PRISM: Strong Hardware Isolation-based Soft-Error Resilient Multicore Architecture with High Performance and Availability at Low Hardware Overheads

HAMZA OMAR and OMER KHAN, Universty of Connecticut

Multicores increasingly deploy safety-critical parallel applications that demand resiliency against soft-errors to satisfy the safety standards. However, protection against these errors is challenging due to complex communication and data access protocols that aggressively share on-chip hardware resources. Research has explored various temporal and spatial redundancy-based resiliency schemes that provide multicores with high soft-error coverage. However, redundant execution incurs performance overheads due to interference effects induced by aggressive resource sharing. Moreover, these schemes require intrusive hardware modifications and fall short in providing efficient system availability guarantees. This article proposes PRISM, a resilient multicore architecture that incorporates strong hardware isolation to form redundant clusters of cores, ensuring a non-interference-based redundant execution environment. A soft error in one cluster does not effect the execution of the other cluster, resulting in high system availability. Implementing strong isolation for shared hardware resources, such as queues, caches, and networks requires logic for partitioning. However, it is less intrusive as complex hardware modifications to protocols, such as hardware cache coherence, are avoided. The PRISM approach is prototyped on a real Tilera Tile-Gx72 processor that enables primitives to implement the proposed cluster-level hardware resource isolation. The evaluation shows performance benefits from avoiding destructive hardware interference effects with redundant execution, while delivering superior system availability.

CCS Concepts: • **Computer systems organization → Multicore architectures**; • **Hardware → Redundancy**; **System-level fault tolerance**;

Additional Key Words and Phrases: Soft-errors, hardware interference, strong isolation

ACM Reference format:
Hamza Omar and Omer Khan. 2021. PRISM: Strong Hardware Isolation-based Soft-Error Resilient Multicore Architecture with High Performance and Availability at Low Hardware Overheads. *ACM Trans. Archit. Code Optim.* 18, 3, Article 31 (June 2021), 25 pages.
https://doi.org/10.1145/3450523

1 INTRODUCTION

Semiconductor technology miniaturization has instigated a serious reliability challenge in microprocessors, making them susceptible to transient faults, also known as *soft-errors*. These faults

This research was supported by the National Science Foundation under Grant No. CNS-1929261. This research was also supported in part by the Semiconductor Research Corporation (SRC).
Authors' address: H. Omar and O. Khan (corresponding author), University of Connecticut, 371 Fairfield Way, Storrs, CT 06269; emails: {hamza.omar, khan}@uconn.edu.

1544-3566/2021/06-ART31
https://doi.org/10.1145/3450523

can potentially cause programs to execute in an incorrect fashion by causing machine code bit-flips, altering signal transfers or stored values. Multicore processors are thriving and commonly deployed in numerous real-time environments [14, 32] to execute a variety of safety-critical applications (e.g., path planning, motion detection) [36], requiring high soft-error resiliency for safety-criticality [43] yet high performance for their timing constraints. However, multicores introduce numerous challenges for soft-error protection due to their complex communication and memory access protocols that aggressively share on-chip resources [11, 24, 39].

Researchers have explored a diverse set of software and hardware-based resiliency schemes for protection against soft-error perturbations. The effectiveness of any given resiliency scheme predominantly depends on three major factors: (1) *availability*: the capability of the resiliency solution to protect against all types of faults (such as deadlocks, crash/hangs, etc.) and provide user(s) with the final output as quickly as possible; (2) *hardware/software intrusiveness*: the hardware and/or software overheads incurred to enable protection against all soft-error faults; and (3) *efficiency*: the ability of the system to enable soft-error protection without hurting the scheduled application's execution performance. Several software-based resiliency solutions [8, 19, 21, 28] have been developed to improve the coverage of **silent data corruptions (SDCs)**. However, these solutions fall short in providing high system availability in presence of a crash, deadlock, and live-lock situations as such events require a system-level reboot. Additionally, these schemes are notorious for their performance and software-support overheads. Hardware-based resilience schemes [35, 39, 45] generally enable redundancy mechanisms, such as **thread-level redundancy (TLR)** [4, 11, 17, 27] or **n-modular redundancy (nMR)** [41, 44]. These solutions perform checkpoint-based temporal or spatial redundant execution, where the check-pointing mechanism either incurs high performance overheads or require complex per-core hardware support for assuring soft-error protection. Regardless, a crash or hang requires these schemes to go through a system-level recovery, hurting the overall system availability. Examples of such hangs/crashes include cache coherence protocol lockups due to soft-error strikes. Even though these cache structures are generally protected via ECC and CRC techniques, a soft-error strike can still cause a message to get lost, resulting in the intended destination to continuously wait and cause the directory to lock-up. Moreover, these schemes provoke extensive performance overheads (~4× average overhead is reported in Reference [44]), primarily due to sharing of hardware resources (such as on-chip caches, network, and off-chip memory) resulting in interference among applications being redundantly executed [11]. To address the performance efficiency challenge, researchers have explored selective resiliency mechanisms for an application such that performance and error coverage demands are both fulfilled simultaneously [13, 18, 26, 29, 38, 40]. Selective resiliency trades off either program *accuracy* [26, 38, 40] or *vulnerability* [18, 29] with resilience overheads. However, these solutions incur intrusive hardware changes to enable redundant execution and system availability. Keeping the aforementioned challenges in mind, the objective is not only to provide multicore systems with performance efficient soft-error protection, but devise such a resiliency solution that also delivers high system availability while incurring non-intrusive hardware modifications.

Inspired by previously explored fault-tolerance concepts, such as sphere-of-replication [47] and fault-containment domains [6, 42], this article proposes PRISM, a multicore resiliency architecture that applies the principle of *strong hardware isolation*[1] to create equally sized spatial clusters of cores in a multicore. The application instances execute in their respective clusters in a **dual-modular redundancy (DMR)** fashion without sharing any data or metadata. Each cluster in PRISM is similar to a containment domain [42], which is provided with its own set of dedicated core-level resources, i.e., core-pipeline, private–shared caches and TLBs, on-chip network

[1]No process must be allowed to access dedicated hardware resources of concurrently executing process(es) [25].

routers, and memory controllers. A per-cluster, light-weight software kernel is implemented that is responsible for creating isolated memory regions (at page granularity) for the application(s) deployed in the respective cluster. Later, this kernel pins/maps these memory regions to clusters' dedicated memory controllers (physically connected to separate memory modules). The kernel then pins/maps each cluster's data (at cache-line granularity) to the dedicated private and shared caches of the dedicated set of physical cores. Naturally, for all clusters, the shared cache misses are forwarded to dedicated on-chip memory controllers. Last, PRISM utilizes deterministic X-Y and Y-X on-chip network routing protocol to disable all means of communication between clusters, i.e., no inter-core coherence or memory traffic is allowed to the cross the cluster boundary. This allows PRISM to enable high performance efficiency for DMR, as these strongly isolated clusters execute independently and do not experience resource sharing (interference) effects, leading to improved system availability and resource utilization. Moreover, this independent execution of clusters ensures high system availability, because even if one cluster faces network (e.g., coherence protocol) lock-ups due to a soft-error strike, the other independent cluster continues with its execution. In terms of the design's intrusiveness, the proposed cluster formulation in PRISM requires reconfigurable selection logic at the shared cache and memory controller levels to form isolated clusters. Such hardware capabilities are also being made available in modern commercial processors [9]; thus, PRISM requires light-weight hardware and software support.

To improve performance, PRISM enables an adaptive strongly isolated clustering capability. At a given time instance, the kernel dynamically reconfigures the system to enable a single cluster of all available core-level resources, or multiple spatially distributed redundant clusters of equally distributed resources. Consequently, this supports an efficient selective resiliency scheme that temporally utilizes the cluster re-sizing capability to guarantee both resiliency and efficiency. For selective resiliency, the kernel first identifies crucial and non-crucial iterations of a given iterative decision algorithm using the approach proposed in Reference [22]. The crucial iterations are executed in the *resilience mode* by spatially executing two instances of the algorithm in two strongly isolated clusters of cores. Before executing non-crucial iterations in the *non-resilience mode*, the kernel intervenes and reallocates the core-level resources by invoking an operation that remaps application data structures from the two clusters to a single cluster of cores comprising all available resources. An application executes crucial iterations in *resilience mode*, while non-crucial iterations with integrated software bounds-based checkers are executed in the *non-resilience mode*.

PRISM is prototyped on a real TileraTile-Gx72 multicore processor [46] because of its hardware level capabilities to enforce the formation of strongly isolated clusters of cores [23–25]. For a set of task-parallel iterative applications, PRISM's redundant execution is shown to improve performance by 12% over state-of-the-art TLR scheme that utilizes all available multicore parallelism. Moreover, the selective resilience capability is shown to further improve performance from to ~43% over the baseline TLR scheme, while ensuring a program output accuracy loss of <1%. Most importantly, alongside the performance advantages, PRISM also guarantees high system availability while incurring low hardware overheads.

2 RELATED WORK

Prior software- and hardware-based resiliency schemes are summarized in Table 1 in terms of performance, hardware overhead, multicore applicability, and soft-error coverage. The soft-error coverage shows four different soft-error effects: (1) *crash*: the system stops functioning correctly and exits; (2) *deadlock*: the system resources become unavailable to applications, leading to a system-level hang or lock-up; (3) *livelock*: the applications constantly change their resource allocation states with none of them progressing; and (4) *SDC*: the state of the application perturbs but remains undetected.

Table 1. Comparisons to Related Works

Resilience Schemes	Performance Overhead	Hardware Overhead	Soft-Error Coverage (System Availability)				Selectively Trading-Off	Multicore Applicability
			Crashing	Deadlock	Livelock	SDC		
Inst. Duplication [8]	Moderate	None	Low	Low	Low	High	None	No
Invariant Check [28]	Moderate	None	Low	Low	Low	High	None	No
TLR/DMR [4, 27, 41, 44]	Moderate	Low	Moderate	Moderate	Moderate	High	None	Yes
HaRE [39]/FluidCheck [11]	Low	Moderate	Moderate	Moderate	Moderate	High	None	Yes
Khudia et.al. [13]	Low	None	Low	Low	Low	High	Accuracy	No
Omar et.al [26]	Low	High	Moderate	Moderate	Moderate	High	Accuracy	Yes
dTune [29]	Moderate	Moderate	Moderate	Moderate	Moderate	Moderate	Vulnerability	Yes
RASTER [18]	Moderate	Moderate	Moderate	Moderate	Moderate	Moderate	Vulnerability	No
PRISM (Proposed)	Low	Low	High	High	High	High	Accuracy	Yes

"High" means that the error can be detected and recovered, "Moderate" implies the scheme is effective for some faults but not all, and "Low" shows that it is unlikely the system is protected under such faults. Better if viewed with colors.

