REPAS: Reliable Execution for Parallel ApplicationS in Tiled-CMPs [⋆]

Daniel Sánchez, Juan L. Aragón and José M. García

Departamento de Ingeniería y Tecnología de Computadores
Universidad de Murcia, 30071 Murcia (Spain)
{dsanchez, jlaragon, jmgarcia}@ditec.um.es

Abstract. Reliability has become a first-class consideration issue for architects along with performance and energy-efficiency. The increasing scaling technology and subsequent supply voltage reductions, together with temperature fluctuations, augment the susceptibility of architectures to errors. Previous approaches have tried to provide fault tolerance but, however, they usually present critical drawbacks concerning hardware duplication or performance degradation which for the majority of common users result unacceptable.
RMT (Redundant Multi-Threading) is a family of techniques based on SMT processors in which two independent threads (master and slave), fed with the same inputs, redundantly execute the same instructions, in order to detect faults by checking their outputs. In this paper, we study the under-explored architectural support of RMT techniques to reliably execute shared-memory applications. We show how atomic operations induce to serialization points between master and slave threads. This bottleneck has an impact of 34% in execution time for several parallel scientific benchmarks. To address this issue, we present REPAS (Reliable execution of Parallel ApplicationS in tiled-CMPs), a novel RMT mechanism to provide reliable execution in shared-memory applications. While previous proposals achieve the same goal by using a big amount of hardware - usually, twice the number of cores in the system - REPAS architecture only needs a few extra hardware, since the redundant execution is made within 2-way SMT cores in which the majority of hardware is shared. Our evaluation shows that REPAS is able to provide full coverage against soft-errors with a lower performance slowdown in comparison to a non-redundant system than previous proposals at the same time it uses less hardware resources.

1 Introduction

The increase in the number of available transistors in a chip has made it possible to build powerful processors. In this way, CMPs have become a good approach

[⋆] This work has been jointly supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 05831/PI/07, also by the Spanish MEC and European Commission FEDER funds under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03".

to improve performance in an energy-efficient way, while keeping a manageable complexity to exploit the thread-level parallelism. Furthermore, for current and future CMP architectures, more efficient designs are tiled-CMPs [18]. These architectures are organized around a direct network, since area, scalability and power constraints make impractical the use of a bus as the interconnection network.

However, this trend in the increasing number of transistors per chip has a major drawback due to the growth of the failure ratio in every new scale generation [14]. Firstly, the fact of having a higher number of transistors per chip increases the probability of a fault. Secondly, the growth of temperature, the decrease of the supply voltage and the subthreshold voltage in the chip, in addition to other non-desirable effects such as higher power supply noise, signal cross-talking, process variation or in-progress wear-out, compromise the system's reliability.

Hardware errors can be divided into three main categories: transient faults, intermittent faults and permanent faults [8, 3]. Both transient and intermittent faults appear and disappear by themselves. The difference is that, while transient faults are originated by external agents like particle strikes over the chip, intermittent faults are caused by intra-chip factors, such as process variation combined with voltage and temperature fluctuations [20]. In addition, transient faults disappear faster than intermittent faults, on average. Finally, permanent faults remain in the hardware until the damaged part is replaced. Therefore, this paper aims at the study of fault tolerant techniques to detect and recover from transient and intermittent faults, also known as soft errors.

Although the fault ratio does not represent a handicap for the majority of users, several studies show how soft errors can heavily damage industry [8]. For instance, in 1984 Intel had certain problems delivering chips to AT&T as a result of alpha particle contamination in the manufacturing process. In 2000, a reliability problem was reported by Sun Microsystems in its UltraSparc-II servers deriving from insufficient protection in the SRAM. A report from Cypress Semiconductor shows how a car factory is halted once a month because of soft errors [21]. Another fact to take into account is that the fault ratio increases due to altitude. Therefore, reliability has become a major design problem in the aerospace industry.

Fault detection has usually been achieved through the redundant execution of the program instructions, while recovery methods are commonly based on checkpointing. A checkpoint reflects a safe state of the architecture in a temporal point. When a fault is detected, the architecture is rolled-back to the previous checkpoint and the execution is restarted.

