minimize interaction between the application and the RCB during normal execution. They also consider the operating system kernel as a part of the RCB by including a microkernel to minimize the code size of the RCB implementation. Eventually, the RCB includes the kernel, OS system services, and Romain to provide error detection and recovery for the applications.

Romain replicates the execution at the binary code level transparently and does not need the source code of the replicated application. The master process creates one thread for each replica in a separate address space, and it is responsible for comparing their states during the execution. It has access to every operation of the replicas, and the interactions between the replicated program and the rest of the system are handled by the master. When a replica raises an exception (such as a system call), the master blocks it until the other replicas reach that state. If the master finds that the replicas are identical, the result of the operation (i.e., system call return value) is sent to all replicas by overwriting their thread states. However, shared memory access is not in the control of the master; the replicas may modify it at any time without the master's control. Although the master does not control directly, it still needs to be informed about the state of the shared memory operations. It can be accomplished either by emulating the memory operations, which is too costly, or by executing them in the master's memory, which is vulnerable due to the non-replicated execution. Romain does not implement any of these features, and the master maintains a representation of each replica's address space redundantly. It uses n-way modular redundancy and provides recovery without checkpoint and rollback techniques.

RomainMT. Operating System Service for Replication of Multithreaded Applications (RomainMT) [13] presents the extension of Romain [14] by replicating parallel binary applications. Due to the undeterministic execution of multithreaded applications, the replication may cause unexpected behavior. RomainMT aims to find out a suitable deterministic multithreading approach for replicating the parallel applications. For a multithreaded application, it creates replicas with multiple threads and defines thread groups as the threads doing the same job in different replicas. The master process handles each thread group. RomainMT adds a debug breakpoint instruction at the beginning of the lock function inside the thread library. When a thread requests a lock, this instruction stops the thread and triggers the master process to serialize accesses to the same lock by different thread groups. Thus, the RomainMT master process implements *enforced determinism* through replicated execution by using a debug exception handler. To overcome the performance issues caused by the debug trap and lock handling, RomainMT also proposes *cooperative determinism* so that the replicas agree on the ordering of the lock operations cooperatively instead of using the trap operations. It implements a replication-aware thread library that replaces *pthreads* lock functions. For fault recovery, RomainMT relies on an external error correction mechanism in case of DMR execution or performs majority voting if three or more replicas exist.

4.5 GPU-Based Redundant Multithreading

Recently, GPU-based RMT techniques have been studied to utilize the massive number of parallel threads in general-purpose graphics processing units (GPGPUs). A GPGPU architecture consists of streaming multiprocessors (SMs) providing data-level parallelism. The threads (or work items) perform the execution of the given function (kernel) for a subset of data by being scheduled as

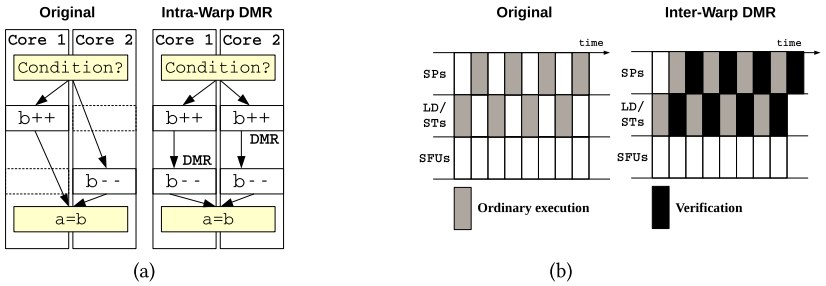

Fig. 9. Examples of Warped-DMR executions [31].

warps (wavefronts) consisting of several threads. A set of threads also form thread blocks (or work groups), and the threads in the same block can communicate through a shared memory [2].

Dimitrov et al. [12] first introduced the software-based redundancy in GPGPUs. Although redundant multithreading is not the main focus of the study, it proposes three main redundancy approaches: R-Naive, R-Scatter, and R-Thread. While R-Naive simply duplicates the computations by executing them one after another, R-Scatter uses underutilized instruction-level parallelism to get the benefit of the VLIW model implemented on specific GPU architectures. On the other hand, R-Thread exploits unused thread-level parallelism. It doubles the number of thread blocks per kernel, and the additional blocks carry out the redundant computations. If the original thread blocks do not utilize the compute units, the redundant copies will increase the reliability by reducing redundancy overhead. Since R-Thread does not seek underutilization conditions explicitly and doubles the execution directly, it does not perform well for the selected benchmarks that already have sufficient thread-level parallelism.

Warped-DMR [31] presents a DMR-based fault detection technique utilizing parallel execution units in GPGPUs. Warped-DMR proposes intrawarp and interwarp DMR to exploit different underutilized resources. A warp includes several threads executing the same code (with a single program counter) and processing different data operands. Some of the threads inside a warp may become idle in case of branch instructions. While some threads take a branch, the others do not take due to data difference. Since all threads use the same program counter, only one set of threads continues its execution; the other threads become inactive. Intrawarp DMR seeks those underutilized execution units and uses them for DMR of the active threads by forwarding an active thread's operands to an inactive thread (Figure 9(a)). To be able to enable forwarding efficiently, intrawarp DMR adds a *Register Forwarding Unit (RFU)* to the microarchitecture. GPGPUs include three different types of execution units: shader processor cores (SPs) for arithmetic operations, LD/ST units for memory operations, and special function units (SFs) for executing complex operations such as sine, cosine, and square root. Since one instruction usually uses only one of those units, the others become idle. Interwarp DMR exploits those idle units to replicate the instructions (Figure 9(b)). Especially, if the instructions using different execution units are interleaved, they can be redundantly executed in the following cycle. For instance, if we have an *add*, *load*, *sub* operation order, interwarp DMR executes the replication of *add* in SP during the original execution of the *load* operation, and the replication of *load* in LD/ST during the original execution of the *sub* operation. For the unverified instructions, interwarp DMR adds a microarchitectural structure called *ReplayQ* to buffer and dequeue the instructions when the execution units become idle.