The software-based resiliency solutions (cf. Table 1: first block), such as instruction duplication [8, 21], and invariant checking [19, 28] have been developed for single core processors to improve the coverage of SDCs. However, they fall short in providing high system availability in presence of a crash, deadlock, and live-lock. Additionally, these schemes are notorious for their performance and software-support overheads. Prior works [35, 45] have also explored *symptom-based* soft-error detection/recovery mechanisms, but they provide low soft-error coverage, since they rely on coarse-grain detectors, such as fatal-traps, hangs, panics, and so on. Under hardware-based resilience schemes [35, 39, 45], the solutions enable redundancy mechanisms, such as TLR [4, 11, 17, 27] or nMR [41, 44] to provide soft-error protection. For instance, prior work [44] focuses on applying DMR on a multicore (GPU) setting, where it redundantly executes two copies of the same application, and delivers high soft-error coverage by performing cross checks in a duplicated thread. However, as shown in Table 1, such schemes introduce added hardware complexity for providing holistic soft-error coverage and fall short in assuring high availability in case of system crash/hang. Additionally, these schemes incur extensive performance overheads (~4× average overhead is reported in Reference [44]) due to (1) sharing of hardware resources that induces interference among redundant applications and (2) reduced hardware's thread-level parallelism opportunities for applications to exploit. Another work, HaRE [39] (cf. Table 1: second block) proposes a resilience scheme for multicores that performs check-point-based temporal (time-sliced) redundant execution, and relies on a per-core re-execution mechanism to support recovery from detected errors. However, every crash or deadlock requires a system-level reboot to start the recovery mechanism, by means of which the system availability suffers. Moreover, HaRE requires hardware modifications to the cache coherence protocol for protection against soft-error lock-ups, as each core needs protected access to data and synchronization variables. Similarly to HaRE, another recent work, FluidCheck [11] proposes a resiliency scheme that relies on temporal redundant execution. However, FluidCheck is also expected to suffer from cache coherence protocol hangs/lock-ups due to a soft-error strike, essentially suffering from low system availability.

To improve the performance efficiency of resilience schemes, researchers have also explored selective resiliency mechanisms for an application, such that performance and error coverage demands are both fulfilled [18, 26, 29, 38, 40]. These schemes are also listed in Table 1 (the third block). For instance, certain works [18, 29, 30] obtain efficiency by providing high resiliency for high vulnerability code; however, they tradeoff performance with soft-error coverage (*vulnerability*). Similarly, SWIFT-R [31] uses the concept of selective hardening to design a mitigation technique that trades off reliability and resilience overheads for flexibility and reduced overheads. Contrarily,

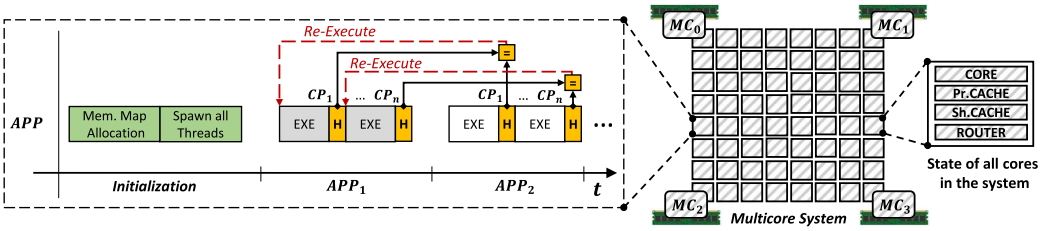

Fig. 1. The temporal dual-modular redundancy scheme is shown on a multicore, where each instance of the application shares and utilizes all available hardware resources of the system.

some solutions [13, 26, 38, 40] assure improved performance by employing selective resiliency that trades off program *accuracy* with resilience overheads. For instance, a recently proposed work [26] exploits *performance–accuracy*, where it bifurcates an application into crucial/non-crucial regions and enables redundancy only for protecting crucial regions, whereas non-crucial regions are partially protected via software resiliency mechanisms. Even though this work assures impeccable soft-error coverage (cf. Table 1), its effectiveness is debatable as it requires intrusive hardware changes to the core-pipeline and private caches for re-execution, and also demands a resilient cache coherence protocol [3] for functional correctness. Additionally, due to the interference effects and moderate soft-error coverage, these works provide low system availability.

The proposed PRISM architecture is inspired by fault-tolerance concepts of sphere-of-replication [47] and fault-containment domains [6, 42]. However, it applies the principle of *strong hardware isolation* [25] to create interference-free equally sized spatial clusters of cores to execute an application using dual-modular redundancy. Forming domains using the strong isolation concept not only enables high-end performance, but also enables high system availability, i.e., protection against system hangs and crashes.

PRISM also supports a selective resiliency capability that is enabled by adaptively reconfiguring the core-level resources in the proposed strongly isolated clusters. The selective resiliency scheme is shown to provide both high resiliency and efficiency for iterative decision algorithms that offer opportunities to tradeoff program output accuracy. To the best of our knowledge, no prior work has adopted strong hardware isolation primitives to formulate a highly available and performance efficient resilient multicore architecture that incurs low hardware overheads.

3 BACKGROUND AND MOTIVATION FOR PRISM ARCHITECTURE

This article primarily focuses on soft-error resiliency in multicore architectures that comprise of numerous cores, per-core private–shared cache hierarchy, per-core interconnection routers, and multiple memory controllers connected to their respective memory channel modules (DIMMs). The *dual-modular redundancy* (TLR/DMR) approach is selected due to it its multicore applicability and low hardware overhead requirements. In DMR, two identical instances of an application are executed concurrently on the multicore system. Upon completion, the output data structures for both instances are passed over to each other for verification to ensure correct execution. This redundant execution is conducted either *temporally* or *spatially* with both incurring different intricacies in their designs.

3.1 Temporal Dual-Modular Redundancy for Resiliency

In temporal dual-modular redundancy schemes (such as TLR [11, 20, 27, 33]), execution of the two application instances is time-sliced on the multicore, where these instances time-share hardware resources for performance. Figure 1 shows a general timeline schema for performing dual-modular redundancy on two identical instances of the application (*APP*) in a time-sliced (temporal) fashion.

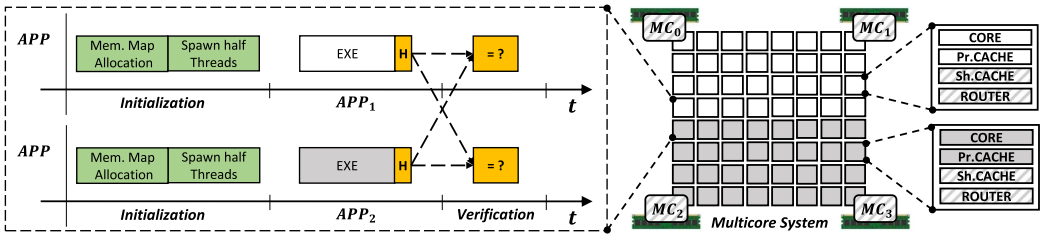

Fig. 2. The spatial dual-modular redundancy scheme is shown on a multicore, where each instance of the application executes spatially on the system, but large stateful memory resources still remained shared.

First, the system performs initialization steps (such as thread spawning and memory allocations), and then starts the execution of application instances APP_1 and APP_2. During the execution of APP_1, either epoch-based [4] or idempotency-based [16] checkpoints ($CP_1, \ldots, CP_n$) are generated to capture the state of the application for proper roll-back in case of a soft-error upset. Upon completion of APP_1's time window, it context switches out of the system and allows the other application instance APP_2 to start with its execution. During execution, the APP_2 instance captures the checkpoint (CP_1) at the same epoch or idempotency point, and compares it with APP_1's checkpoint CP_1 state. If the checkpoint values match, then the application instances continue their execution. However, if they mismatch, then the APP_2 context switches and APP_1 re-executes from the point of failure. These comparison steps continue till the last checkpoint value (CP_n), and APP_2 terminates when all checkpoints pass their comparison check.

Temporal isolation for dual-modular redundancy provides the processor with high soft-error resiliency against SDC, as shown in Table 1. However, such schemes do not ensure strong protection against crashes and deadlocks, since they allow all cores to access the directory and off-chip memory without any protection, potentially causing cache coherence protocol lock-ups in case of a soft-error strike. Temporal redundancy schemes also exhibit low *system availability*, as each crash or hang requires a system-level reboot before letting the system continue from the saved checkpoint state. Another aspect that requires attention is that all hardware resources (core pipeline, on-chip networks, cache hierarchy, and memory controllers) are time-shared across the temporally executing applications instances (cf. Figure 1). This indeed amplifies the utilization of each of these shared hardware resources. However, it causes the application instances to compete for utilizing them. This competition in turn induces destructive interference across the application instances, which further exacerbates performance [24]. For example, when instances compete for same shared cache resources, it increases the stress on available cache capacity, resulting in reduced data locality.

3.2 Spatial Dual-Modular Redundancy for Resiliency

The spatial dual-modular redundancy schemes rely on spatial execution of two copies/instances of the same application. The timeline of spatial dual-modular redundancy on a multicore is shown in Figure 2, where two identical instances of the application (*APP*) execute spatially. Similarly to the temporal scheme, the system first goes through necessary initialization steps. However, the thread spawning is done such that each instance's threads are spawned on half of the cores. Each application instance (APP_1) then executes on its dedicated cores, where it shares the large stateful memory resources (i.e., shared caches/TLBs, and off-chip memory) with the other application instance (APP_2). During the concurrent execution, each application instance captures its own system state (i.e., collecting check-points or computing hash of the output), which are later used for verification and roll-back purposes as done for the temporal redundancy scheme (cf. Section 3.1).

Similarly to temporal redundancy schemes, spatial dual-modular redundancy also provides systems with high soft-error protection against SDCs (cf. Table 1). However, this scheme also does not provide protection against deadlocks, essentially yielding low *system availability* due the necessary system-level reboot in case of soft-error induced crash or hang. In addition to these shortcomings, spatial redundant execution also experiences significant performance degradation, e.g., a prior work [44] has shown average dual-modular redundancy overheads of ~4×. Unlike temporal redundancy schemes, this degradation is not only attributed to destructive interference due to aggressive hardware resource sharing (cf. Section 3.1) but also due to the loss of thread-level parallelism, which further aggravates performance [24].

3.3 Motivation for PRISM

The aforementioned sections list out the potential drawbacks for the temporal and spatial DMR schemes. Various optimizations have been adopted to make these schemes efficient in terms of performance, such as dynamically using resources of the SMT cores for checking the results of other threads [11]. However, at the fundamental level, these schemes still suffer from performance loss due to destructive interference on shared hardware resources. Moreover, they incur intrusive hardware modifications for soft-error resiliency, yet enable low system availability against crashes and hangs.