There are several proposals to achieve fault tolerance in microarchitectures. RMT (Redundant Multi-Threading) is a family of techniques based on redundant execution in which two threads execute the same instructions. Simultaneous and Redundantly Threaded processors (SRT) [12] and SRT with Recovery (SRTR) [19] are two of them, implemented on SMT processors in which two independent and redundant threads are executed with a delay respect to the other which speeds up their execution. These early approaches are attractive since they do not require many design changes in a traditional SMT processor. In addition, they only add some extra hardware for communication purposes between the threads. However, the major drawback of SRT(R) is the inherent non-scalability of SMT processors as the number of threads increases. The arrival of CMP architectures contributed to solve this problem, with proposals such as Chip-level Redundant Threaded processors (CRTR) [2], Reunion [16] and Dynamic Core Coupling (DCC) [5]. These techniques, in contrast, suffer from high inter-core communication through either dedicated buses or the underlying interconnection network.

Although there are different approaches using SRTR and CRTR with sequential or independent multithreaded applications [19][4], the architectural support for redundant execution with shared-memory workloads is not well suited, since atomic operations induce a serialization point between master and slave threads, as we will show in Section 3.1. On the other hand, more recent proposals such as Reunion or DCC, aimed at parallel workloads, use a shared-bus as interconnection network. Therefore, when they are moved to a more scalable environment such as a direct-network, they suffer from significant performance degradation as the number of cores increases (as studied in [17]), in addition to the extra number of cores that they require.

To address all these issues, in this paper we propose *Reliable Execution of Parallel ApplicationS* (REPAS) in tiled-CMPs. The main features that characterize REPAS are:

– A scalable solution built by adding dual SMT cores to form a tiled-CMP.
– Able to detect and recover from transient and intermittent faults.
– Reduced performance overhead compared to previous proposals (13% less slowdown than CRTR and 7% less than DCC for 16-thread workloads – recall that DCC uses twice the number of cores (i.e., 32 *vs.* 16 cores) for those reported slowdowns.
– Low decoupling between master and slave threads which allows both quicker fault detection and faster recovery times (reduced to just tens of cycles). The former represents an interesting feature for critical applications where fault detection cannot be delayed for long; being the latter an important feature in case of a burst of faults, as it usually occurs for intermittent faults [20].

The rest of the paper is organized as follows. Section 2 provides some background and reviews the related work. Section 3 details the previous CRTR architecture and shows its performance issues in shared-memory environments. In Section 4 we describe REPAS and Section 5 shows the experimental results. Finally, Section 6 summarizes the main conclusions of our work.

2 Related work

When comparing different fault-tolerant mechanisms, we can point out four main characteristics. Firstly, the *sphere of replication or SoR* [12], which determines the components in the microarchitecture that are replicated. Secondly, the *synchronization*, which indicates how often redundant copies compare their computation results. Thirdly, the *input replication* method, which defines how redundant copies observe the same data. Finally, the *output comparison* method, which defines how the correctness of the computation is assured. Table 1 summarizes the main characteristics of the proposals covered in this section.

Table 1. Main characteristics of several fault-tolerant architectures.

	SoR	**Synchronization**	**Input Replication**	**Output Comparison**
SRT(R) CRT(R)	Pipeline, Registers	Staggered execution	Strict (Queue-based)	Instruction by instruction
Reunion	Pipeline, Registers, L1Cache	Loose coupling	Relaxed input replication	Fingerprints
DCC	Pipeline, Registers, L1Cache	Thousands of instructions	Consistency window	Fingerprints, Checkpoints
HDTLR	Pipeline, Registers, L1Cache	Thousands of instructions	Sub-epochs	Fingerprints, Checkpoints

One of the first approaches to full redundant execution is Lockstepping [1], a proposal in which two statically bound execution cores receive the same inputs and execute the same instructions step by step. Later, the family of techniques Simultaneous and Redundantly Threaded processors (SRT) [12], SRT with Recovery (SRTR) [19], Chip-level Redundantly Threaded processors (CRT) [9] and CRT with Recovery (CRTR) [2] was proposed, based on a previous approach called AR-SMT [13]. In SRT(R) redundant threads are executed within the same core. The SoR includes the entire SMT pipeline but the first level of cache. The threads execute in a *staggered execution* mode, using strict input replication and output comparison on every instruction. Other studies have chosen to allocate redundant threads in separate cores. This way, if a permanent fault damages an entire core, a single thread can still be executed. Among these studies it is worth mentioning CRT(R) [9, 2], Reunion [16], DCC [5] and HDTLR [11]. In all these proposals, a fundamental point is how redundant pairs communicate with each other, as we will summarize later.