Wadden et al. [77] present a real-world evaluation of compiler-based GPU RMT by implementing redundant multithreading in OpenCL programs. Since the study is based on an OpenCL compiler, it uses work group and work item terms for RMT implementation instead of the warp and thread

terms in the hardware-based Warped-DMR approach. A work group consists of several work items having their own private memory. Work items from one work group have access to the local memory of their own work group but are prevented from accessing the local memory of another work group. The study proposes Intra-Group RMT and Inter-Group RMT: Intra-Group RMT duplicates the computation between the work items within a work group, while Inter-Group RMT replicates among entire work groups. Intra-Group RMT increases the size of the work group by duplicating each work item (creates work-item pairs). Since work-item pairs can communicate through the work-group local memory, it is possible to use local data share (LDS) for the output comparison. Intra-Group RMT offers two options for the SOR: Intra-Group RMT+LDS, including LDS inside the SOR, and Intra-Group RMT-LDS, excluding LDS from the SOR. In the first option, the data and the load/store operations inside LDS are doubled, while the latter executes the memory operations once by comparing output for store operations. Since the computation is replicated within a wavefront (similar term as the warp in the Warped-DMR approach), the scalar register file (SRF), scalar unit (SU), instruction fetch (IF), and instruction decode (ID), which are shared for an entire wavefront, are not protected. Inter-Group RMT doubles the number of work groups globally and assigns work-item pairs in separate work groups. Since the work items in different work groups are not able to communicate through the local memory, the global application memory needs to be used for the output comparison. Unlike Intra-Group RMT, the entire scalar unit, instruction fetch, and decode units are protected in this approach, since the redundant work-item copies are replicated among different wavefronts. Inter-Group RMT also implements synchronization among the work items due to the lack of synchronization support among the work groups. The study presents a detailed performance and power analysis for both approaches. They show that Inter-Group RMT incurs higher overheads due to the communication through the global memory. While most RMT approaches focus on theoretical aspects and are based on simulations, Wadden et al. present a real-world implementation with the authors mostly from industry having opportunities directly in the field.

Gupta et al. [24] extend the work done in [77] by using cross-lane operations and fingerprinting to decrease the synchronization overhead. Intra-Permute RMT presents an optimization for the Intra-Group RMT approach based on the currently introduced cross-lane operations. The cross-lane operations (permute instructions) offer a register-level communication between different wavefront/warps. Intra-Permute RMT uses those permute instructions, instead of the memfence instruction (instruction for synchronizing memory operations) through local data share, for the synchronization among the work-item pairs. Inter-Fingerprinting RMT proposes an optimization for the Inter-Group RMT approach. In order to reduce the number of synchronization points between the redundant work-item pairs, it uses fingerprinting to combine several synchronization events. Before a store operation, both the leading and trailing work items generate a hash using the memory address, the value to be stored, and the previous hash value. If the threshold is reached for the synchronization event, the leading work item sends the computed hash, which may include the hash of the values from several synchronization points, to the trailing thread. The trailing thread compares its hash value with the received one, and it raises an error in case of a mismatch.

4.6 Power-Efficient Redundant Multithreading

Most of the proposed redundant multithreading approaches have optimistic results in terms of the performance and reliability; however, executing two copies of the same thread on either the same or different cores incurs high power consumption overhead. Therefore, there are several approaches that aim to provide close performance and reliability results to previously proposed studies (such as CRT [46] and CRTR [19]) by minimizing power consumption overhead.

The study proposed by Rashid et al. [54] (named PVA) is one of the first attempts that aims to minimize the power consumption overhead of the provided redundancy. They provide core-level redundancy similar to CRT. In their approach, there is a leader thread running on a lead processor and a trailer thread executing on multiple checker processors. They reduce the power consumption overhead by using two approaches that are the parallel verification process and prefetch effect of the leader thread. Parallel verification is achieved by partitioning the unverified instructions of the leader thread on multiple OoO checker cores. Each checker takes the checkpoint of the corresponding chunk that has register file content and starting PC of the chunk. There is a PCB for the leader thread, which keeps the address and data of the store instructions. The checkers utilize the PCB of the leading core to verify each chunk. Additionally, the trailer thread takes advantage of information flow from the leader thread, which assists the execution of it. Since both the parallel verification and the prefetch effect of the leader thread accelerate the execution of the trailer thread, they apply lower supply voltage for the checker cores compared to the lead core. Using two checker cores run at half frequency and voltage levels provides lower energy consumption compared to CRT without affecting performance.