Given these challenges, the goal of PRISM architecture is to provide safety-critical systems with exquisite soft-error protection (high system availability), while ensuring minimal hardware modifications for simplicity, and efficient performance for satisfying the real-time constraints. The performance of aforementioned schemes suffer due to redundant application instances competing for the shared hardware resources, resulting in destructive interference effects. The proposed PRISM architecture is inspired by fault-tolerance concepts of sphere-of-replication [47] and fault-containment domains [6, 42], and applies the principle of *strong hardware isolation* [25] to create interference-free equally sized spatial clusters of cores to execute an application using dual-modular redundancy. In PRISM, each cluster of cores is provided with its own set of hardware resources, such that each cluster's code and data remains mapped to it, and never interferes with the other cluster's resources. Strong isolation ensures that the network (coherence) messages never overlap the cluster boundaries, essentially requiring no modification to the cache coherence protocol and hardware. Consequently, PRISM offers impeccable *system availability*, as one of the two strongly isolated clusters always remains *active* even if the other cluster hangs as a consequence of a soft-error strike. To provide further improvements, PRISM also incorporates a selective resiliency mechanism, whilst satisfying error coverage and application output accuracy demands simultaneously.

4 THE PROPOSED PRISM ARCHITECTURE

This section provides a detailed overview of the proposed PRISM architecture. Section 4.1 highlights the essential steps for ensuring strong hardware isolation for interference-free execution of redundant application instances in a multicore setting, whereas Section 4.2 provides details on the proposed selective multicore resiliency in PRISM.

4.1 Ensuring Non-Interference for Dual-Modular Redundancy

Figure 3 shows the PRISM architecture for performing interference-free dual-modular redundancy, where the two identical application instances execute in their respective clusters, i.e., instance APP_1 is executed on $Cluster_1$, while instance APP_2 executes on $Cluster_2$. This section outlines the core-level formation of two strongly isolated clusters in PRISM. Note that traditional spatial clustering approaches generally suffer from *load-imbalance*. However, at the fundamental level,

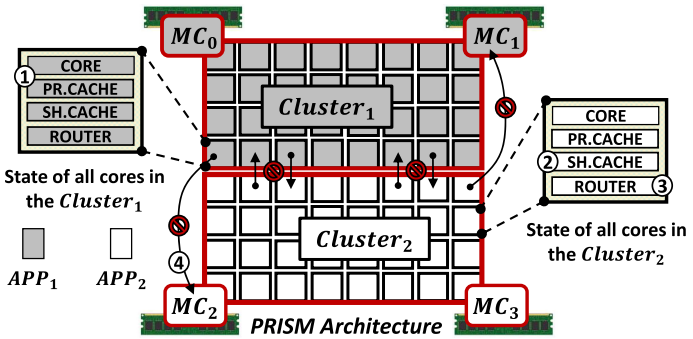

Fig. 3. The multicore PRISM architecture with strong hardware isolation primitives for non-interference.

dual-modular redundancy requires the two clusters to be exactly equal in terms of their size and resources, as the two application instances are completely identical. Therefore, PRISM does not suffer from load-imbalance challenge. Each cluster in PRISM implements a light-weight software kernel (or even an operating system) that enforces the strong isolation capabilities across the cluster. This kernel executes temporally alongside the user application(s) within its own cluster. The strong isolation capabilities enforced by the kernel are discussed next.

4.1.1 Isolation of Core-Pipeline and Private Caches. The PRISM architecture forms two clusters of cores that spatially execute the redundant instances of the same application, i.e., APP_1 and APP_2. Each cluster is assigned a set of cores, and the respective process threads are pinned to its assigned cluster. For strong isolation, cores are allocated such that clusters do not overlay with each other, i.e., $\{CPU_{Cluster_1} \cap CPU_{Cluster_2}\} \in \emptyset$. Naturally, providing application instances with dedicated sets of cores also results in spatially partitioning the private cache and TLB resources for both application instances (cf. Figure 3: ①).

4.1.2 Isolation of Shared Cache Resources. Multicores deploy last-level cache that is logically shared, but physically distributed as cache slices across all cores. By default, an entire memory page is interleaved across all shared caches at cache line granularity, essentially forming interference channel(s). To avoid this situation, it is important to keep each application's data within its own set of shared cache slices (clustered together). Therefore, PRISM adopts a *local homing* policy, where an entire memory page (or data structure) is mapped to a single shared cache slice. Data replication in last-level cache is disabled for the case of inter-cluster data sharing. This ensures that each access to a shared cache slice in a given cluster is made by the application executing in that cluster. Essentially, this limits clusters from accessing each others shared cache slices (cf. Figure 3: ②). It is noteworthy that spatial partitioning schemes for the last-level cache are now being adopted in various commercial processors [9].

4.1.3 On-Chip Network Isolation. For each cluster, the network traffic needs to be routed in such a way that all network packets remain within the boundary of the cluster. Thus, a deterministic network routing protocol supporting bidirectional routing [37] (allows both X-Y and Y-X routing) is employed to enable isolation of network traffic. The X-Y routing with two-dimensional (2D) mesh network topology recognizes each router by its coordinates (X, Y), and transmits packets first in the X direction followed by the Y direction, whereas in Y-X routing, the packets traverse the network in the Y direction first. In a square floor plan, rows of cores are assigned to each cluster with respective memory controller(s) on the outside edges. However, if cores within a row

are allocated among two clusters, then the Y-X routing gets triggered to ensure that packets do not drift outside the cluster boundary (cf. Figure 3: ③).

4.1.4 Isolation of Off-Chip Memory. Multicore processors deploy multiple memory controllers, each connected to their physically isolated memory channels (DIMMs). Generally, all memory pages of applications are interleaved (hashed) across all memory controllers to maximize for memory bandwidth. However, shared buffers/queues in the memory controllers become a reason for causing destructive interference. To ensure chip level isolation, all means of data accesses must be isolated among clusters. Therefore, PRISM statically partitions the on-chip memory controllers across the two clusters, as shown in Figure 3: ④. The memory pages of any given application are mapped in such a way that they are only accessible from their own dedicated on-chip memory controller(s). This allows each cluster to access an independent physical channel, a memory bank, and a memory row. Naturally, the last-level cache misses of any given application are routed to dedicated memory controller(s) that map their respective memory pages. For example, the $Cluster_1$'s accesses to off-chip DDR memory components are realized by forwarding its traffic to MC_0 and MC_1 memory controllers (cf. Figure 3). Note, this work primarily focuses on DMR; therefore, PRISM forms only two clusters. To implement *n-modular redundancy* with n strongly isolated clusters, the number of memory controllers (and their respective physical memory channels) present in the system determine the limit for n to ensure ensure complete physical isolation across the data for n clusters.

4.1.5 Output Verification. For dual-modular redundancy in PRISM, the redundant application instances APP_1 and APP_2 execute on their respective clusters, utilizing their dedicated isolated hardware resources. Upon completion, each cluster computes a *64-bit XOR hash* of the application output(s) to capture its own cluster/system state. For ensuring correct execution, these computed hash values need to be compared and verified (cf. Sections 3.1 and 3.2). Indeed, this requires both clusters to share their hash values with each other. The shared memory protocol is generally adopted for this verification step, where a shared memory inter-process communication region (accessible to both application instances) is created to exchange respective hash values. Similarly, this verification can also be done using *in-hardware core-to-core messages*, which have been explored by numerous multicore processor architectures [2, 34, 46]. We have empirically observed that both protocols incur similar verification overheads. However, PRISM adopts the latter approach as it enables a redundant communication method for verifying the output hashes, as compared to each cluster's data accesses. In PRISM, each cluster compares its locally computed hash value with the received hash value of the other cluster. If *hash values* from both clusters do not match, then a roll-back mechanism re-executes the redundant application instances. Otherwise, the application is allowed to proceed with executing new inputs, or terminate.

4.2 Adaptive Selective Resiliency for Improved Performance

The PRISM architecture mitigates performance degradation from interference of shared hardware resources due to redundant execution. However, resilience with redundancy leads to loss of parallelism, and hence performance. To improve efficiency, prior works [26, 38, 40] have explored avenues to selectively apply resiliency schemes on various safety-critical applications by partitioning them into crucial and non-crucial regions. These selective resiliency schemes have been shown to improve the performance and efficiency of systems by trading off resiliency overheads with program output accuracy. The primary reason for achieving the benefits is because resiliency and performance demands for safety-critical systems vary based on the surrounding conditions and constraints. For example, an unmanned aerial vehicle would require high resiliency guarantees while maneuvering a mission through harsh weather. However, during normal conditions,

providing weak resilience against soft-errors is acceptable, as long the system output converges to a satisfactory and acceptable result.

In this context, PRISM also enables an adaptive selective resilience capability that provides an efficient soft-error resiliency solution. The system enables dynamic reconfiguration between a single cluster and two spatially redundant clusters of equally distributed core-level resources. However, this capability is devised while keeping a non-intrusive resiliency solution in mind. In this selective resiliency solution, the crucial and non-crucial regions of any given iterative decision algorithm are first identified. The crucial regions are executed in the *resilience mode* by spatially executing two instances of the algorithm in two strongly isolated clusters of cores (cf. Section 4.1). Before executing non-crucial regions in the *non-resilience mode*, the kernel intervenes and reallocates the hardware resources from the two clusters to a single cluster of cores. For ensuring functional correctness of the underlying application in the *non-resilience* mode, the kernel implements a smart *bound checking* mechanism to verify that the computed results are within certain bounds.

4.2.1 Selective Resilience for Iterative Parallel Applications. A variety of parallel iterative applications are deployed in real-time systems, such as **single source shortest path (SSSP)** for path planning [5]. These parallel applications iterate over a shared output data structure for (monotonic) convergence, based on the ordering constraints [1]. Numerous applications have been shown to benefit from selective resilience by partitioning them into crucial and non-crucial regions [26, 38]. Moreover, a recent work [22] has shown that the initial iterations for output convergence of such iterative parallel application are more sensitive in defining the output accuracy of the application. Therefore, the idea of selective resilience is employed here by executing certain initial iterations of the parallel application in *resilience* mode (that incorporates *dual-modular redundancy* aspects described in Section 4.1), whereas the remaining iterations execute in the clear without strong resiliency guarantees (referred to as *non-resilience* mode in this article). The process of application bifurcation is driven based on the conditions surrounding the system that define the fault injection rate, the application's output accuracy, and naturally, the crucial iterations. Upon determining the bifurcated regions, the bound checkers are added to the application to ensure functional and output correctness. This bound checking process requires understanding of the application's iterative behavior and later, the bound checkers are added based on this understanding. In this article, these modifications are made at the higher-level application loops using a similar strategy adopted in Reference [22]. Indeed, modifications at finer-grain levels of the loops can potentially open more performance optimization opportunities, but they are expected to become more intrusive, since they will require profiling support [26].