In Reunion, the *vocal core* is responsible for coherently accessing and modifying shared-memory. However, the *mute core* only accesses memory by means of non-coherent requests called *phantom requests*, providing redundant access to the memory system. This approach is called *relaxed input replication.* In order to detect faults, the current architectural state is interchanged among redundant cores by using a compression method called *fingerprinting* [15] through a dedicated point-to-point fast bus. Relaxed input replication leads to input incoherences which are detected as faults. As a result, checking intervals must be short (hundred of instructions) to avoid excessive penalties. Violations in relaxed input replication induce to a serialized execution (very similar to lock-stepped execution) between redundant cores, affecting performance with a degradation of 22% over a base system when no faults are injected.

Dynamic Core Coupling (DCC) [5] does not use any special communication channel and reduces the overhead of Reunion by providing a decoupled execution of instructions, making larger comparison intervals (thousand of instructions) and reducing the network traffic. At the end of each interval, the state of redundant pairs is interchanged and, if no error is detected, a new checkpoint is taken. As shown in [5], the optimal checkpoint interval for DCC is 10,000 cycles, meaning that the time between a fault happens and its detection is usually very high. Input incoherences are avoided by a consistency window which forbids data updates, while the members of a pair have not observed the same value. However, DCC uses a shared bus as interconnection network, which simplifies the consistency window mechanism. Nevertheless, this kind of buses are not scalable due to area and power constraints. In [17], DCC is studied in a direct-network environment, and it is shown that the performance degradation rises to 19%, 39% and 42% for 8, 16, and 32 core pairs.

Recently, Rashid et al. proposed Highly-Decoupled Thread-Level Redundancy (HDTLR) [11]. HDTLR architecture is similar to DCC, in which the recovery mechanism is based on checkpoints which reflect the architecture changes between *epochs*, and modifications are not made visible to L2 until verification. However, in HDTLR each redundant thread is executed in different hardware contexts (*computing wavefront* and *verification wavefront*), maintaining coherency independently. This way, the consistency window is avoided. However, the asynchronous progress of the two hardware contexts could lead to memory races, which result in different execution outcome, masking this issue as a transient fault. In a worst-case scenario, not even a rollback guarantees forward progress, so an order tracking mechanism is proposed which enforces the same access pattern in redundant threads. This mechanism implies the recurrent creation of sub-epochs by expensive global synchronizations. Again, in this study the interconnection network is a non-scalable shared-bus.

3 CRTR as a building block for reliability

CRTR is a fault tolerance architecture proposed by Gomaa et al. [2], an extension to SRTR [19], for CMP environments. In CRTR, two redundant threads are executed on separate SMT processor cores, providing transient fault detection. Furthermore, since redundant threads are allocated in distant hardware, the architecture is potentially able to tolerate permanent faults as well. These threads are called *master* (or *leading*) and *slave* (or *trailing*) threads, since one of them runs ahead the other by a number of instructions called *slack*. As in a traditional SMT processor, each thread owns a PC register, a renaming map table and a register file, while all the other resources are shared.

The master thread is responsible for accessing memory to load data which bypasses to the slave thread, along with the accessed address. Both data and addresses are kept in a FIFO structure called Load Value Queue (LVQ) [12]. This structure prevents the slave thread from observing different values from those the master did, a phenomenon called *input incoherence*. The philosophy in CRTR is to avoid cache updates until the corresponding value has been verified. In order to do that, when a store instruction commits in the master, the value and accessed address are bypassed to the slave which keeps them in a structure called Store Value Queue (SVQ) [12]. When a store commits in the slave, it accesses the SVQ and if the check successes, the L1 cache is updated. Other structures used in CRTR are the Branch Outcome Queue (BOQ) [12] and the Register Value Queue (RVQ) [19]. The BOQ is used by the master to bypass the destination of a branch to the slave which uses this information as branch predictions. Availability for these hints is assured thanks to the slack, because by the time the slave needs to predict a branch, the master knows the correct destination which bypasses to the slave. This way, the execution speed of the latter is increased because branch mispredictions are avoided. Finally, the RVQ is used to bypass register values of every committed instruction by the master, which are needed for checking. Upon detecting a fault, the recovery mechanism is initiated. The slave's register file is a safe point since no updates are performed on it until a successful verification. Therefore, the slave bypasses the content of its register file to the master; pipelines of both threads are flushed and execution is restarted from the detected faulty instruction.