Madan and Balasubramonian analyze the main factors that affect the power consumption in RMT approaches analytically in [41] and provide a simulation-based study by considering these effects in [42] (named P-CRTR). In these studies, they consider a redundancy model similar to CRTR. They observe that the IPC value of the trailer thread is higher than the leader thread since it has no branch mispredictions or cache misses due to helper structures such as LVQ, BOQ, RVQ, and StB [41]. By taking advantage of the high IPC value for the trailer thread, the authors propose to reduce the power consumption by considering in-order execution, dynamic frequency scaling (DFS), and parallel verification. Based on their observations, the power consumption of an OoO checker core is higher than the in-order checker core, although the OoO core has better IPC values. Executing the leader thread on an OoO core and the trailer thread on an in-order core has better power consumption results since using the in-order core for the trailer thread compensates for the high intercore traffic overhead. Using the DFS technique, the frequency of cores is changed dynamically according to the IPC values of the executing threads. Compared to PVA, they observe that the parallelization of the verification process has little advantage over using an in-order checker core since it reduces the dynamic power but increases the leakage power even if DFS is used.

Subramanyan et al. propose two studies ([68] and [70], named EERE) in which a core-level RMT approach is used similar to CRT. There is a leader thread executing on a primary core and a trailer thread executing on a redundant core. Branch outcomes, load values, and fingerprints are delivered using a shared bus between these cores. In both approaches, they execute redundant core at a lower voltage since the IPC value of the trailer thread is higher than the leader thread due to the helper structures. They utilize a dynamic voltage and frequency scaling (DVFS) technique based on the number of outstanding instructions at helper structures such as BOQ and LVQ. If the number of instructions is higher than the high threshold, they increase the frequency of the redundant core; and if it is less than the low threshold, they decrease the frequency of it. As a difference between these two studies, they shut down the data cache of the trailer thread to reduce leakage power in [68] since the trailer thread does not access the data cache (due to LVQ). Additionally, they monitor the effects of LVQ and BOQ sizes and different high thresholds on performance, energy, and $EnergyxDelay^2$ metrics in [68]. Compared to PVA and CRT, both of their approaches consume less power relative to the nonredundant execution.

In another study of the same research group (named RECVF [69]), the leader thread forwards the results of critical instructions to the trailer thread to accelerate the execution of it apart from the helper structures. In that way, the voltage-frequency level of the trailing core can be decreased to minimize the energy consumption. The critical instructions are a small set of instructions on the

Table 5. Experimental Characteristics of Power-Efficient RMT Approaches

Approach	Power Estimation	Simulator	Power/Energy Results	Performance
PVA [54]	Wattch	SimpleScalar	28% extra energy compared to nonredundant execution	No performance loss compared to nonredundant execution
P-CRTR [42]	Wattch	SimpleScalar	15% better ED^2 compared to CRTR [19]	12% worse performance than CRTR [19]
Subramanyan et al. [68]	First-order power model	SESC	79% average energy savings for redundant core	1.2% performance overhead compared to non-fault-tolerant baseline
EERE [70]	*Not specified*	SESC	1.48 times the energy of non-fault-tolerant baseline	0.3% performance overhead compared to non-fault-tolerant baseline
RECVF [69]	Wattch	SESC	1.26 times the energy of non-fault-tolerant baseline	1.2% performance overhead compared to non-fault-tolerant baseline

critical path that affects the performance of the program significantly (such as mispredicted branch instructions). These instructions are identified at the leading core by using various heuristics. The *fanoutN* heuristic, which identifies an instruction as critical if the output of it is used by at least *N* in-flight instructions, shows the best performance. Since forwarding the results of the critical instructions causes a slack between the leader and trailer threads, this overhead is compensated by executing the trailing core at a low voltage-frequency level. The slack may change during the execution of different program phases; therefore, the voltage-frequency level of the cores is arranged dynamically at runtime. In their approach, the IPC values of both leader and trailer threads are monitored during regular time periods, and the frequency of cores is arranged based on the scaled ratio of IPC values. According to their results, by forwarding a small set of instructions, high performance results can be achieved with significant power gains.

Comparison. Although the studies mentioned above use different hardware configurations on different simulators, we summarize the power/energy and performance results reported by them in Table 5 to give an insight on the outcomes. The studies use similar benchmark suites (i.e., SPEC CPU2000) by considering different baseline architectures such as non-fault-tolerant architecture, CRT, or CRTR. Among the studies that consider the non-fault-tolerant architecture as baseline, it is observed that the PVA consumes 28% extra energy than the baseline, the EERE consumes 1.48 times the energy of the baseline (where CRT consumes 1.99 times and PVA consumes 1.79 times the energy of the baseline, respectively), and RECVF consumes 1.26 times the energy of the baseline (where CRT consumes 1.52 times and PVA consumes 1.32 times the energy of the baseline, respectively) based on the reported experimental results. Since RECVF forwards only critical instructions to the trailing thread, it might be required infrequent bandwidth usage might be required; therefore, it might present fewer energy consumption results. Apart from these studies, the study proposed by Subramanyan et al. [68], which uses a simple first-order power model, reduces the dynamic power in the redundant core by 79% based on their results. In P-CRTR, they show that the overhead of the redundant core is 10% of the leader core's power dissipation by using in-order checker and OoO leader cores. Additionally, their approach show 15% better ED^2 compared to CRTR for the multithreaded workloads. The proposed techniques improve the power/energy consumption by presenting comparable performance results; however, they do not evaluate the fault coverage of the proposed systems. Their conclusion about the fault coverage is based on the assumptions. Most of the proposed studies model the power/energy consumption by using

estimation tools; however, these tools might be limited with design parameters and make the estimations inaccurately.

4.7 Thread Mapping for Redundant Multithreading

There are several studies that try to improve the performance of RMT techniques by efficient mapping of threads on many-core architectures.