As shown in Figure 4, an efficient selective resiliency mechanism is proposed and adopted in PRISM for iterative algorithms that utilize the cluster reconfiguration capability to guarantee both resiliency and efficiency. The scheme initializes two clusters, $Cluster_1$ and $Cluster_2$, with half of the core-level resources allocated to each cluster. Initially, for the *resilience* mode, the redundant application instances (APP_1 and APP_2) execute on their respective clusters, utilizing their respective dedicated (half) system resources for a certain number of application iterations (say, **X**%). Executing these **X**% of iterations results in computing an intermediate output (*∂-Output*). To ensure correct resilient execution, *∂-Output* verification is done for both clusters by comparing the 64-bit hash of the outputs (cf. Section 4.1). Upon completion of *resilience* mode, PRISM activates its resource reconfiguration capability and switches the system state to the *non-resilience* mode for executing remaining **N-X**% iterations (cf. Figure 4: ① and ②). In the *non-resilience* mode, the $Cluster_1$ is provided with all available core-level resources to exploit thread-level parallelism for performance (cf. Figure 4: ②). This is done by re-spawning APP_1's threads on all system cores and remapping the memory pages of that process from initially allocated home slices to all available

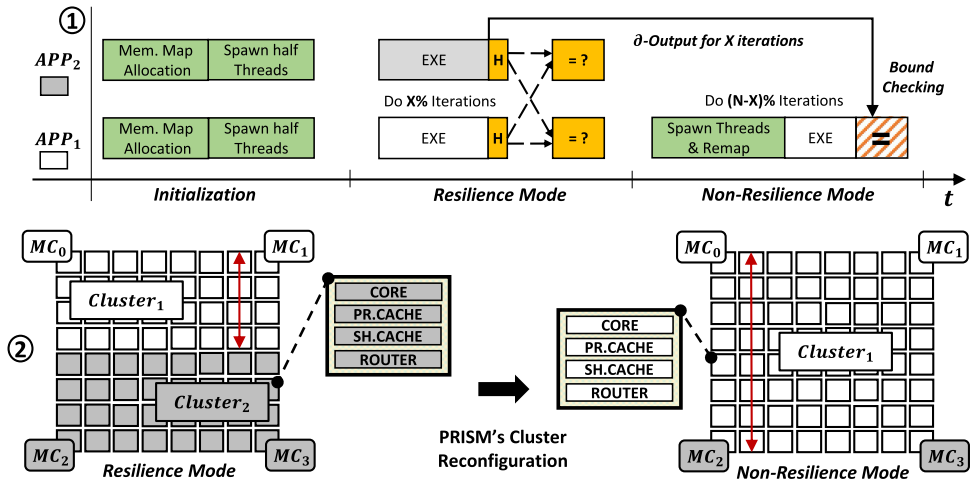

Fig. 4. Design flow of PRISM's selective resilience scheme is shown in ①. APP_1 and APP_2 execute X% crucial iterations in the *resilience* mode. ② shows APP_1 cluster reconfiguration for efficient execution of the remaining (N-X)% non-crucial iterations in the *non-resilience* mode.

shared cache slices. However, $Cluster_1$ still executes APP_1 with half of the memory controller resources (cf. Figure 4: ②). The allocations for the on-chip memory controllers are not modified, since they incur substantial overheads due to re-initialization of all structures.

To achieve output correctness, a *bound checking* mechanism is added in the *non-resilience* mode to verify that the computed results are within certain bounds (cf. Figure 4: ①). The primary idea of adopting *bound checkers* is that if the specified bounds of the program output are not satisfied, then this indicates a soft-error perturbation (potentially a *silent data corruption*) and the computed final result in no longer acceptable. Therefore, PRISM uses the *∂-Output* (computed at the end of the *resilience* mode) as a back up to commit it to the final output. The selective resiliency mechanism is primarily evaluated using iterative parallel applications; however, this scheme is not limited to only such applications. Evaluating the selective resiliency mode of PRISM for more application domains is left as part of future work. For applying selective resiliency on iterative applications, the process of adding bound checking varies from one application to the other, and the *bound checkers* are designed based on the unique properties of these iterative parallel applications. The key insight here is that these parallel applications iterate over a common output data structure and generally converge to the final output in a monotonically increasing or decreasing fashion [1]. The selective resiliency of PRISM takes advantage of this monotonic convergence property to decide the output bounds, independent of whether the final output is known or unknown beforehand. The target iterative applications considered in this work and their respective bound checking mechanisms are discussed next.

4.2.2 Target Iterative Parallel Applications. This article adopts seven real-time graph analytic applications [1], and one mission-planning algorithm from advanced driver-assistance system [24]. The graph applications are provided with California road network graph as an input [7], whereas the mission-planning algorithm is provided with radar image stream. Each application is discussed in the context of its adoption for PRISM.

(1) *The SSSP* algorithm finds shortest paths from a source to all vertices in a graph. A distance array in shared memory, $D[\,]$ maintains the distances for each vertex from the source vertex. These distance values are initialized at a large number, and these values monotonically reduce as the

algorithm progresses. Upon completion of the algorithm, it uses the final converged distance values to return the *path-cost* as its final output. If a soft-error strike perturbs the distance values to become larger or smaller during the execution of the algorithm, then these perturbation effects indeed reflect in the final computed *path-cost*. Certainly, we have strong confidence on the *∂-Output* (i.e., list of $D[\,]$ values and *path-cost*), as it is computed via redundancy checks in the *resilience mode*. Therefore, *∂-Output* is used as a pivot to define the upper and lower bounds for the correctness of the final *path-cost*. As each vertex relaxation step in SSSP must always monotonically decrease the $D[\,]$ values with increasing iteration counts, the final computed $D[\,]$ values (and the *path-cost*) must never exceed *∂-Output*. Hence, the upper bound is set to the *∂-Output* values for the *path cost*. Contrarily, for the lower bound, the final $D[\,]$ values (*path-cost*) are again compared against the *∂-Output*, such that these computed values decrease no more than a predetermined percentage (say, 10%) of the *∂-Output*. If any of these defined bound checks fail, then this indicates a soft-error effect resulting in an unacceptable output and PRISM uses *∂-Output* as a back up to commit it to the final output.

(2) ***A* Shortest Paths (A-STAR)*** relies on a heuristic to prune the work done by the traditional SSSP algorithm by not visiting all vertices of the input graph. Similarly to SSSP, the heuristic distances from the source vertex are also tracked using a monotonically decreasing distance array, $D[\,]$ to return a final *path-cost* as an output. Given their extensive similarities, A-STAR algorithm employs exactly similar *bound checking* mechanisms as devised for SSSP.

(3) ***Minimum Spanning Tree (MST)*** uses a priority queue and checks for keys based on the input graph to update critical sections. These checks on keys decrease monotonically, which are tracked using the shared array, $Key[\,]$. The MST algorithm returns the *minimum cost* of the tree as an output, similar to the *path-cost* provided by SSSP and A-STAR. Given the computational and behavioral similarities, the *bound checker* employed for MST follows identical bounds (using *minimum cost*) as adopted for SSSP and A-STAR.

(4) ***Breadth-First Search (BFS)*** starts from a source vertex, and searches vertices in a graph using *edge-first* method. As edges are searched, the distance of the search increases from the source vertex. This distance increases monotonically, and tracked using a shared distance array, $D[\,]$. As it final output, this algorithm uses these $D[\,]$ values to return the number of successfully searched vertices in the provided graph as it output. Similarly to SSSP, *∂-Output* is used as a reference point for defining the upper and lower bounds to the satisfy the number of correctly searched vertices of the input graph. As $D[\,]$ values of BFS monotonically increase with increasing iteration count, the final computed $D[\,]$ values (used to extract the number of searched vertices) must never be lower than the *∂-Output*. Therefore, the lower bound is set to the *∂-Output* values for returning the final searched graph vertices. For the upper bound, the final $D[\,]$ values are compared against *∂-Output* values, such that these computed values must never escalate more than a predefined percentage of *∂-Output* values.

(5) ***Connected Components (CC)*** labels edges to a component in an input graph. These vertex's components in the input graph increase monotonically, and are tracked using the shared array, $CC[\,]$. Like BFS, CC returns a per-vertex connected component count as its output, and thus employs exactly a similar *bound checker* as adopted for BFS.

(6) ***Graph Coloring (COLOR)*** implements vertex coloring based on their saturation degree. These vertex colors increase monotonically, and are tracked using the shared array, $Color[\,]$. This algorithm returns the number of unique identifiers required to color each vertex in the input graph as it output, which is identical to the BFS and CC algorithms. Thus, COLOR employs exactly similar *bound checking* primitives as employed for BFS and CC.

(7) ***Page-Ranking (PR)*** algorithm compute ranking of pages in a given graph using a probabilistic model that specifies the likelihood of a person visiting a certain page (vertex). The algorithm provides per-vertex probability value, *PR*[ ] in the range of 0 and 1, as its final output. Since the lower and upper bounds for the output (i.e., 0 and 1, respectively) are inherently defined in PR, these bounds are directly employed within the *bound checker*.

(8) ***Artificial Bee Colony (ABC)*** algorithm iterates over the autonomous vehicle's radar image stream to compute the velocity, acceleration, and distance values using matrix computations. It returns an output vector, *V*[ ], which satisfies predefined upper and lower bound values for each of the aforesaid vehicle's characteristics. Similarly to PR, ABC algorithm also comprises of predefined lower and upper bounds, which directly form the *bound checker*.

4.2.3 Accuracy-Centric Fault-Injection Analysis. Clearly, the selective resilience capability in PRISM enables dynamic cluster re-sizing to exploit performance benefits; however, it leads to lower output accuracy. To measure the impact of exploiting this tradeoff on the final output accuracy of the iterative parallel applications, this work also performs a fault injection analysis to test the effectiveness of PRISM's bound checking mechanism. Upon identifying the crucial and non-crucial iterations to execute in respective modes of PRISM, the entire application is subjected to program-level fault injection. Priors works [10, 15] have proposed various methods to inject faults at both hardware and software-level. The difference between hardware and software methods mainly lies in the fault injection points they have access to, the cost they adhere, and the intensity level of that perturbation. This article primarily focuses on program-level (or software-level) fault injection modeling, and follows the similar accuracy analysis approach as proposed in Reference [48]. The soft-errors can impact an application's data being processed with a noise phenomenon anytime, anywhere in the application. Due to the unpredictable and uncontrollable nature of soft-errors, random errors are introduced during the execution of non-crucial iterations via software-level fault injection. In this work, both realistic (single error) and aggressive error rates are applied to build up the confidence. However, this article focuses more on single soft-error perturbations happening during the program's execution, as it reflects a more realistic scenario [12].