As it is said before, separating the execution of a master thread and its corresponding slave in different cores adds the ability to tolerate permanent faults. However, it requires a wide datapath between cores to bypass all the information required for checking. Furthermore, although wire delays may be hidden by the slack, the cores bypassing data must be close to each other to avoid stalling.

3.1 CRTR in shared-memory environments

Although CRTR was originally proposed and evaluated for sequential applications [9, 2], the authors argue that it could be used for multithreaded applications, too. However, we have found that, with no additional restrictions, CRTR can lead to wrong program execution for shared-memory workloads in a CMP scenario, even in the absence of transient faults.

In shared-memory applications, such as those that can be found in SPLASH-2, the access to critical sections is granted by primitives which depend on atomic instructions. In CRTR, the master thread never updates memory. Therefore, when a master executes the code to access a critical section, the value of the "lock" variable will not be visible until the slave executes and verifies the same instructions. This behavior makes it possible that two (or more) master threads can access a critical section at the same time, potentially leading to a wrong execution.

In order to preserve sequential consistency and, therefore, the correct program execution, the most straightforward solution is to synchronize master and slave threads whenever an atomic instruction is executed. This conservative approach introduces a noticeable performance degradation (an average 34% slowdown as evaluated in Section 5.2). The duration of the stall of the master thread depends on two factors: (1) the size of the slack, which determines how far the slave thread is, and (2) the number of write operations in the SVQ, which must be written in L1 prior to the atomic operation to preserve consistency.

4 REPAS architecture

At this point, we propose *Reliable Execution for Parallel ApplicationS* (REPAS) in tiled-CMPs. We create the reliable architecture of REPAS by adding CRTR cores to form a tiled-CMP. However, we avoid the idea of separating master and slave threads in different cores. We justify this decision for two main reasons: (1) as a first approach, our architecture will not tolerate permanent faults whose appearance ratio is still much lower than the ratio of transient faults [10], and (2) we avoid the use of the expensive inter-core datapaths. An overview of the core architecture can be seen in Figure 1. As in a traditional SMT processor, issue queues, register file, functional units and L1-cache are shared among the threads. The shaded boxes in Figure 1 represent the extra hardware introduced by CRTR and REPAS as explained in Section 3.

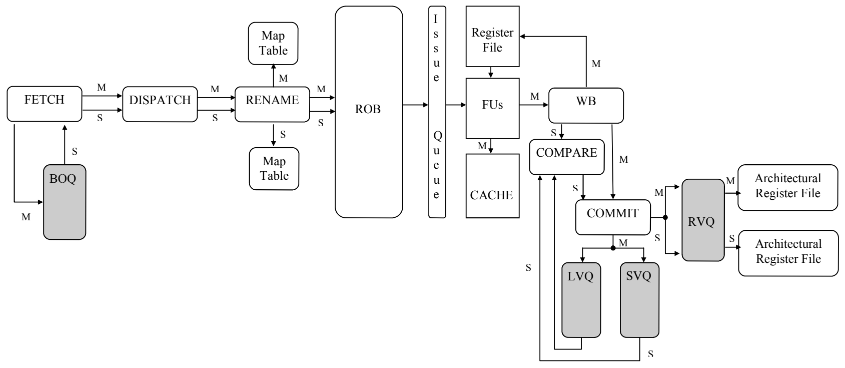


Fig. 1. REPAS core architecture overview.

In benchmarks with high contention resulting from synchronization, the constraint described in Section 3.1 for CRTR, may increase the performance degradation of the architecture dramatically. To avoid frequent master stalls derived from consistency, we propose an alternative management of stores. Instead of updating memory just after verification, a better approach is to allow updates in L1 cache without checking. By doing so, the master thread will not be stalled as a result of synchronizations.

Collaterally, we clearly reduce the pressure on the SVQ. In the original CRTR implementation, a master's load must look into the SVQ to obtain the value produced by an earlier store. This implies an associative search along the structure for every load instruction. In REPAS, we eliminate these searches since the up-to-date values for every block are stored in L1 cache where they can be accessed as usual.

