J Supercomput (2012) 61:997–1023
DOI 10.1007/s11227-011-0670-9

A fault-tolerant architecture for parallel applications in tiled-CMPs

Daniel Sánchez · Juan L. Aragón · José M. García

Published online: 30 August 2011

Abstract Nowadays, hardware reliability is considered a first-class issue 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.

With the development of CMPs, the interest for using parallel applications has increased. Previous proposals for providing fault detection and recovery have been mainly based on redundant execution over different cores. RMT (Redundant Multi-Threading) is a family of techniques based on SMT (Simultaneous Multi-Threading) 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 in tiled-CMPs.

Initially, we show how atomic operations induce serialization points between master and slave threads, degrading the execution time by 35% for several parallel scientific and multimedia benchmarks. To address this issue, we introduce REPAS (Reliable Execution of Parallel ApplicationS in tiled-CMPs), a novel RMT mechanism to provide reliable execution in shared-memory applications in environments prone to transient faults. REPAS architecture only needs few extra hardware since the redundant execution is performed within 2-way SMT cores in which the majority of hardware is shared. Experimental results show that REPAS is able to provide fault tolerance against soft errors with a lower execution time overhead (around 25% includ-

D. Sánchez (✉) · J.L. Aragón · J.M. García
Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, 30100 Murcia, Spain
e-mail: dsanchez@ditec.um.es

J.L. Aragón
e-mail: jlaragon@ditec.um.es

J.M. García
e-mail: jmgarcia@ditec.um.es

ing the cost of redundancy) in comparison to a non-redundant system than previous proposals while using less hardware resources. Additionally, we show that REPAS supports huge fault ratios with negligible impact on performance (less than 2% for a fault ratio of 100 faults per million cycles).

Keywords Fault tolerance · Soft errors · SMT architectures · Parallel Systems

1 Introduction

The advance in the scale of integration allows to increase the number of transistors in a chip, which are used to build powerful processors such as CMPs (Chip Multiprocessors) [4, 20, 23, 30]. At the same time, manufacturers have started to notice that this trend along with voltage reduction and temperature fluctuation are challenging CMOS technology because of several reliability issues. Among others, we can cite the increasing appearance of hardware errors and other related topics such as process-related cell instability, process variation or in-progress wear-out. Another fact to take into account is that the fault ratio increases due to altitude [17, 35]. Therefore, reliability has become a major design problem in the aerospace industry.

Hardware errors are classified as transient, intermittent or permanent [7, 18]. On the one hand, permanent faults, which are usually caused by electromigration, remain in the hardware until the damaged component is replaced. On the other hand, voltage variation and thermal emergencies are the main cause of intermittent faults. Transient faults, also known as soft errors, appear and disappear by themselves. They can be induced by a variety of sources such as transistor variability, thermal cycling, erratic fluctuations of voltage and radiation external to the chip [18]. Radiation-induced events include alpha-particles from packaging materials and neutrons from atmosphere. It is well established that the charge of an alpha particle or a neutron striking a logical device can overwhelm the circuit inducing its malfunction.

It is hard to find documented cases concerning soft errors in commercial systems. This is because of both the difficulty which involves detecting a soft error and the convenient silence of manufacturers about their reliability problems. However, several studies show how soft errors can heavily damage industry. For instance, in 1984 Intel had certain problems delivering chips to AT&T as a result of alpha particle contamination in the manufacturing process [18]. In 2000, a reliability problem was reported by Sun Microsystems in its UltraSparc-II servers deriving from insufficient protection in the SRAM [18]. A report from Cypress Semiconductor showed how a car factory was halted once a month because of soft errors [36].

Nowadays, several measures have been introduced in microarchitectural designs in order to detect and recover from transient errors such as error detection and correction codes. They are created by specific rules of construction to avoid information loss in the transmission of data. ECC (Error Correction Codes) codes are commonly used in dynamic RAM. However, these mechanisms cannot be extensively used across all the hardware structures. Instead, at the architecture level, DMR (Dual Modular Redundancy) or TMR (Triple Modular Redundancy) have been proposed. In these approaches, fault detection is provided by means of dual and triple execution redundancy.

In this fashion, we find RMT (Redundant Multi-Threading), a family of techniques in which two threads redundantly execute the program instructions. Simultaneous and Redundantly Threaded processors (SRT) [22] and SRT with Recovery (SRTR) [31] are two of them, implemented on SMT (Simultaneous Multi Threading) 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.

In order to provide more scalability, several approaches were designed on top of CMP architectures. Among them, it is worth mentioning proposals such as Reunion [29], Dynamic Core Coupling (DCC) [11] or High Decoupled Thread Level Redundancy (HDTLR) [21]. However, solutions using this kind of redundancy achieve a severe degradation in terms of power, performance and specially in area since they use twice the number of cores to support DMR. Therefore, these approaches are not well suited for general markets as industry claims that a fault-tolerant mechanism should not impose more than 10% of area overhead in order to be effectively deployed [27]. Hence, solutions based on redundant multithreading using SMT cores seem a good approach to achieve fault tolerance without sacrificing too much hardware [13].