These errors are exposed to the instruction's *op-codes* and *operands*. To model a silent-data corruption, the operand data is perturbed in such a way that variable(s) of a random program instruction (belonging to the non-crucial program iterations) are exposed to random values (determined based on the data type). For instance, if variable(s) belonging to the randomly selected program instruction of any given non-crucial iteration has an *integer* data type, then any random value from the range of −2147483646 to +2147483647 (minimum and maximum values for an *integer* data type) is committed to that variable(s). The impact of injecting such errors is reflected in the application's final output in the sense that it becomes larger and/or smaller, essentially impacting the application's output accuracy. Contrarily, to model a crash/hang, the instruction *opcode* is perturbed with random values in a similar fashion; however, this injection results in the application to experience faults and/or exceptions.

The application specific output structures and their respective employed *bound checkers* are listed in Section 4.2.2. The accuracy metric is defined as "the absolute percentage difference between the *golden reference output* (program's output with error-free execution) and the output observed when the error was injected." Following this definition, the accuracy for SSSP, A-STAR, and MST is the percentage difference between the minimum path/tree costs. The accuracy for COLOR is the percentage difference in the number of unique identifiers (*CC*[ ]) required to color an input graph, whereas the percentage difference in the vehicle's output vector (*V*[ ]) determines the accuracy for ABC. The remaining algorithms, i.e., BFS, CC, PR, return per-vertex-based values in their output arrays *D*[ ], *CC*[ ], and *PR*[ ], respectively. Thus, accuracy measurements for

such algorithms is done by computing the percentage difference in values at a per-vertex basis. However, measuring accuracy using a single metric is insufficient, as not all vertices' values for such algorithms are pivotal for the real-time system. For example, finding the ranks of all web-pages (vertices) is not always important when PR is employed in a real-world setting. Thus, the effective accuracy now becomes a function of the number of *web-pages of interest*. In such a case, remaining web-pages/vertices are irrelevant and must not be considered for measuring the accuracy. Therefore, for such algorithms, different accuracy metrics (such as monitoring top 10 or 20 vertices and computing their percentage difference for accuracy measurement) are considered to model different queries issued by an analyst/programmer/consumer using the system.

Note, the crucial iteration count varies from one application (and its respective deployment environment) to the other. For example, a system deployed in a harsh environment would require a higher number of crucial iteration count (more resilient execution) for better accuracy and system availability, alongside reasonable performance. In contrast, if that same system is deployed in a normal environment, then the crucial iteration count could be dropped (by ample amount) for capturing acceptable accuracy, and high performance and availability. Thus, determining the number of crucial iterations for selective resilience highly depends on the surrounding environmental conditions and must factor into account the system's constraints and accuracy demands.

4.3 Key Features of PRISM

4.3.1 Performance Efficiency. Researchers have shown that redundancy frameworks for resiliency generally suffer from degraded performance due to sharing of hardware resources, and/or loss of core-level parallelism [24]. To limit the adverse effects of resource sharing on performance and efficiency of the system, PRISM creates equally sized spatial clusters of cores for *dual-modular redundancy*. Moreover, to cater for performance degradation as a consequence of losing parallelism, PRISM enables an adaptive selective resiliency capability by means of which the core-level resources of clusters are allowed to be reconfigured for exploiting *resiliency–accuracy* tradeoff space. To the best of our knowledge, no prior work has been done that incorporates hardware isolation principle in the context of assuring performance efficient *dual-modular redundancy* (resiliency) in multicore systems.

4.3.2 Improved System Availability. Similarly to the resiliency schemes listed in Table 1, PRISM enables high soft-error protection via isolation-driven *dual-modular redundancy*. It provides system-level crash/hang protection and resiliency against coherence protocol lock-ups. This is primarily attributed to the adoption of strong hardware isolation primitive for every shared hardware resource in PRISM, which ensures that the coherence messages never overlap across the two clusters. Hence, even if any given cluster hangs, the other cluster continues with its execution without being effected. This property allows PRISM to provide high *system availability*. These resiliency guarantees remain stringent in case of adopting the selective resilience capability of PRISM. This is because, even though the *non-resilience* mode does not implement redundancy, it employs a *bound checking* mechanism that ensures output correctness. If a soft-error event causes an upset in the *non-resilience* mode, then the *system availability* never gets impacted as the *∂-Output* (computed by the *resilience* mode) is always available for the system to utilize.

4.3.3 Non-Intrusive Hardware Design. Generally, TLR schemes require additional hardware support to provide soft-error protection guarantees. Among others, a general problem in TLR-based schemes (e.g., References [11, 39]) is that they allow all cores to access the directory and off-chip memory resources without any protective measures, essentially leading to cache coherence protocol hangs. Coherence protocol implementation is notoriously complicated, and even if these cache structures are protected via ECC and CRC techniques, a soft-error strike can still cause

a message to get lost, resulting in the intended destination to continuously wait and causing the directory to lock-up. Additionally, the added hardware complexity in temporal TLR schemes increases when selective resiliency concepts are employed in the architecture. For example, a prior work [26] extends [39] to improve the performance by trading off program output accuracy with resilience overheads. However, these performance enhancements come at the cost of intrusive hardware changes to the core-pipeline and private caches, on top of the cache coherence protocol modifications. The proposed PRISM architecture not only makes resiliency more efficient in terms of performance with hardware isolation and selective resilience capabilities, it also brings out the potency in terms of its effectiveness as it does not require intrusive modifications to cache coherency, and clusters' core-level resources (i.e., core-pipeline, private–shared caches, and TLBs).

5 METHODOLOGY

The PRISM architecture is implemented on a real multicore TileraTile-Gx72 processor [46]. *Tile-Gx72* is a tiled multicore architecture comprising of 72 tiles with each tile featuring a 64-bit multi-issue in-order core, 32 KB private L1-I/D caches, and a 256 KB shared L2 cache slice. PRISM is prototyped using 64 of 72 available cores. The off-chip DRAM memory is accessible using four on-chip 72-bit ECC protected DDR3 controllers that are attached to independent physical memory channels. Moreover, it consists of five independent 2D mesh networks with *X-Y routing*, one for on-chip cache coherence traffic, one for memory controller traffic, and others for core-to-core and I/O traffic. The Tilera Multicore Components API library is used for ensuring isolation, that includes facilities to form clusters of cores, manage network traffic across clusters, regulate on-chip and off-chip data access controls, and manage shared cache data placement. The target iterative decision applications used for evaluation are listed in Section 4.2.2.

5.1 Architectural Modeling

5.1.1 Temporal Dual-Modular Redundancy. The **temporal dual-modular redundancy (T-DMR)** setup (also referred to as, *thread-level redundancy*) is considered as a baseline in this work, which is modeled on *TileraTile-Gx72* using 64 of 72 available cores. These cores, their respective core-level resources (i.e., private–shared caches, on-chip network routers), and on-chip memory controllers are all time-shared across the redundant instances of the application (cf. Figure 1 in Section 3.1). The *data correctness checker* for verification purposes is implemented using the shared memory model, where a 64-bit hash for both instances' outputs are compared with each other.

5.1.2 Spatial Dual-Modular Redundancy. The **spatial dual-modular redundancy (S-DMR)** setup is also modeled, where the redundant application instances spatially execute on the system. In this setup, each application instance's threads are pinned to dedicated cores using `tmc_cpus_set_my_cpu()`, such that each application instance is provided with 32 cores, essentially forming *clusters*. However, the redundant application instances share the large stateful resources, i.e., L2 slices and memory controllers (cf. Figure 2). The S-DMR implements a similar *data correctness checker* as discussed for T-DMR.

5.1.3 The PRISM Architecture. The implementation details of the proposed PRISM architecture on *TileraTile-Gx72* are similar to S-DMR setup (cf. Section 5.1.2). However, strong hardware isolation for all hardware resources is ensured across the redundantly executing application clusters. To isolate shared hardware resources, the default *hash-for-homing* scheme is overridden to use the *local homing* scheme for pinning data structures on specified L2 cache slices. This pinning of data on dedicated L2s is done using `tmc_alloc_set_home(&alloc, core_id)` API call. To ensure complete shared cache isolation, *L2-replication* is also disabled. For off-chip memory isolation, each application cluster is provided with its own set of on-chip memory controllers via

`tmc_alloc_set_nodes_interleaved (&alloc, pos)`. Here, `pos` represents the bit-mask representation of memory controllers to be selected, e.g., `pos = 0b1100` suggests mapping the data on MC_2 and MC_3 controllers. The *data correctness checker* is implemented using the **user-dynamic network (UDN)** of *Tile-Gx72* multicore that implements *core-to-core messaging* that does not interfere with the cache coherence traffic. For verification, each cluster computes a 64-bit hash of its output, and sends it over to the other cluster using UDN. The overheads associated with data correctness (verification) checks are including in the completion time breakdown.

5.2 Selective Resiliency

A **selective resilience scheme (SEL)** is also applied on both T-DMR and PRISM architectures. The SEL scheme is evaluated for various iterative applications (listed in Section 4.2.2), where the starting **X**% of an application iterations are executed in *dual-modular redundancy* (the *resilience* mode). The remaining iterations (i.e., **N-X**%) execute in the *non-resilience* mode, which implements *bound checkers* to ensure correct functionality and acceptable application output convergence (cf. Section 4.2.2). To switch from *resilience* to *non-resilience* mode, the redundant instance in case of T-DMR scheme is terminated for letting the other instance execute without time-sharing hardware resources. Contrarily, in the case of PRISM, one of the two clusters is terminated and the other cluster is provided with all 64 cores to exploit core-level parallelism. Moreover, the shared cache slices are reallocated to the single expanded cluster, where the pages are first un-mapped from their current L2 home cache slices using `tmc_alloc_unmap (*addr, size)` API call, followed by remapping the cluster's data structures and memory pages to all available 64 L2 cache slices via `tmc_alloc_remap (&alloc, size, new_size)` call. The overheads for *bound checking* and cluster resource reallocation (for PRISM only) are added to the completion time breakdown.