Kalayappan and Sarangi [32] perform different thread mapping techniques in which the leader and checker threads might be scheduled across different SMT cores. They assume that the number of the leader and checker threads is more than the number of available cores; therefore, the leader (or checker) thread of an application can be paired with a checker (or leader) thread of a different application. They define an IPC value for each core that is the summation of IPC values of threads executing on it. Then, they sort the cores based on the IPC values in increasing order. When they map a single thread, they select the first core in the list that has an empty slot to map the thread onto it. The target is simply to map a high IPC thread with a low IPC one to balance the load among the cores. At certain time periods (named *epochs*), they calculate the IPC values of the current mapping and reschedule the threads based on their activity for the next period. To map a set of threads, they apply three algorithms. In *Pinned Leaders*, the leader threads are bound to a random set of cores statically. At each epoch, the checker threads are sorted based on their IPC values in decreasing order and they are bound to the selected cores (based on the criteria given above) dynamically. In *Unpinned Leaders*, all threads are treated as the same and they are scheduled based on their activity at each epoch. In *Unpinned - All Leaders First*, a hybrid of the previous two approaches is applied in which threads are bound to the cores dynamically based on their IPC values; however, it gives priority to the leader threads since they need more resources than the checkers. Among these scheduling techniques, *Unpinned - All Leaders First* shows the best performance compared with the other proposed mapping techniques since the leader threads, which have high demand on resources, are prioritized over the checker ones. They also report that the proposed mapping algorithm shows better performance than SRT [56] and CRT [46]. Since the leader and checker threads of an application might be bound to different cores, the communication overhead of the proposed approach is decreased by utilizing a network-on-chip (NoC) architecture and proposing a set of enhancements such as selective forwarding of cache lines and forwarding filters to reduce the network traffic. As forwarding filters, the *recently forwarded buffer (RFB)* is used by the leader thread to eliminate sending the same cache lines multiple times to the trailer thread by keeping the recently sent cache lines in the buffer. The *lines to be forwarded buffer (LFB)* contains the actual cache lines to be sent to the trailer thread based on RFB entries. The proposed technique provides a complete fault coverage based on their assumptions.

Chen et al. [8] propose a Mixed Redundant Threading (MRT) approach, which is a mixture of SRT and CRT under limited time and core constraints. In their TMR-based RMT approach, two replicated threads can be executed on two cores in parallel as in CRT, while one of the redundant threads runs on the same core with the original thread sequentially as in SRT. They determine the redundancy level of tasks as SRT, CRT, or MRT, then use Federated Scheduling to schedule tasks on the cores. In Federated Scheduling, the tasks are divided into two groups as the high-utilization tasks and the low-utilization tasks, where the former ones are scheduled first and the latter ones are scheduled on the remaining cores. Based on this scheduling, a task can be mapped onto a dedicated task executing one core or multitasking cores. For scheduling of the high-utilization tasks, list scheduling is used where the parallel subtasks of the task are mapped on the cores dedicated to it. For scheduling of the low-utilization tasks, they use the remaining cores by using rate-monotonic priority assignment. In order to assign the redundancy level for each task by considering the task timing constraint, they use a dynamic programming approach. Furthermore,

they select the optimal redundancy level for task stages by adapting the dynamic programming approach since each task stage may require different redundancy levels under Federated Scheduling. In that fine-grained optimization approach, a stage of a task can be executed with TMR, DMR, or no redundancy. Their approach outperforms the greedy approach in terms of reliability penalty, which is calculated as the probability of an error having a visible effect on the output during the execution of a specific task, under the limited number of cores constraint.

Pouyan et al. [52] propose a reliability-aware CMP architecture in which online Architectural Vulnerability Factor (AVF) [47] estimation is utilized for threads to activate a redundancy mechanism under reliability and performance constraints. They estimate the AVF of a thread by considering the vulnerability of hardware components allocated to that thread at runtime. If the AVF value of the thread exceeds a predefined threshold value, then a redundant copy of it starts to execute. Since providing full redundancy has bottlenecks in terms of performance and energy consumption, their approach utilizes partial redundancy by activating/deactivating the redundancy mechanism based on the estimated AVF value. In their adaptive redundancy approach, a dynamic scheduling technique is used to map threads on the cores to improve system throughput. They apply thread migration among the cores to balance the load and improve performance in addition to a traditional round-robin-based scheduling technique. If a core has a high vulnerability value having threads with a high Thread Vulnerability Factor (TVF), the thread with low TVF executing on that core is migrated to another core since a redundant copy of the thread with high TVF starts to execute on the original core. Here, the migrated core is the one that has the highest throughput value among the cores. Therefore, load balancing and throughput improvement can be achieved with the proposed scheduling and thread migration technique compared to the static scheduling approaches.

Efficient task mapping techniques for the RMT approach are proposed in several studies by considering the process variation effect [7, 15, 35]. Process variation results from a manufacturing variability that may cause significant frequency changes among the cores and leads to performance asymmetry in the system even though they have similar architectural characteristics. It also affects the system reliability negatively with aging factors.