Unfortunately, this measure complicates the recovery mechanism. When a fault is detected, the L1 cache may have unverified blocks. Upon detection, all unverified blocks must be invalidated. Furthermore, when a store is correctly checked by the slave, the word in L1 must be written-back into L2. This way, the L2 cache remains consistent, even if the block in L1 is invalidated as a result of a fault. This mechanism is carried out in background, and despite the increased L1-to-L2 traffic, it does not have a noticeable impact on performance.

4.1 Implementation details of REPAS

To avoid error propagation deriving from a wrong result stored in L1 cache by the master, unverified blocks in cache must be identified. In order to do this, we introduce an additional bit per L1 cache block called *Unverified bit* which is activated on any master write. When the Unverified bit is set on a cache block, it cannot be displaced or shared with other nodes, effectively avoiding the propagation of a faulty block. Eventually, the Unverified bit will be cleared when the corresponding slave thread verifies the correct execution of the memory update.

Clearing the Unverified bit is not a trivial task. A master thread can update a cache block several times before a verification takes place. If the first check performed by the slave is successful, it means that the first memory update was valid. However, it does not imply that the whole block is completely verified, since the rest of the updates has not been checked yet. One simple way of knowing if a block needs more checks before clearing the unverified bit is by looking if the block appears more than once in the SVQ. If it does, more verifications need to be performed. Yet, this measure implies an associative search in the SVQ. Nonetheless, as we said before, we eliminate much of the pressure produced by master's loads. In quantitative terms, in the original CRTR proposal there was an associative search every master's load, and now we have an associative search every slave's store. This results in a significant reduction of associative searches within the SVQ given that the load/store ratio for the studied benchmarks is almost 3 to 1. Furthermore, as this operation is performed in parallel to the access to L1 cache we do not expect an increase of the cycle time to access L1-cache.

5 Evaluation

5.1 Simulation environment

We have implemented REPAS as well as the proposed extensions to CRTR to support the execution of parallel applications, evaluating the performance results by using the functional simulator Virtutech Simics [6], extended with the execution-driven multiprocessor simulator GEMS [7].

Our study has been focused on a 16-way tiled-CMP where each core is a dual-threaded SMT, which has its own private L1 cache, a portion of the shared L2 cache and a connection to the on-chip network. The architecture follows the sequential consistency model with the write-read reordering optimization. The main parameters of the architecture are shown in Table 2(a). For the evaluation, we have used a selection of scientific applications: Barnes, FFT, Ocean, Radix, Raytrace, Water-NSQ and Water-SP are from the SPLASH-2 benchmark suite.

Table 2. Characteristics of the evaluated architecture and used benchmarks.

(a) System characteristics

16-Way Tiled CMP System		**Cache Parameters**	
Processor Speed	2 GHz	Cache line size	64 bytes
Max. Fetch / retire rate	4 instructions / cycle	**L1 cache**	
Consistency model	Sequential Consistency	Size, associativity	64KB, 4 ways
Memory parameters		Hit time	1 cycle
Coherence protocol	MOESI	**Shared L2 cache**	
Write Buffer	64 entries	Size, associativity	512KB/tile, 4 ways
Memory access time	300 cycles	Hit time	15 cycles
Network parameters		**Fault tolerance parameters**	
Topology	2D mesh	LVQ/SVQ	64 entries each
Link latency (one hop)	4 cycles	RVQ	80 entries
Flit size	4 bytes	BOQ	64 entries
Link bandwidth	1 flit/cycle	Slack Fetch	256 instructions

(b) Input sizes

Benchmark	**Size**	**Benchmark**	**Size**
Barnes	8192 bodies, 4 time steps	Tomcatv	256 points, 5 iterations
FFT	256K complex doubles	Unstructured	Mesh.2K, 5 time steps
Ocean	258 x 258 ocean	Water-NSQ	512 molecules, 4 time steps
Radix	1M keys, 1024 radix	Water-SP	512 molecules, 4 time steps
Raytrace	10Mb, teapot.env scene		

Tomcatv is a parallel version of a SPEC benchmark and Unstructured is a computational fluid dynamics application. The experimental results reported in this work correspond to the parallel phase of each program only. Sizes problems are shown in Table 2(b).