5.2.1 Accuracy Analysis for SEL. The SEL scheme executes certain percentage of application iterations in the *non-resilience* mode for achieving better performance by trading off resilience overheads with program output accuracy. To observe the effectiveness of the adopted *bound checkers*, soft-error fault injection is performed. Considering soft-errors as bit-flips, which can happen anywhere during the execution of non-crucial iterations and have unpredictable effects to the variables about to commit, the fault injection analysis mimics these soft-errors using random values. Although the occurrence of soft-errors is rare [12], under extreme operating conditions, this rate can be higher. Therefore, the accuracy tradeoff is explored by evaluating the impact of a single soft-error strike, as well as higher error rates on the program execution. The fault-inject setup injects random values (based on the data type) in any given (randomly selected) non-crucial iteration of the application. Using such a metric of injecting faults based on a range allows us to cater for nearly all the possible bit-flip scenarios [26]. To measure the accuracy of every application, the percentage difference between "the *golden output reference* (program's output with error-free execution) and the output observed when the error was injected" is computed. The outputs returned by each application in a realistic setting are discussed in Section 4.2.2. For applications that provide per-node- (or per-vertex-) based outputs (such as BFS, PR, and CC), a variety of accuracy metrics are considered to observe the overall impact on accuracy. These metrics are considered to model different queries issued by an analyst (programmer or user of the application). However, the accuracy of remaining applications is measured using a single metric, as they return a single output vector. Multiple error injection simulations are performed (~1,000 times in this article) to obtain the average program accuracy for each iterative parallel application. The 1,000 single bit faults introduced refer to per application, meaning that in total 8,000 faults were injected for all 8 applications considered in this work. These accuracy results are collected offline and this application profiling involves measuring the application's accuracy sensitivity to injected errors. For every

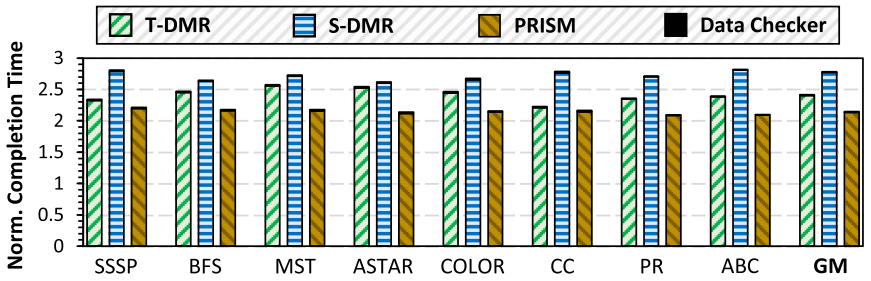

Fig. 5. Normalized completion times for all applications with T-DMR, S-DMR, and PRISM are shown.

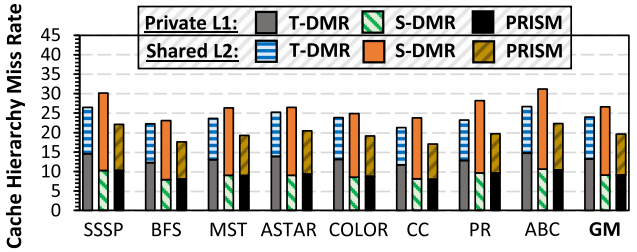

Fig. 6. Cache hierarchy miss rates for each application.

application, it takes ~2–4 ms to inject 1,000 faults and we measure the accuracy by comparing gathered results with the error-free golden reference.

6 EVALUATION

The proposed PRISM resiliency architecture is evaluated against the temporal and spatial redundancy schemes, i.e., T-DMR and S-DMR. Section 6.1 solely focuses on dual-modular redundancy and highlights the performance and system availability advantages over the baseline schemes. Section 6.2 evaluates the selective resiliency capability (referred to as SEL) on PRISM and T-DMR schemes, where the performance and accuracy variations for both schemes are analyzed with varying crucial iteration counts. The section concludes by highlighting numerous sensitivity studies.

6.1 Performance and Availability Analysis of PRISM

Figure 5 shows the completion time comparison of T-DMR against S-DMR and the proposed PRISM architecture. The reported numbers show the completion time (left y-axis) for each application (x-axis). These results are normalized to a baseline scheme with no resiliency support. The T-DMR scheme performs *dual-modular redundancy* using all 64 system cores, shared cache slices, and all available memory controllers. It incurs an overhead of ~2.4× over the no-resiliency baseline. Contrarily, the S-DMR scheme incurs an average overhead ~2.79× over the baseline scheme with no resilience support, i.e., ~18% worse compared to T-DMR (cf. Figure 5). These overheads are primarily due to loss of thread-level parallelism (each cluster operates using 32 cores), as well as the interference on shared hardware resources, i.e., shared L2 slices, on-chip network router, and memory controllers. In the context of interference, the application instances compete for same shared L2 cache resources, which leads to increased stress on the available cache capacity (discussed later in this section using Figure 6). These factors reduce data locality and stress the memory controller queues, causing higher contention delays to access the off-chip memory. Finally, both instances (clusters) of the application generate traffic that interferes in the routers, causing

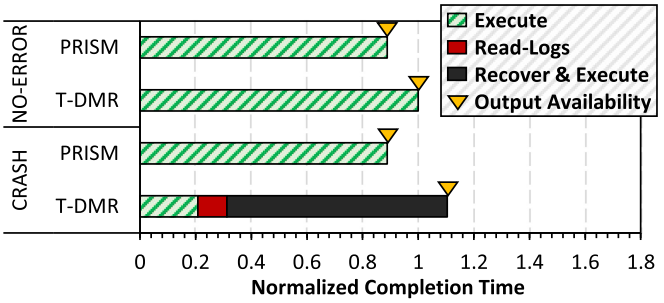

Fig. 7. The output availability of T-DMR and PRISM.

high contention delays in the on-chip networks. On the contrary, T-DMR experiences similar resource sharing effects; however it overcomes these interference effects by exploiting all available core-level parallelism for performance.

The proposed PRISM scheme elegantly isolates the hardware resources across clusters to limit interference, which in turn results in providing performance improvements of ∼12% over the T-DMR scheme (∼30% improvement over S-DMR). These improvements are primarily because of isolating the hardware resource to diminish resource sharing effects. Even though the T-DMR scheme exploits all available core-level parallelism, it allows time-sharing of core-level resources resulting in higher number of cache misses, on-chip network router contention delays, and increased stress on the memory controller queues. On the contrary, PRISM isolates all resources at the hardware level, such that each cluster is provided with its dedicated set of cores-level resources. The reported completion times in Figure 5 also include the overheads associated with *data correctness checker* for verification purposes. These overheads include time taken by both applications to compute, send, receive, and compare the *64-bit XOR hash* of the output. Both shared memory checkers (used by T-DMR and S-DMR) and the UDN *checker* incur insignificant overheads of less than 1%.

Figure 6 shows the cache hierarchy (per-core L1 and L2) miss rates under all evaluated resiliency schemes to understand the impact of resource sharing. As expected, both T-DMR and S-DMR schemes suffer from higher cache hierarchy miss rates, compared to the PRISM architecture. The S-DMR scheme exhibits higher L2 misses as the spatially co-executing application instances compete for the L2 cache slices, and result in a higher number of misses. However, the L1 caches are better utilized, since each L1 is occupied by its allocated application instance. However, the T-DMR scheme primarily suffers from higher L1 cache misses, since it temporally executes both application instances on a given L1 cache, resulting in higher capacity and conflict misses. The benefit of using T-DMR are observed for L2 misses, since the application instances are temporally separated to better utilize this shared resource. The PRISM architecture seamlessly adopts the L1 cache behaviors of S-DMR. Moreover, the strong isolation of the L2 cache resources result in much better utilization as compared to S-DMR. In fact, the L2 cache utilization is observed to nearly match the advantages of the T-DMR approach. On average, PRISM improves overall cache performance by ∼16% and ∼27% compared to T-DMR and S-DMR, respectively. Figure 7 compares T-DMR with PRISM in the context of *system availability*, which is defined as "*the time instance at which the final result/output is returned by the application to the system user.*" This *availability* metric is measured in two different situations (y-axis), i.e., when there exists no soft-error and when the system crashes as a consequence of a soft-error strike. The geometric mean completion time of all applications, normalized to T-DMR with NO-ERROR, is reported. In case of error-free scenario, PRISM improves performance by ∼12% compared to T-DMR, resulting in better *output availability*.

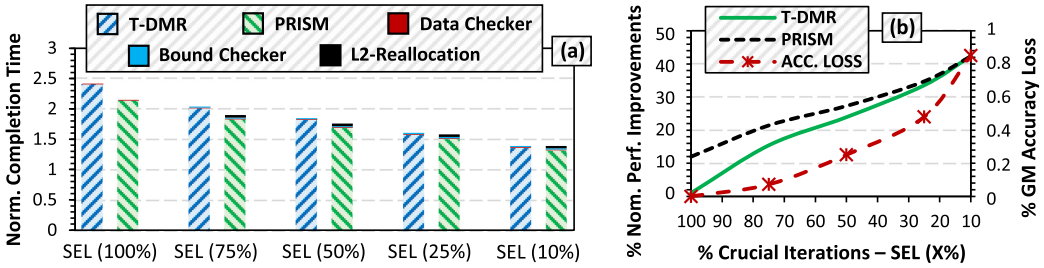

Fig. 8. Normalized completion times of T-DMR and PRISM with selective resiliency are shown in (a). The performance improvements and accuracy loss with decreasing crucial iterations are shown in (b).

To model the CRASH scenario, a single error is injected in any one of the two application instances that raises an exception and causes that application to terminate. A *check-pointing* mechanism is also modeled for the T-DMR scheme that *logs* all the modifications made to the output data structure by an application. This *logging* is done in parallel to the application execution; thus, it does not incur any overheads. Assuming the crash happens at 0.2 s of the reported normalized time, the T-DMR scheme requires a system-level reboot, where it first reads the generated *logs* and starts the execution of the application from the point of failure. Evidently, the overheads of reading these logs further impact the performance and *output availability* of the T-DMR scheme. However, PRISM provides *crash resiliency* alongside impeccable *output availability*, which is in fact similar in both scenarios (i.e., NO-ERROR and CRASH). This is primarily due to the strong hardware isolation primitive that makes sure that clusters' resources are inaccessible to each other. Indeed, isolation of hardware resources assures that the coherence messages and network packets never overlap across clusters. All in all, one of the two clusters always stays online and active, even if the other cluster goes offline due to a soft-error crash. The key insight here is that during the event of one cluster crashing down, the final output from the other (active) cluster is always going to be correct, because the probability of two separate soft-error strikes perturbing the two independent clusters is quite negligible. Thus, the correct output from PRISM becomes available for use at the same time when the unaffected cluster terminates.

6.2 Selective Resiliency Analysis for PRISM

Under selective resilience scheme, each application requires a *bound checking* mechanism during the non-resilient iterations of the application. The *bound checkers* for each of the target applications are discussed in Section 4.2.2. Both T-DMR and PRISM are evaluated for the proposed selective resiliency scheme, where initial **X%** iterations of the application execute in *resilience* (i.e., *dual-modular redundancy*) mode. Upon finishing the *resilience* mode, the system is switched to the *non-resilience* mode to execute the remaining **(N-X)%** iterations. When switched to the *non-resilience* mode, the T-DMR scheme terminates redundant execution of application instances and only lets one instance to execute while utilizing all system resources, whereas PRISM halts one of the two clusters, and provides all 64 cores/threads and shared L2 cache slices to the *active* cluster. However, this active cluster still accesses off-chip memory using its original two memory controllers.