Although there are different proposals based on SRTR with either sequential or independent multithreaded applications [9, 31], the architectural support for redundant execution with shared-memory workloads is not well suited. As we will show in Sect. 4.1, in shared-memory parallel applications, the use of atomic operations may induce serialization points between master and slave threads affecting performance depending on the memory consistency model provided by the hardware.

To address all these issues, in this paper we propose REPAS *Reliable Execution of Parallel ApplicationS* in tiled-CMPs. The main contributions of this paper are: (a) identification of a performance problem of traditional RMT implementations; (b) design of a scalable RMT solution built on top of dual SMT cores to form a tiled-CMP; (c) implementation of our proposal in a full-system simulator to measure their effectiveness and execution time overhead. We show that REPAS is able to reduce the execution time overhead down to 25% with respect to a non fault-tolerant architecture while significantly outperforming a traditional RMT mechanism by 13%. Previous proposals such as DCC results in a better performance for specific environments such as Multimedia and Web Server applications. However, REPAS achieves the same goal by using half the hardware used in DCC. Additionally, our mechanism is able to recover from transient faults with negligible performance impact even with extremely high and unrealistic fault rates.

In [26], we presented a preliminary version of REPAS. This article extends our previous work by thoroughly introducing our ideas in order to improve the reader's understanding. The evaluation section has been extended with the study of several new applications from the ALPBench benchmark suite in addition to web server applications such as Apache and SpecJBB. Additionally, we have included a sensitivity analysis as well as a stress study for the L1 cache size.

The rest of the paper is organized as follows. Section 2 reviews some related work. In Sect. 3 we introduce DCC, a fault-tolerant mechanism, for comparison purposes.

Section 4 introduces CRTR and presents its major drawbacks in a parallel shared-memory environment. We present REPAS's architecture in Sect. 5. Section 6 analyzes the performance of REPAS in fault-free and faulty environments. Finally, Sect. 7 summarizes the main conclusions of this work.

2 Related work

There is a large body of literature on detection of soft errors which can be classified in Error Coding, Redundancy and Symptom-based techniques, as we can see in Fig. 1.

Error detection and correction codes are based on the use of extra bits appended to some data in a way that if a fault corrupts the information, this event can be detected or even corrected. In this category, we can include detection techniques (such as parity, checksum, CRC) and recovery techniques (such as ECC), which are implemented in a large variety of memory devices from CDs and DVDs to dynamic RAM. However, these techniques cannot be easily deployed in functional units [18].

Another approach to fault detection follows a scheme based on symptoms [13] which is inspired in ReStore [32]. This study presents a characterization of how errors affect either application or OS behavior with almost no hardware overhead. The detection mechanism is based on the observation of abnormal events such as fatal hardware traps, application exits or hangs in either the program or the OS. Upon a fault is detected, the execution is rolled-back to a previous safe state. However, these approaches cannot provide a solution for those errors which do not modify the behavior of applications such as those affecting values but not control flow.

Finally, we can find redundancy-based techniques which are, so far, the most studied and those in which we focus on. Unlike error correction codes in which each structure is individually protected, these mechanisms are able to cover multiple hardware structures. Therefore, they are usually used to provide fault-tolerant architectures.

When comparing different redundancy mechanisms, we can point out four main characteristics. Firstly, the *sphere of replication (SoR)* [22], which determines the components in the microarchitecture that are replicated. Secondly, the *synchronization*, which indicates how often redundant copies compare their computation results.

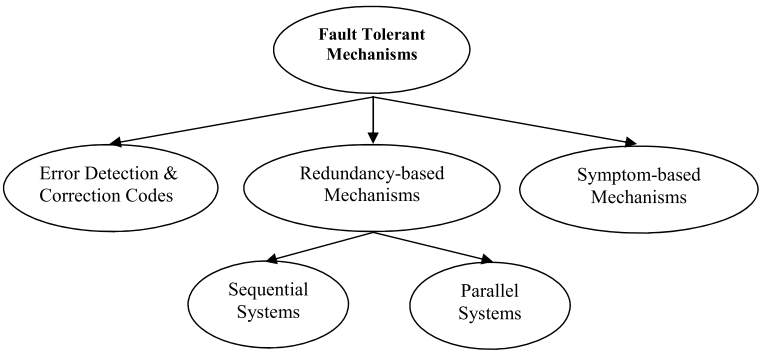

Fig. 1 Fault Tolerant mechanisms

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 described in this section.