5.2 Performance analysis

We have simulated the benchmarks listed in Table 2(b) in a tiled-CMP with 16 cores. Figure 2 compares CRTR with REPAS. The results are normalized with regards to a system in which no redundancy is introduced. Overall, these results clearly show that REPAS performs better than CRTR. This tendency is more noticeable in benchmarks such as Unstructured or Raytrace which present many more atomic synchronizations than the rest of the studied benchmarks, as it can be observed in Table 3.

Table 3. Frequency of atomic synchronizations (per 100 cycles).

	Barnes	**FFT**	**Ocean**	**Radix**	**Raytrace**	**Tomcatv**	**Unstructured**	**Water-NSQ**	**Water-SP**
Synchronizations	0.162	0.039	0.142	0.013	0.2	0.02	3.99	0.146	0.014
Cycles per synchronization	478.7	405.1	376.7	451.2	561.9	405.6	566	563.7	408.8

CRTR obtains an average performance degradation of 34% in comparison to a non-redundant system. On the contrary, REPAS is able to reduce this degradation down to 21%. Atomic operations damage CRTR in two ways. Firstly, they act as serialization points: the slave thread must catch up with the master. Secondly, all the stores in the SVQ must be issued to memory before the actual atomic operation, in order to preserve the sequential consistency model.

We have also evaluated DCC in a 32-core CMP with a 2D-mesh network, using the same parameters shown in Table 2(a). As studied in [17], DCC per-

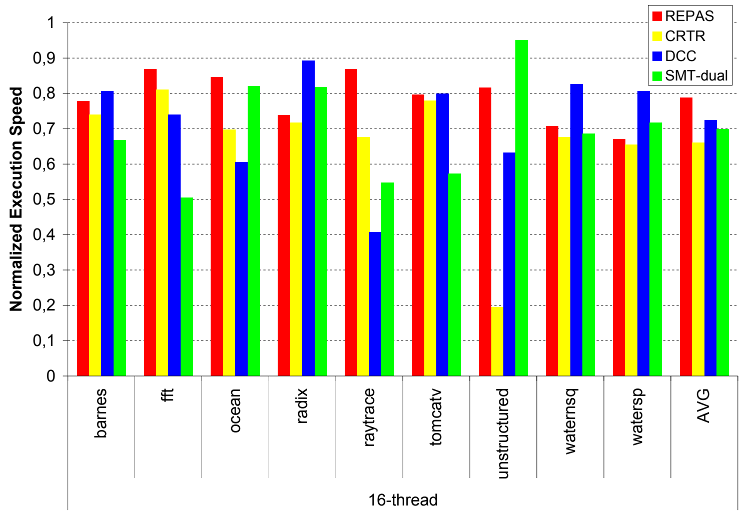


Fig. 2. Execution time overhead over a non fault-tolerant architecture.

forms poorly in this environment due to the latency imposed by the *age table*, introduced to maintain the master-slave consistency. As we can see in Figure 2, REPAS is 7% faster than DCC on average. However, it is important to note that DCC uses twice as much hardware as REPAS, since the redundant threads are executed in different cores. This represents another advantage of REPAS over DCC.

Finally, we show the performance of SMT-dual, a coarse-grained redundancy approach which represents a 16-core 2-way SMT architecture executing two copies (A and A') of each studied application. In each core, it is executed one thread of A and one thread of A'. As mentioned in [12], this helps to illustrate what the performance degradation of a SMT processor is when two copies of the same thread are running within the same core. As we can see, REPAS is 9% faster than SMT-dual on average which, at the same time, is faster than CRTR in 2%.

5.3 Speculative sharing

REPAS does not allow the sharing of unverified blocks as a conservative measure to avoid the propagation of errors among cores. On the contrary, DCC [5] is based on a speculative sharing policy. Given that blocks are only verified at checkpointing creation intervals (i.e., 10,000 cycles), avoiding speculative sharing in DCC would degrade performance in an unacceptable way.

For comparison purposes, we have studied the effect of sharing unverified blocks in REPAS. In order to avoid an unrecoverable situation, speculatively delivered data block the commit stage of the requestor. In this way, we introduce new data in the pipeline to operate with (similarly to conventional speculative execution to support branch prediction). Furthermore, a speculative block can be shared with two or more requestors. When the producer validates the block, it sends a signal to all the sharers confirming that the acquired block was correct and the commit stage is re-opened in their pipelines. If a core which produced shared speculatively data detects a fault, an invalidation message is sent to all the sharers in order to flush their pipelines, undoing the previous work.