Figure 8(a) shows the geometric mean completion time of all iterative decision algorithms when deployed on the selective resilience scheme (denoted as SEL (X%)) for both T-DMR and PRISM. The factor **X%** in SEL (X%) refers to the number of crucial iterations executed in the *resilience* mode of the SEL scheme. Note, SEL (100%) represent the complete *dual-modular redundant* execution in T-DMR and PRISM (as shown in Figure 5). The numbers reported in Figure 8(a) are normalized to a baseline scheme with no resiliency support. The overheads for switching from *resilience* to

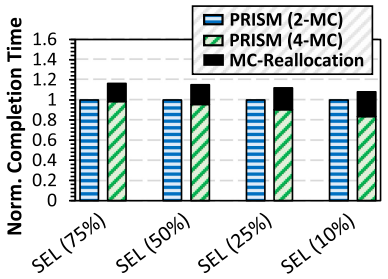

Fig. 9. PRISM's memory controller reallocation overheads.

non-resilience mode are all added to the Compute component of the normalized completion time breakdown. Evidently, performance improves for both schemes with lowering the crucial iteration count, i.e., moving from SEL (100%) to SEL (10%). This is primarily due to the decreasing crucial iteration count, where both T-DMR and PRISM schemes perform reduced number of redundant computations.

Figure 8(b) shows the performance improvements over T-DMR at SEL (100%) (left y-axis) for both T-DMR and PRISM when the crucial iteration count (x-axis) is reduced. The geometric mean output accuracy loss for all applications (right y-axis) when subjected to a single soft-error injection is also reported. In terms of performance, PRISM performs ∼12% better compared to the T-DMR baseline at SEL (100%) due to its resistance against adverse interference effects (cf. Section 6.1). Moving from SEL (100%) to SEL (10%), both schemes continue to improve performance over the T-DMR scheme at SEL (100%) baseline primarily due to reduction in the redundant work. These improvements are observed to increase to ∼43% (reduction to ∼1.31× over no resiliency scheme), whereas performance improvements to ∼42.7% (reduction to ∼1.33× over no resiliency scheme) are observed for T-DMR. However, with the *bound checker* employed, advancing from SEL (100%) to SEL (10%) results in an increase in the accuracy loss of ∼0.07% to ∼0.85%.

Clearly, the difference in performance gained from both schemes is observed to become smaller, as the number of crucial iterations are lowered from SEL (100%) to SEL (10%). Later at SEL (10%) point, this difference becomes approximately zero, exhibiting that T-DMR now performs at par with PRISM. As the crucial iteration count decreases, the interference effects in T-DMR also reduce due to the reduction in the redundant work. Therefore, the margin for PRISM to take advantage by limiting interference also gets narrower. Another aspect for this difference is that PRISM does not reallocate on-chip memory controllers when selective resilience is applied. This limits PRISM to exploit memory-level parallelism. Contrarily, T-DMR continues to fully exploit its core and memory level parallelism.

6.2.1 Sensitivity to Memory Controller Reallocation. Under selective resiliency, PRISM reallocates all core-level resources to the single active cluster to exploit core-level parallelism. However, it does not reallocate all memory controllers, essentially disallowing the active cluster to exploit all available memory bandwidth. The reason for not reallocating on-chip memory controllers is because it requires a new memory initialization setup for the application. Figure 9 shows the normalized completion time for PRISM with (2-MC) and without (4-MC) memory controller reallocation for different SEL schemes. For each SEL entry, the time for PRISM (4-MC) is normalized to its respective PRISM (2-MC) number. Evidently, re-allocating memory controllers allows applications to better exploit memory-level (especially at lower crucial iteration count, e.g., SEL (10%)), which in turn improves performance. However, the reallocation procedure incurs additional performance overhead, overcoming the improvements obtained via memory-level parallelism. Overall,

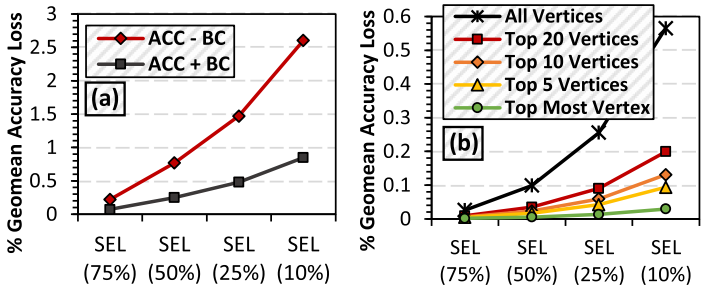

Fig. 10. (a) Single-output accuracy loss with and without *bound checker*, and (b) shows accuracy loss of PR, CC, and BFS.

performance degradation of ~16% to ~8% is observed, when crucial iteration count is reduced from SEL (75%) to SEL (10%). Thus, memory controller reallocation is not adopted for PRISM.

6.2.2 Sensitivity to Bound Checking. Figure 10(a) shows the geometric mean accuracy loss of all applications that provide a single result as their output (i.e., SSSP, ASTAR, MST, COLOR, and ABC), when subjected to a single-error injection. The accuracy comparison is performed with and without employing the designed **bound checkers (BC)**. As expected, the accuracy loss for these applications increases as the number of crucial iterations (executing in the *resilience mode*) employed under (SEL) reduce. However, the magnitude of the accuracy loss is quite high when there exists no bound checker, i.e., the accuracy loss rises from ~0.3% to ~2.7% when shifting from SEL (75%) to SEL (10%). This is due to the fact that when an application does not employ *bound checkers*, the perturbed value(s) get propagated to the final output that impacts the application output. On the contrary, the accuracy loss reduces significantly when the proposed *bound checkers* are employed with the selective resiliency scheme, i.e., the accuracy improves by ~32% and always stays below 1%. This is primarily because BC allows only those values to be committed to the final output that satisfy the implemented bounds. If the bounds do not match for certain values, then they are not allowed to be propagated to the final output and are replaced with intermediate output values computed at the end of *resilience mode*.

6.2.3 Sensitivity to Different Accuracy Metrics. Figure 10(b) shows the geometric mean accuracy loss of iterative applications with per-vertex output values (i.e., PR, CC, and BFS), when subjected to a single soft-error injection. This sensitivity shows impact of using different accuracy measuring metrics on the final accuracy. These metrics model different queries issued by an analyst (programmer/system-user) to compute the applications' output accuracy. For example, the metric "All Vertices" implies that the accuracy loss is measured using output values of all the vertices present in the input graph, whereas the metric "Top Most Node" refers to measuring the accuracy loss using the output value of just the top most vertex in the input graph. The remaining three metrics are modeled in a similar fashion. For better understanding, consider Google's PageRank algorithm as an example. Certainly, the rank values of all visited web pages are not always important. Instead, the most visited web pages are of more interest. Thus, the accuracy loss can also be measured in different aspects, i.e., finding the accuracy loss by comparing the top 10 or 20 vertex values. The insight here is that the accuracy loss not only increases with decreasing crucial iteration count, but also with increasing per-vertex output values considered in defining and measuring the accuracy. For instance, the geometric mean accuracy loss of ~0.26% is reported for SEL (25%) in case of the "All Vertices" metric. However, at the same selective resiliency point, the accuracy loss tends to be significantly lesser, i.e., ~0.02%, when the "Top Most Node" accuracy measuring

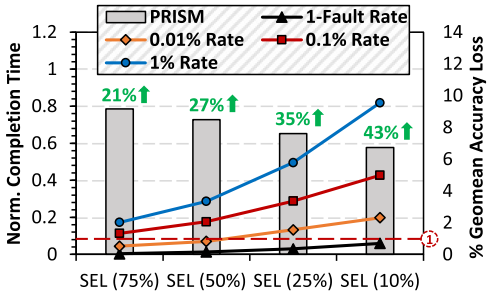

Fig. 11. The *performance–accuracy* tradeoff space at various aggressive soft-error rates (ϵ).

metric is considered. Thus, it is absolutely imperative to consider a reasonable accuracy measuring metric. Note, the geometric mean accuracy loss reported in Figure 8(b) uses the conservative "All Vertices" metric.

6.2.4 Sensitivity to Aggressive Error-Rates. Indeed, the resiliency and performance demands for safety-critical systems vary based on the surrounding environmental conditions. To model these conditions, Figure 11 considers different aggressive soft-error rates (i.e., 0.01%, 0.1%, and 1%) to observe their impact on the accuracy of the applications. This fault injection for an error rate of ϵ% is done in such a way that ϵ% of the total instructions belonging to each non-crucial iteration (executing in the *non-resilience* mode) are exposed to random values. For example, if the error rate is set to 0.1% Rate and an application executes 50 non-crucial iterations where each iteration has 100 instructions, then random values will be introduced in 1 (100 * 0.1%) randomly chosen instruction for each of the 50 non-crucial iterations. The "1-Fault Rate" represents the accuracy loss at a single error injection. Evident from Figure 11, the geometric mean accuracy loss for applications increases with the increase in error rate. This is expected, as with higher error rates, application(s) experience increased number of faults; thus, impacting the application output accuracy. For instance, the accuracy loss increases from ~0.35% to ~5.8% when the error rate is increased from "1-Fault Rate" to 1% Rate. Note, these accuracy loss numbers are acquired in the presence of *bound checkers*. A consistent tradeoff trend is observed for all error rates, that is the application output accuracy always drops, whereas the performance always improves with decreasing crucial iterations (i.e., scanning from SEL (75%) to SEL (10%)).

The reported numbers on top of each bar in Figure 11 represent the overall geometric mean performance improvements over PRISM (or SEL (100%)) assuming an accuracy threshold of 1%. Clearly, for the accuracy threshold of 1%, the SEL (75%) and SEL (50%) system can withstand an error rate of 0.01% Rate, essentially improving the performance by ~21% and ~27%, respectively. Contrarily, with low crucial iterations, SEL (25%) and SEL (10%) can only withstand 1-Fault Rate to satisfy threshold of 1%, resulting in performance boosts of ~35% and ~43%, respectively. To conclude, given these realistic and aggressive soft-error rates and their impact on accuracy, it is imperative to choose such an accuracy loss threshold that satisfies the system's output accuracy demands, whilst guaranteeing high-end performance.

7 CONCLUSION

Multicores have emerged as the norm for general-purpose and domain-specific computing. However, they introduce numerous challenges for protection against soft-errors due to complex communication and memory access protocols. The resiliency schemes proposed in the literature

generally incur overheads due to adverse interference effects caused by aggressive hardware sharing. This work proposes PRISM, an efficient resilience architecture that creates spatially isolated clusters of cores, where redundant execution does not experience interference. With strong hardware isolation, the PRISM architecture enables impeccable system availability at light-weight hardware modifications to a multicore processor. PRISM is prototyped on a real TileraTile-Gx72 multicore processor, where it is shown to improve the performance of redundant execution by 12% over a TLR scheme that exploits multicore parallelism. Moreover, PRISM enables a novel selective resiliency scheme that trades off application output accuracy for lower resilience overheads. This scheme is implemented for iterative decision algorithms, where PRISM allows certain iterations of the algorithm to execute redundantly. However, the remaining iterations execute in the clear without strong resiliency protections, while assuring acceptable output accuracy. PRISM with its selective resilience capability is shown to further improve performance by 43% over the TLR scheme, at the cost of <1% accuracy loss in presence of a singe soft-error.