Kriebel et al. [35] propose a variation-aware thread mapping technique that also considers RMT management (enabling/disabling RMT) for a set of tasks and selection of different compiled versions of the tasks under the process variation and aging effect. Their aim is to decrease the aging rate of the slow cores to balance the aging profile of the system in RMT mode. The core allocation is decided based on the aging profile and performance constraints of the application tasks. To enable this, the cores are sorted based on their decreasing frequency levels, and the high-performance cores are selected first to compensate for the aging rate of slow cores at minimum. Since the application tasks with high duty cycles can speed up the aging of the cores, the low-activity tasks are mapped on the slowest cores. Additionally, they provide an aging-aware RMT management in which RMT of low-vulnerability threads is disabled at runtime under the constraint of a tolerable error rate. The aging profile (based on duty cycles) and vulnerability level analysis (based on fault injection experiments) are performed for each different compiled version and a version is selected under the reliability-performance tradeoff. First, they select a compiled version based on the duty cycles of it to minimize aging of the cores while meeting the system performance constraint. Then, RMT of the tasks is disabled for high resilient ones while meeting the system vulnerability constraint. Therefore, they balance the aging profile of the system with an aging-aware task mapping and core allocation technique.

Chen et al. [7] propose a soft-error-resilient application execution technique that considers both task mapping and task execution mode of applications. They assume that the cores have different frequencies due to process variation, aging, or architectural design. In their approach, the

homogeneous tasks are mapped onto the heterogeneous performance cores efficiently by using the Hungarian Algorithm to find an optimal solution with minimal reliability penalty in polynomial-time complexity. In the case of heterogeneous tasks, some of which require TMR-based RMT technique and some do not need redundancy (NR) technique, they prefer to map TMR-based RMT tasks onto the low-frequency cores first since the reliability penalty of these tasks is negligible. Then, the NR tasks are mapped onto the high-performance and resilient cores by using the Hungarian Algorithm to meet their deadline miss rate constraints. They also present an iterative execution mode adaptation technique that decides the execution mode of the tasks as TMR-based RMT tasks or NR tasks. In that approach, all tasks are considered as NR tasks initially, and the execution mode of the tasks with maximum reliability penalty are upgraded to the TMR-based RMT tasks gradually until there is no more improvement. Additionally, they enhance their mapping solution by using the XY-routing algorithm to estimate the communication overhead for the data dependencies. Based on their results, the proposed application execution technique improves the system reliability compared to the greedy algorithm, which maps TMR-based RMT tasks onto the high-performance cores.

Dong et al. [15] propose a variation-aware scheduling technique that considers intrapair and interpair variations among redundant threads and application sensitivity to these variations. While intrapair variation considers the performance asymmetry within a core pair executing the leader and trailer threads, interpair variation is related with the performance asymmetry in different core pairs executing the leader threads. They observe the application sensitivity to the variations mentioned above and model the problem as a 0-1 programming problem. Based on this, they propose a scheduling algorithm to maximize system throughput by minimizing these variations for a case in which the number threads is equal to the number of cores. Their thread-level redundancy approach is similar to CRT and does not present a recovery mechanism.

The studies mentioned above consider thread mapping techniques for the RMT approach in different aspects. While some of the studies give different priorities to the redundant threads for the efficient mapping of them to the available cores [8, 32], the thread migration approach is used in another study to improve the throughput of the system with load balancing [52]. A couple of studies propose efficient thread mapping techniques for heterogeneous cores resulting from the process variation or aging [7, 15, 35]. In some of the studies, the redundancy level of the executing threads is changed at runtime and they utilize efficient scheduling techniques by considering both reliability and performance aspects at the same time [7, 8, 35, 52]. We do not present the experimental results of these studies since their evaluation metric is different for each case, such as the system throughput, performance, aging profile, or reliability.

5 CONCLUSIONS AND RESEARCH DIRECTIONS

In this article, we survey redundant multithreading techniques proposed throughout the last 20 years. We review the basic work done for soft error fault detection and correction by using thread-level redundancy on different design spaces. Since it is possible to perform time redundancy both on hardware and software, the studies are proposed over a broad range of implementations. We cover the main single-core, multicore, and GPU designs by explaining the common features. Due to the performance overhead of redundant execution, partial RMT techniques, with an acceptable error coverage and lower execution times, propose instruction elimination or adaptive redundancy. Furthermore, we include the more recent extensions to give insight into future trends in the area. For software approaches, RMT-based fault tolerance techniques implemented on an operating system, compiler, or application level are explained as an alternative to hardware-level approaches. As can be seen in the related sections, both hardware and software techniques share major patterns and similar architectures such as providing multiple execution streams for

redundant execution. While hardware approaches propose architecture-level modifications to handle RMT execution, software approaches try to hide the redundant execution implementation from the system by dealing with the redundant threads in the higher abstraction levels. Since the power efficiency becomes another concern in redundant fault tolerance systems, we analyze the studies optimizing power consumption in a separate section by explaining the main points and differences between them. RMT introduces redundant thread execution, and assigning those threads on redundant execution units gains importance in terms of performance overhead. Therefore, we also examine thread mapping strategies proposed for RMT systems.

We develop a taxonomy to help potential users find a suitable method for their requirement and to guide researchers planning to work in this area. Our classification table offers a large set of RMT studies classified according to different criteria. People who want to work on a specific architecture or on a specific implementation level can find the most relevant work for the identified design choice.

Since computer architecture research evolves rapidly, RMT methods may be improved by architectural innovations. RMT implementation utilizing GPU cores that support a high number of threads, especially during long latency global memory access times, may be a good choice. Currently, a limited number of RMT studies have been applied on such architectures. Another popular acceleration architecture, the Google Tensor Processing Unit (TPU), may be evaluated as an RMT alternative, as the reliability is becoming a design criterion for this architecture [83].