We have not considered to migrate unverified data speculatively since an expensive mechanism would be necessary to keep track of the changes in the ownership, the sharing chains as well as in the original value of the data block for recovery purposes.

Table 4. Number of speculative sharings and time needed to verify those blocks.

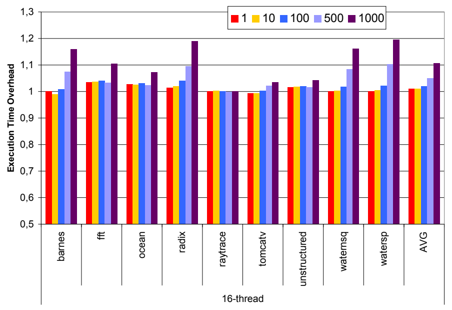
	Barnes	FFT	Ocean	Radix	Raytrace	Tomcatv	Unstructured	Water-NSQ	Water-SP	AVG
# Speculative sharings	12077	50	9217	901	39155	163	224286	244	252	-
Time to verification	102	91.72	96.71	81.8	80.79	107.337	113.14	102.377	76.76	95

Table 4 reflects that speculations are highly uncommon, and that all the time in which we could benefit from, 95 cycles on average, cannot be fully amortized because pipeline is closed at commit. This explains why speculative sharings do not obtain much benefit in REPAS. The performance evaluation shows a neligible performance improvement on average, although for those benchmarks as Barnes, Raytrace and Unstructured which show a greater number of sharings, the performance is increased around 1% over the basic mechanism. Overall, the performance increase due to speculative sharing seems inadequate, since it is not worth the incremented complexity in the recovery mechanism of the architecture.

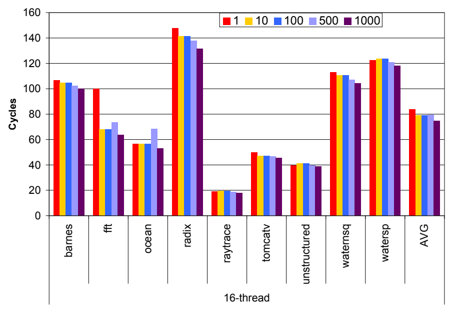
5.4 Transient fault injection

We have shown that REPAS is able to reduce the performance degradation of previous proposals in a fault-free scenario. However, faults and the recovery mechanism to solve them introduce an additional degradation which must be also studied. Figure 3 shows the execution overhead of REPAS under different fault rates. Failure rates are expressed in terms of faulty instructions per million of cycles per core. These rates are much higher than expected. However, they are being evaluated to overstress the architecture and to show the scalability of the system.

As we can see in Figure 3(a), REPAS is able to tolerate rates of 100 faulty instructions per million per core with an average performance degradation of 2% over a test with no injected faults. The average overhead grows to 10% when the fault ratio is increased to 1000 per million. The time spent in every recovery depends on the executed benchmark and it is 80 cycles on average, as we can see in Figure 3(b). This time includes the invalidation of all the unverified blocks and the rollback of the architecture up to the point where the fault was detected. Although we have not completely evaluated it, other proposals as DCC spend thousands of cycles to achieve the same goal (10,000 cycles in a worst-case scenario). This clearly shows the greater scalability of REPAS in a faulty environment.



(a) Overhead under different fault rates



(b) Average time needed for recovery

Fig. 3. Execution time overhead and rollback time under different failure rates.

6 Conclusions

Processors are becoming more susceptible to transient faults due to several factors such as technology scaling, voltage reduction, temperature fluctuations, process variation or signal cross-talking. Although there are many approaches exploring reliability for single-threaded applications, shared-memory environments have not been thoroughly studied.

In this paper, we first study the under-explored architectural support for CRTR to reliably execute shared-memory applications. We show how atomic operations induce a serialization point between master and slave threads. A bottleneck which has an average impact of 34% in the execution time over several parallel scientific benchmarks.

To address this issue, we propose REPAS: Reliable Execution for Parallel ApplicationS in Tiled-CMPs, where we allow updates in L1 cache before verification. Thus, we obtain a more decoupled execution, reducing the stall time due to synchronization. To avoid fault propagation among cores, unverified data reside in L1 cache, in which sharing is not allowed as a conservative measure. With this mechanism, we can reduce the overall performance degradation to 21% with regards to a non-redundant system, advantaging other recent approaches such as Dynamic Core Coupling (DCC) which has an overall impact of 28% using twice the number or cores. Finally, we have also shown that our proposal is able to tolerate fault rates up to 100 faulty instructions per million of executed instructions per core, with an overall performance overhead of hardly 2% over a system with no injected faults.