REFERENCES

[1] M. Ahmad, M. Shan, A. Rehman, and O. Khan. 2020. Accelerating relax-ordered task-parallel workloads using multi-level dependency checking. In *Proceedings of the ACM International Conference on Supercomputing* (2020).

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. 105–117.

[3] Konstantinos Aisopos and Li-Shiuan Peh. 2011. A systematic methodology to develop resilient cache coherence protocols. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, 47–58. DOI: https://doi.org/10.1145/2155620.2155627

[4] I. Akturk and U. R. Karpuzcu. 2020. ACR: Amnesic checkpointing and recovery. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. 30–43.

[5] F. Busato and N. Bombieri. 2016. An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures. *IEEE Trans. Parallel Distrib. Syst.* 27, 8 (Aug. 2016), 2222–2233. DOI: https://doi.org/10.1109/TPDS.2015.2485994

[6] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. 2012. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. 1–11. DOI: https://doi.org/10.1109/SC.2012.36

[7] C. Demetrescu, A. V. Goldberg, and D. S. Johnson (Eds.). 2009. *The Shortest Path Problem.*

[8] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, 385–396. DOI: https://doi.org/10.1145/1736020.1736063

[9] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. 2016. Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'16)*.

[10] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. 1997. Fault injection techniques and tools. *Computer* 30, 4 (Apr. 1997), 75–82. DOI: https://doi.org/10.1109/2.585157

[11] Rajshekar Kalayappan and Smruti R. Sarangi. 2015. FluidCheck: A redundant threading-based approach for reliable execution in manycore processors. *ACM Trans. Archit. Code Optim.* 12, 4, Article 55 (Dec. 2015), 26 pages. DOI: https://doi.org/10.1145/2842620

[12] T. Karnik, B. Bloechel, K. Soumyanath, V. De, and S. Borkar. 2001. Scaling trends of cosmic ray induced soft errors in static latches beyond 0.18 /spl mu/. In *Proceedings of the 2001 Symposium on VLSI Circuits*. 61–62. DOI: https://doi.org/10.1109/VLSIC.2001.934195

[13] D. S. Khudia and S. Mahlke. 2014. Harnessing soft computations for low-budget fault tolerance. In *Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 319–330. DOI: https://doi.org/10.1109/MICRO.2014.33

[14] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. 2013. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *Proceedings of the ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS'13)*. 31–40.

[15] M. Kooli and G. Di Natale. 2014. A survey on simulation-based fault injection tools for complex systems. In *Proceedings of the 2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS'14)*. 1–6. DOI: https://doi.org/10.1109/DTIS.2014.6850649

[16] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, Q. Chen, M. Guo, and V. Janapa Reddi. 2020. Asymmetric resilience: Exploiting task-level idempotency for transient error recovery in accelerator-based systems. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*.

[17] T. Li, R. Ragel, and S. Parameswaran. 2012. Reli: Hardware/software checkpoint and recovery scheme for embedded processors. In *Proceedings of the 2012 Design, Automation Test in Europe Conference Exhibition (DATE'12)*. 875–880. DOI: https://doi.org/10.1109/DATE.2012.6176621

[18] T. Li, M. Shafique, J. A. Ambrose, S. Rehman, J. Henkel, and S. Parameswaran. 2013. RASTER: Runtime adaptive spatial/temporal error resiliency for embedded processors. In *Proceedings of the 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC'13)*. 1–7.

[19] A. Meixner, M. E. Bauer, and D. J. Sorin. 2008. Argus: Low-cost, comprehensive error detection in simple cores. *IEEE Micro* 28, 1 (Jan. 2008), 52–59. DOI: https://doi.org/10.1109/MM.2008.3

[20] 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. DOI: https://doi.org/10.1109/ISCA.2002.1003566

[21] N. Oh, S. Mitra, and E. J. McCluskey. 2002. ED4I: Error detection by diverse data and duplicated instructions. *IEEE Trans. Comput.* 51, 2 (Feb. 2002), 180–199. DOI: https://doi.org/10.1109/12.980007

[22] H. Omar, M. Ahmad, and O. Khan. 2017. GraphTuner: An input dependence aware loop perforation scheme for efficient execution of approximated graph algorithms. In *Proceedings of the2017 IEEE International Conference on Computer Design (ICCD'17)*. 201–208. DOI: https://doi.org/10.1109/ICCD.2017.38

[23] H. Omar, B. D'Agostino, and O. Khan. 2020. OPTIMUS: A security-centric dynamic hardware partitioning scheme for processors that prevent microarchitecture state attacks. *IEEE Trans. Comput.* 69, 11 (Nov. 2020), 1558–1570. DOI: https://doi.org/10.1109/TC.2020.2996021

[24] H. Omar, H. Dogan, B. Kahne and O. Khan. 2018. Multicore resource isolation for deterministic, resilient and secure concurrent execution of safety-critical applications. *IEEE Computer Architecture Letters* 17, 2 (2018), 230–234. DOI: 10.1109/LCA.2018.2874216

[25] H. Omar and O. Khan. 2020. IRONHIDE: A secure multicore that efficiently mitigates microarchitecture state attacks for interactive applications. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. 111–122.

[26] Hamza Omar, Qingchuan Shi, Masab Ahmad, Halit Dogan, and Omer Khan. 2018. Declarative resilience: A holistic soft-error resilient multicore architecture that trades off program accuracy for efficiency. *ACM Trans. Embed. Comput. Syst.* 17, 4, Article 76 (Jul. 2018), 27 pages. DOI: https://doi.org/10.1145/3210559

[27] M. W. Rashid and M. C. Huang. 2008. Supporting highly-decoupled thread-level redundancy for parallel programs. In *Proceedings of the2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 393–404. DOI: https://doi.org/10.1109/HPCA.2008.4658655

[28] V. Reddy and E. Rotenberg. 2008. Coverage of a microarchitecture-level fault check regimen in a superscalar processor. In *Proceedings of the 2008 IEEE International Conference on Dependable Systems and Networks with FTCS and DCC (DSN'08)*. 1–10. DOI: https://doi.org/10.1109/DSN.2008.4630065

[29] S. Rehman, F. Kriebel, Duo Sun, M. Shafique, and J. Henkel. 2014. dTune: Leveraging reliable code generation for adaptive dependability tuning under process variation and aging-induced effects. In *Proceedings of the 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC'14)*. 1–6.

[30] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. 2005. Software-controlled fault tolerance. *ACM Trans. Archit. Code Optim.* 2, 4 (Dec. 2005), 366–396. DOI: https://doi.org/10.1145/1113841.1113843

[31] Felipe Restrepo-Calle, Antonio Martínez-Álvarez, Sergio Cuenca-Asensi, and Antonio Jimeno-Morenilla. 2013. Selective SWIFT-R. *J. Electr. Test.* 29, 6 (01 Dec. 2013), 825–838. DOI: https://doi.org/10.1007/s10836-013-5416-6

[32] V. Roberge, M. Tarbouchi, and G. Labonte. 2013. Comparison of parallel genetic algorithm and particle swarm optimization for real-time UAV path planning. *IEEE Trans. Industr. Inf.* 9, 1 (Feb. 2013), 132–141. DOI: https://doi.org/10.1109/TII.2012.2198665

[33] 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*. 84–91. DOI: https://doi.org/10.1109/FTCS.1999.781037

[34] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. 2010. Flexible architectural support for fine-grain scheduling. *SIGPLAN Not.* 45, 3 (Mar. 2010).

[35] Siva Kumar Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, and Sarita V. Adve. 2009. mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, 122–132. DOI: https://doi.org/10.1145/1669112.1669129

[36] I. Sato and H. Niihara. 2014. Beyond pedestrian detection: Deep neural networks level-up automotive safety. In *Proceedings of the GPU Technology Conference*.

[37] Daeho Seo, Akif Ali, Won-Taek Lim, Nauman Rafique, and Mithuna Thottethodi. 2005. Near-optimal worst-case throughput routing for 2-D mesh networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA'05)*.

[38] Q. Shi, H. Hoffmann, and O. Khan. 2015. A cross-layer multicore architecture to tradeoff program accuracy and resilience overheads. *IEEE Comput. Arch. Lett.* 14, 2 (Jul. 2015), 85–89. DOI: https://doi.org/10.1109/LCA.2014.2365204

[39] Q. Shi and O. Khan. 2013. Toward holistic soft-error-resilient shared-memory multicores. *Computer* 46, 10 (Oct. 2013), 56–64. DOI: https://doi.org/10.1109/MC.2013.262

[40] Qingchuan Shi, Hamza Omar, and Omer Khan. 2017. Exploiting the tradeoff between program accuracy and soft-error resiliency overhead for machine learning workloads. arxiv:1707.02589. Retrieved from http://arxiv.org/abs/1707.02589.

[41] T. J. Siegel, E. Pfeffer, and J. A. Magee. 2004. The IBM eServer Z990 microprocessor. *IBM J. Res. Dev.* 48, 3-4 (May 2004), 295–309. DOI: https://doi.org/10.1147/rd.483.0295

[42] J. Tian. 2005. *Fault Tolerance and Failure Containment*. 267–283. DOI: https://doi.org/10.1002/0471722324

[43] A. Vega, C. C. Lin, K. Swaminathan, A. Buyuktosunoglu, S. Pankanti, and P. Bose. 2015. Resilient, UAV-embedded real-time computing. In *Proceedings of the 2015 33rd IEEE International Conference on Computer Design (ICCD'15)*. 736–739. DOI: https://doi.org/10.1109/ICCD.2015.7357189

[44] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron. 2014. Real-world design and evaluation of compiler-managed GPU redundant multithreading. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA'14)*. 73–84. DOI: https://doi.org/10.1109/ISCA.2014.6853227

[45] N. J. Wang and S. J. Patel. 2005. ReStore: Symptom based soft error detection in microprocessors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*. 30–39.

[46] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. C. Miao, J. F. Brown III, and A. Agarwal. 2007. On-chip interconnection architecture of the tile processor. *IEEE Micro* 27, 5 (Sep. 2007), 15–31. DOI: https://doi.org/10.1109/MM.2007.4378780

[47] Xin Xu and H. Howie Huang. 2015. DualVisor: Redundant hypervisor execution for achieving hardware error resilience in datacenters. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGRID'15)*. 485–494.

[48] Xiangyu Zhang, Ramin Bashizade, Yicheng Wang, Cheng Lyu, Sayan Mukherjee, and Alvin R. Lebeck. 2020. Beyond Application End-Point Results: Quantifying Statistical Robustness of MCMC Accelerators. arxiv:eess.SP/2003.04223. Retrieved from https://arxiv.org/abs/2003.04223.

Received July 2020; revised December 2020; accepted February 2021