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 of Simultaneous and Redundantly Threaded processors (SRT) [22], SRT with Recovery (SRTR) [31], Chip-level Redundantly Threaded processors (CRT) [19] and CRT with Recovery (CRTR) [6] were proposed, all of them based on a previous approach called AR-SMT [24]. 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) [6, 19], Reunion [29], DCC [11] and HDTLR [21]. In all these proposals, a fundamental point is how redundant pairs communicate with each other, as we summarize next.

In Reunion, the *vocal core* is responsible for accessing and modifying shared memory coherently. 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* [28] 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 a serialized

Table 1 Main characteristics of several redundant 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

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) [11] 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 [11], the optimal checkpoint interval for DCC is 10,000 cycles, meaning that the time between a fault happening 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 do not have 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 [25], DCC is studied using a direct network, and in this environment it is shown that the performance degradation rises 19%, 39% and 42% for 8, 16, and 32 core pairs.

Recently, Rashid et al. proposed Highly-Decoupled Thread-Level Redundancy (HDTLR) [21]. 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 outcomes. These events are masked by the architecture as a transient fault. In a worst-case scenario, not even a rollback would guarantee forward progress. Thus, an order tracking mechanism, which enforces the same access pattern in redundant threads, is proposed. This mechanism implies the recurrent creation of sub-epochs by expensive global synchronizations. Finally, as well as in DCC, the interconnection network used in this study is a non-scalable shared-bus, which makes the communication process easier than in a direct network.

3 Dynamic Core Coupling in a direct-network environment

Dynamic Core Coupling (DCC) [11] is a fault-tolerant mechanism for both sequential and parallel applications. DCC implements dual modular redundancy (DMR) by binding pairs of cores in a CMP connected by a shared-bus. To provide fault tolerance, cores in a pair re-execute the program instructions to verify each other's execution. In this section we deeply analyze the major benefits and drawbacks of DCC. In particular, we focus on the impact over the coherence and consistency systems that DCC has when it is ported from a shared-bus to a more scalable direct network.

3.1 DCC in a shared-bus

In DCC, a pair is formed by two cores: the master core and the slave core. To verify the correct execution, at the end of a checkpoint interval each master–slave pair interchange the compressed state of their register file and all the updates performed to

memory. In order to amortize the compression time and save bandwidth these checkpoints intervals are in the order of 10,000 cycles.

Both master and slave cores are allowed to read memory. However, only the master is allowed to modify and share memory values. Writes to memory values are marked in L1 cache by means of an *unverified bit* [16]. This bit indicates that the modification of the block has not been verified yet. In order to avoid the propagation of errors, unverified blocks are not allowed. At the end of every checkpoint interval all the unverified bits are cleared.

To provide a correct execution in a parallel environment, DCC needs several changes in both the coherence and consistency system. As said before, the master core is responsible to share unverified data. From the point of view of coherence, this means that the slave core is not allowed to respond to forwarded requests (request from other cores), although invalidations should be taken accordingly by evicting blocks from cache (without updating lower levels of the memory hierarchy). A complete list of changes to the coherence protocol can be found in the original paper [11].

However, the major difficulty is to provide the master–slave consistency. This means to ensure that both cores obtain the same view of the memory at all times. The pair consistency is violated if between the time a redundant read is performed, an intervening write modifies the value, preventing the second read to obtain the same value than the first one. This problem is solved in DCC by a set of constraints referred to the *master–slave consistency window*. Logically, a window represents a time interval in which any remote intervention could cause a violation of the consistency. For example, a consistency read window is open on any master read and is closed once the slave core commits the same read. To avoid consistency violations, it must be ensured that no write windows are opened for an address in which another window has been previously open.

DCC implements this mechanism by means of an age table. The age table keeps for every load and store, the number of committed loads and stores since the last checkpoint. In Fig. 2(a) we can see how this mechanism works. A node requests an upgrade or a read-exclusive for a block through the shared-bus (the request is seen by all nodes) (1). Each core checks its LSQ (Load Store Queue) in case a speculative load has been issued. If this is the case, the request is answered with a NACK (Negative Acknowledgement) (2). In parallel, each core accesses its age table and reports it to its pair (2). In the following cycle, every master core checks its own age with the slave one. In the case there is a mismatch, it means that a window is open and, therefore, the request is not accepted (NACKed) to avoid a master–slave inconsistency (3). If no mismatch is found the request can be satisfied.

3.2 DCC in a direct-network environment

As the number of cores in a system grows, we observe undesirable effects affecting the scalability of several elements. One of these elements is the interconnection network. As shown in [10], the area required by a shared-bus or a crossbar as the number of cores grows increases to the point of becoming impractical. Hence, we evaluated [25] the performance impact of moving DCC toward a point-to-point unordered network, a more scalable alternative for CMP designs.