Although existing RMT approaches solve the reliability problem successfully, they might have overheads in terms of additional memory usage, performance degradation, and energy consumption. To overcome such overheads, novel machine-learning techniques might be utilized for activating redundant multithreading automatically and efficient mapping of redundant threads to the available cores. Furthermore, the activation threshold of RMT can be adjusted depending on the target application domain. On the other hand, multithreaded workloads have been rarely utilized in existing RMT approaches. Providing cost-effective solutions to the synchronization problem among the redundant threads may be a good alternative research direction.

We believe that RMT-based fault tolerance techniques will be adapted as the parallel systems evolve, and our survey will guide the system developers for this adaptation by providing relevant work in the area. Moreover, application-specific solutions may be developed by using the existent work to find a way for optimized redundant execution with high error coverage.

REFERENCES

[1] 2003. Cisco 12000 Single Event Upset Failures Overview and Work Around Summary. Retrieved from http://www.cisco.com/en/US/ts/fn/200/fn25994.html. Cisco Systems.

[2] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers. 2018. *General-Purpose Graphics Processor Architectures*. Morgan and Claypool Publishers.

[3] N. Abu-Ghazeleh, J. Sharkey, D. Ponomarev, and K. Ghose. 2006. Exploiting short-lived values for low-overhead transient fault recovery. In *Workshop on Architectural Support for Gigascale Integration (ASGI'06)*.

[4] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (Jan. 2004), 11–33.

[5] M. T. Bohr and I. A. Young. 2017. CMOS scaling trends and beyond. *IEEE Micro* 37, 6 (2017), 20–29.

[6] S. Borkar. 2005. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro* 25, 6 (Nov. 2005), 10–16.

[7] K. H. Chen, J. J. Chen, F. Kriebel, S. Rehman, M. Shafique, and J. Henkel. 2016. Task mapping for redundant multithreading in multi-cores with reliability and performance heterogeneity. *IEEE Transactions on Computers* 65, 11 (2016), 3441–3455.

[8] K. H. Chen, G. V. Der Bruggen, and J. J. Chen. 2018. Reliability optimization on multi-core systems with multi-tasking and redundant multi-threading. *IEEE Transactions on Computers* 67, 4 (2018), 484–497.

[9] Y. Chen and P. Chen. 2016. A software-based redundant execution programming model for transient fault detection and correction. In *International Conference on Parallel Processing Workshops (ICPPW'16)*.

[10] J. A. Clark and D. K. Pradhan. 1995. Fault injection. *Computer* 28, 6 (1995), 47–56.
[11] M. Daňhel, F. Štěpánek, and H. Kubátová. 2017. Dependability prediction involving temporal redundancy and the effect of transient faults. In *2017 Euromicro Conference on Digital System Design (DSD'17)*. 360–363.
[12] M. Dimitrov, M. Mantor, and H. Zhou. 2009. Understanding software approaches for GPGPU reliability. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU'09)*.
[13] B. Döbel and H. Härtig. 2014. Can we put concurrency back into redundant multithreading? In *International Conference on Embedded Software (EMSOFT'14)*.
[14] B. Döbel, H. Härtig, and M. Engel. 2012. Operating system support for redundant multithreading. In *International Conference on Embedded Software (EMSOFT'12)*.
[15] J. Dong, L. Zhang, Y. Han, G. Yan, and X. Li. 2009. Variation-aware scheduling for chip multiprocessors with thread level redundancy. In *15th IEEE Pacific Rim International Symposium on Dependable Computing*. 17–22.
[16] B. Fechner. 2006. A result propagation scheme for redundant multithreaded systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems and Applications (PDPTA'06)*. 64–69.
[17] J. Fu, Q. Yang, R. Poss, C. R. Jesshope, and C. Zhang. 2013. On-demand thread-level fault detection in a concurrent programming environment. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'13)*. 255–262.
[18] J. Fu, Q. Yang, R. Poss, C. R. Jesshope, and C. Zhang. 2014. A fault detection mechanism in a data-flow scheduled multithreaded processor. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE'14)*. 1–4.
[19] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. 2003. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture, 2003.* 98–109.
[20] M. A. Gomaa and T. N. Vijaykumar. 2005. Opportunistic transient-fault detection. In *International Symposium on Computer Architecture (ISCA'05)*.
[21] R. Gong, K. Dai, and Z. Wang. 2008. Transient fault recovery on chip multiprocessor based on dual core redundancy and context saving. In *2008 9th International Conference for Young Computer Scientists*. 148–153.
[22] R. Gong, K. Dai, and Z. Wang. 2008. Transient fault tolerance on chip multiprocessor based on dual and triple core redundancy. In *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*. 273–280.
[23] B. Greskamp and J. Torrellas. 2007. Paceline: Improving single-thread performance in nanoscale CMPs through core overclocking. In *16th International Conference on Parallel Architecture and Compilation Techniques*. 213–224.
[24] M. Gupta, D. Lowell, J. Kalamatianos, S. Raasch, V. Sridharan, D. Tullsen, and R. Gupta. 2017. Compiler techniques to reduce the synchronization overhead of GPU redundant multithreading. In *Design Automation Conference (DAC'17)*.
[25] P. Hazucha, T. Karnik, J. Maiz, S. Walstra, B. Bloechel, J. Tschanz, G. Dermer, S. Hareland, P. Armstrong, and S. Borkar. 2003. Neutron soft error rate measurements in a 90-nm CMOS process and scaling trends in SRAM from 0.25-/spl mu/m to 90-nm generation. In *IEEE International Electron Devices Meeting, 2003.* 21.5.1–21.5.4.
[26] M. Y. Hsiao. 1970. A class of optimal minimum odd-weight-column SEC-DED codes. *IBM Journal of Research and Development* 14, 4 (July 1970), 395–401.
[27] S. Hukerikar and R. F. Lucas. 2016. Rolex: Resilience-oriented language extensions for extreme-scale systems. *Journal of Supercomputing* 72, 12 (2016), 4662–4695.
[28] S. Hukerikar, K. Teranishi, P. C. Diniz, and R. F. Lucas. 2014. An evaluation of lazy fault detection based on adaptive redundant multithreading. In *High Performance Extreme Computing Conference (HPEC'14)*.
[29] S. Hukerikar, K. Teranishi, P. C. Diniz, and R. F. Lucas. 2014. Opportunistic application-level fault detection through adaptive redundant multithreading. In *International Conference on High Performance Computing and Simulation*.
[30] S. Hukerikar, K. Teranishi, P. C. Diniz, and R. F. Lucas. 2018. RedThreads: An interface for application-level fault detection/correction through adaptive redundant multithreading. *International Journal of Parallel Programming* 46, 2 (2018), 225–251.
[31] H. Jeon and M. Annavaram. 2012. Warped-DMR: Light-weight error detection for GPGPU. In *International Symposium on Microarchitecture (MICRO'12)*.
[32] R. Kalayappan and S. R. Sarangi. 2015. FluidCheck: A redundant threading-based approach for reliable execution in manycore processors. *ACM Transactions on Architecture and Code Optimization* 12, 4 (Dec. 2015), 55:1–55:26.
[33] I. Koren and S. Y. H. Su. 1979. Reliability analysis of n-modular redundancy systems with intermittent and permanent faults. *IEEE Transactions on Computers* C-28, 7 (July 1979), 514–520.
[34] E. Koser, K. Berthold, R. Kumar Pujari, and W. Stechele. 2016. A chip-level redundant threading (CRT) scheme for shared-memory protection. In *International Conference on High Performance Computing Simulation*. 116–124.
[35] F. Kriebel, S. Rehman, M. Shafique, and J. Henkel. 2016. ageOpt-RMT: Compiler-driven variation-aware aging optimization for redundant multithreading. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC'16)*.
[36] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar. 2007. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 317–326.