References

[1] J. Bartlett and J. Gray et al. Fault tolerance in tandem computer systems. In *The Evolution of Fault-Tolerant Systems*. 1987.

[2] M. Gomaa and C. Scarbrough et al. Transient-fault recovery for chip multiprocessors. In *Proc. of the 30th annual Int' Symp. on Computer architecture (ISCA'03)*, San Diego, California, USA, 2003.

[3] A. González and S. Mahlke et al. Reliability: Fallacy or reality? *IEEE Micro*, 27(6), 2007.

[4] S. Kumar and A. Aggarwal. Speculative instruction validation for performance-reliability trade-off. In *Proc. of the 2008 IEEE 14th Int' Symp. on High Performance Computer Architecture (HPCA'08)*, Salt Lake City, USA, 2008.

[5] C. LaFrieda and E. Ipek et al. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proc. of the 37th Annual IEEE/IFIP Int' Conf. on Dependable Systems and Networks (DSN'07)*, Edinburgh, UK, 2007.

[6] P. Magnusson and M. Christensson et al. Simics: A full system simulation platform. *Computer*, 35(2), 2002.

[7] M. K. Martin and D. J. Sorin et al. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4), 2005.

[8] S. Mukherjee. *Architecture design for soft errors.* Morgan Kauffman, 2008.

[9] S. Mukherjee and M. Kontz et al. Detailed design and evaluation of redundant multithreading alternatives. In *Proc. of the 29th annual Int' Symp. on Computer architecture (ISCA'02)*, Anchorage, AK, USA, 2002.

[10] M. Pizza and L. Strigini et al. Optimal discrimination between transient and permanent faults. In *In Third IEEE International High-Assurance Systems Engineering Symposium*, pages 214–223, 1998.

[11] M. Rashid and M. Huang. Supporting highly-decoupled thread-level redundancy for parallel programs. In *Proc. of the 14th Int' Symp. on High Performance Computer Architecture (HPCA'08)*, Salt Lake City, USA, 2008.

[12] S. K. Reinhardt and S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proc. of the 27th annual Int' Symp. on Computer architecture (ISCA'00)*, Vancouver, BC, Canada, 2000.

[13] E. Rotenberg. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In *Proc. of the 29th Annual Int' Symp. on Fault-Tolerant Computing (FTCS'99)*, Madison, WI, USA, 1999.

[14] P. Shivakumar and M. Kistler et al. Modeling the effect of technology trends on soft error rate of combinational logic. In *Proc. of the Int' Conf. on Dependable Systems and Networks (DSN'02)*, Bethesda, MD, USA, 2002.

[15] J. C. Smolens and B. T. Gold et al. Fingerprinting: Bounding soft-error-detection latency and bandwidth. *IEEE Micro*, 24(6), 2004.

[16] J. C. Smolens and B. T. Gold et al. Reunion: Complexity-effective multicore redundancy. In *Proc. of the 39th Annual IEEE/ACM Int' Symp. on Microarchitecture (MICRO 39)*, Orlando, FL, USA, 2006.

[17] D. Sánchez and J. L. Aragón et al. Evaluating dynamic core coupling in a scalable tiled-cmp architecture. In *Proc. of the 7th Int. Workshop on Duplicating, Deconstructing, and Debunking (WDDD'08). In conjunction with ISCA*, 2008.

[18] M. B. Taylor and J. Kim et al. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[19] T. Vijaykumar and I. Pomeranz et al. Transient fault recovery using simultaneous multithreading. In *Proc. of the 29th Annual Int' Symp. on Computer Architecture (ISCA'02)*, Anchorage, AK, 2002.

[20] P. M. Wells and K. Chakraborty et al. Adapting to intermittent faults in multicore systems. In *Proc. of the 13th Int' Conf. on Architectural support for programming languages and operating systems (ASPLOS'08)*, Seattle, WA, USA, 2008.

[21] J. F. Zielger and H. Puchner. *SER-History, Trends and Challenges.* Cypress Semiconductor Corporation, 2004.