[37] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO'04)*.
[38] D. Lyons. 2000. Sun Screen. Retrieved from http://members.forbes.com/global/2000/1113/0323026a.html. Forbes Magazine.
[39] Y. Ma and H. Zhou. 2006. Efficient transient-fault tolerance for multithreaded processors using dual-thread execution. In *International Conference on Computer Design (ICCD'06)*.
[40] N. Madan and R. Balasubramonian. 2006. Exploiting eager register release in a redundantly multi-threaded processor. In *2nd Workshop on Architectural Reliability (WAR-2'06), Held in Conjunction with MICRO-39*.
[41] N. Madan and R. Balasubramonian. 2006. A first-order analysis of power overheads of redundant multi-threading. In *2nd Workshop on System Effects of Logic Soft Errors (SELSE-2'06)*.
[42] N. Madan and R. Balasubramonian. 2007. Power efficient approaches to redundant multithreading. *IEEE Transactions on Parallel and Distributed Systems* 18, 8 (Aug. 2007), 1066–1079.
[43] K. Mitropoulou, V. Porpodas, and T. M. Jones. 2016. COMET: Communication-optimised multi-threaded error-detection technique. In *International Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES)*.
[44] S. S. Mukherjee. 2008. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers, San Francisco, CA.
[45] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. 2005. The soft error problem: An architectural perspective. In *11th International Symposium on High-Performance Computer Architecture*. 243–247.
[46] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. 2002. Detailed design and evaluation of redundant multi-threading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. 99–110.
[47] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO'03)*.
[48] J. V. Neumann. 1956. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, Vol. 34. Princeton University Press, 43–99.
[49] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. 1996. The case for a single-chip multiprocessor. *SIGPLAN Notices* 31, 9 (Sept. 1996), 2–11.
[50] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. 2006. 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*.
[51] F. Pouyan, A. Azarpeyvand, S. Safari, and S. M. Fakhraie. 2015. Reliability aware simultaneous multithreaded architecture using online architectural vulnerability factor estimation. *IET Computers and Digital Techniques* 9, 2 (2015), 124–133.
[52] F. Pouyan, A. Azarpeyvand, S. Safari, and S. M. Fakhraie. 2016. Reliability aware throughput management of chip multi-processor architecture via thread migration. *Journal of Supercomputing* 72, 4 (2016), 1363–1380.
[53] M. W. Rashid and M. C. Huang. 2008. Supporting highly-decoupled thread-level redundancy for parallel programs. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 393–404.
[54] M. W. Rashid, E. J. Tan, M. C. Huang, and D. H. Albonesi. 2005. Exploiting coarse-grain verification parallelism for power-efficient fault tolerance. In *14th International Conference on Parallel Architectures and Compilation Techniques*.
[55] V. K. Reddy, S. Parthasarathy, and E. Rotenberg. 2006. Understanding prediction-based partial redundant threading for low-overhead, high-coverage fault tolerance. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*.
[56] S. K. Reinhardt and S. S. Mukherjee. 2000. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA'00)*.
[57] E. Rotenberg. 1999. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS'99)*.
[58] D. Sánchez, J. L. Aragón, and J. M. García. 2008. Evaluating dynamic core coupling in a scalable tiled-CMP architecture. In *International Workshop on Duplicating, Deconstructing, and Debunking (WDDD'08) in conjunction with ISCA'08*.
[59] D. Sánchez, J. L. Aragon, and J. M. Garcia. 2009. Extending SRT for parallel applications in tiled-CMP architectures. In *International Symposium on Parallel and Distributed Processing (IPDPS'09)*.
[60] D. Sánchez, J. L. Aragón, and J. M. García. 2009. REPAS: Reliable execution for parallel applications in tiled-CMPs. In *Euro-Par 2009 Parallel Processing*. Springer, Berlin, 321–333.
[61] D. Sánchez, J. L. Aragón, and J. M. García. 2012. A fault-tolerant architecture for parallel applications in tiled-CMPs. *Journal of Supercomputing* 61, 3 (Sept. 2012), 997–1023.
[62] E. Schuchman and T. N. Vijaykumar. 2007. BlackJack: Hard error detection with redundant threads on SMT. In *International Conference on Dependable Systems and Networks (DSN'07)*.

[63] J. Sharkey, N. Abu-Ghazeleh, D. Ponomarev, K. Ghose, and A. Aggarwal. 2006. Trade-offs in transient fault recovery schemes for redundant multithreaded processors. In *International Conference on High-Performance Computing (HiPC'06)*.

[64] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. 2002. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of International Conference on Dependable Systems and Networks*.

[65] T. Siddiqua and S. Gurumurthi. 2009. Balancing soft error coverage with lifetime reliability in redundantly multithreaded processors. In *IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*. 121–130.

[66] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. 2006. Reunion: Complexity-effective multicore redundancy. In *International Symposium on Microarchitecture (MICRO'06)*.

[67] H. So, M. Didehban, Y. Ko, A. Shrivastava, and K. Lee. 2018. EXPERT: Effective and flexible error protection by redundant multithreading. In *Design, Automation and Test in Europe Conference and Exhibition*.

[68] P. Subramanyan, V. Singh, K. K. Saluja, and E. Larsson. 2009. Power efficient redundant execution for chip multiprocessors. In *Proceedings of the 3rd Workshop on Dependable and Secure Nanocomputing in Conjunction with DSN*. 1–6.

[69] P. Subramanyan, V. Singh, K. K. Saluja, and E. Larsson. 2010. Energy-efficient fault tolerance in chip multiprocessors using critical value forwarding. In *IEEE/IFIP International Conference on Dependable Systems Networks*. 121–130.

[70] P. Subramanyan, V. Singh, K. K. Saluja, and E. Larsson. 2010. Energy-efficient redundant execution for chip multiprocessors. In *Proceedings of the 20th Symposium on Great Lakes Symposium on VLSI*. 143–146.

[71] P. Subramanyan, V. Singh, K. K. Saluja, and E. Larsson. 2010. Multiplexed redundant execution: A technique for efficient fault tolerance in chip multiprocessors. In *Design, Automation Test in Europe Conference Exhibition*. 1572–1577.

[72] P. Subramanyan, V. Singh, K. K. Saluja, and E. Larsson. 2011. Adaptive execution assistance for multiplexed fault-tolerant chip multiprocessors. In *2011 IEEE 29th International Conference on Computer Design (ICCD'11)*. 419–426.

[73] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. 2000. Slipstream processors: Improving both performance and fault tolerance. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[74] D. M. Tullsen, S. J. Eggers, and H. M. Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *International Symposium on Computer Architecture (ISCA'95)*.

[75] N. J. van Eck and L. Waltman. 2014. Visualizing bibliometric networks. In *Measuring Scholarly Impact*, Y. Ding, R. Rousseau, and D. Wolfram (Eds.). Springer, 285–320.

[76] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. 2002. Transient-fault recovery using simultaneous multithreading. In *International Symposium on Computer Architecture (ISCA'02)*.

[77] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron. 2014. Real-world design and evaluation of compiler-managed GPU redundant multithreading. In *International Symposium on Computer Architecture*.

[78] C. Wang, H. Kim, Y. Wu, and V. Ying. 2007. Compiler-managed software-based redundant multi-threading for transient fault detection. In *International Symposium on Code Generation and Optimization*.

[79] N. J. Wang and S. J. Patel. 2005. ReStore: Symptom based soft error detection in microprocessors. In *International Conference on Dependable Systems and Networks (DSN'05)*.

[80] N. J. Wang, A. Mahesri, and S. J. Patel. 2007. Examining ACE analysis reliability estimates using fault-injection. *SIGARCH Computer Architecture News* 35, 2 (June 2007), 460–469.

[81] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. 2004. Techniques to reduce the soft error rate of a high-performance microprocessor. In *31st Annual International Symposium on Computer Architecture*.

[82] J. Yu, D. Jian, Z. Wu, and H. Liu. 2011. Thread-level redundancy fault tolerant CMP based on relaxed input replication. In *2011 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT'11)*. 544–549.

[83] J. J. Zhang, T. Gu, K. Basu, and S. Garg. 2018. Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator. In *36th VLSI Test Symposium (VTS'18)*.

[84] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August. 2010. DAFT: Decoupled acyclic fault tolerance. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*.

Received August 2018; revised November 2018; accepted December 2018