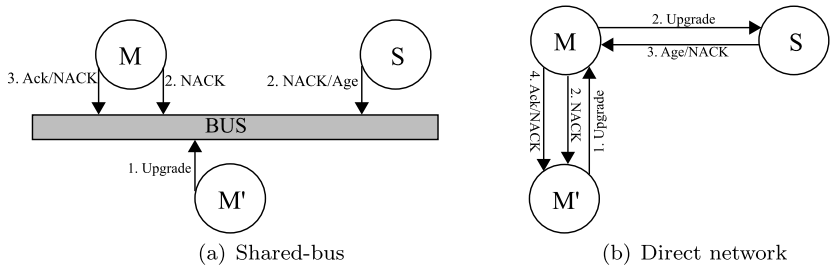

Fig. 2 DCC master–slave consistency

In order to accommodate the behavior of DCC to a direct network, we should introduce several changes in both the coherence and consistency systems. In both cases the problem is the same: without additional support, slave cores are unaware of coherence actions because of the loss of the shared-bus and its "broadcast" capabilities. We solve this issue by redirecting coherence messages (upgrade, read-exclusive and invalidation request) which arrive to master cores to their slave pairs, introducing, unfortunately, a delay in the communication.

We can see how this problem affects the way in which the consistency window works in Fig. 2(b). Upgrade, read-exclusive or invalidations are sent to master cores which are the visible cores in the system (1). These requests can be directly NACKed in case a speculative load is performed in the master (2). In parallel, the request needs to be sent to the slave core which, so far, was unaware of the coherence action. The slave core can deny the request through the master core in case a speculative load is found in its LSQ (3). In the other case, it sends its age to its master pair. Finally, the master core checks for a window violation and then informs the requestor (4). As we can see, we introduce an additional hop in the communication for every coherence action. The impact of these measures over the performance in studied and analyzed in Sect. 6.3.

4 CRTR as a building block for reliability

As opposed to DCC in which the redundancy is taken by using master–slave pairs in different cores (increasing the hardware overhead), another alternative consists of the use of SMT cores. This way we can reduce both the hardware overheads and the delay because of the communications between redundant pairs through the network. Among different alternatives using SMT cores we focus on Chip-level Redundantly Threaded multiprocessor (CRTR).

CRTR is a fault-tolerant architecture proposed by Gomaa et al. [6], an extension to SRTR [31] for CMP environments. In CRTR, two redundant threads are executed on separate SMT processor cores, providing transient fault detection. 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 determined by the *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.

In CRTR the master thread is responsible for accessing memory to load data. After a master load commits, it bypasses it to the slave thread along with the accessed address through a FIFO structure called Load Value Queue (LVQ) [22]. This structure is accessed by the slave thread, preventing to observe different values from those the master did, a phenomenon called *input incoherence*. To avoid associative searches in the LVQ, the slave thread executes loads in program order so it only has to look up the head of the queue. Fortunately, this handicap does not impact on the slave's performance in comparison to the master's because the possible slowdown is compensated with a speedup due to two factors:

- The memory latency of a slave load is very low since data are provided by the LVQ (slave loads behave as cache hits).
- Branch mispredictions are avoided thanks to the Branch Outcome Queue (BOQ) [22]. Therefore, the slave thread executes less instructions than the master.

The master uses the BOQ to bypass the outcome of a committed branch. Then, the slave accesses the BOQ at a branch execution obtaining accurate predictions (perfect outcomes, in fact). Availability for these hints is ensured thanks to the slack since, by the time the slave needs to predict a branch, the master has already logged the correct destination of the branch in the BOQ.

To avoid data corruptions, CRTR never updates cache before values are verified. To accomplish this, when a store instruction is committed by the master, the value and accessed address are bypassed to the slave through a structure called Store Value Queue (SVQ) [22]. When a store commits in the slave, it verifies the SVQ and, if the check succeeds, the L1 cache is updated. Finally, other structure used in CRTR is the Register Value Queue (RVQ) [31]. The RVQ is used to bypass register values of every committed instruction by the master, which are needed for checking.

Whenever a fault is detected, the recovery mechanism is triggered. The slave 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 was 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 in order to bypass all the information required for checking. Furthermore, although wire delays may be hidden by the slack, the cores exchanging data must be close to each other to avoid stalling.

4.1 Memory consistency in LVQ-based architectures

Although CRTR was originally evaluated with sequential applications [6, 19], the authors argue that it could be used for multithreaded applications, too. In LVQ-based systems such as CRTR in which loads are performed by the master thread and stores are performed by the slave thread there is a significant reordering in the memory instructions from the external perspective. In a sequential environment, it does not represent any problem. However, for shared-memory workloads in a CMP scenario, if no additional measures are taken, CRTR can lead to a severe performance degradation due to consistency model constraints.

Our evaluated architecture is a SPARC V9 [8] implementing the Total Store Order (TSO) consistency model. In this consistency model, stores are buffered on a store miss but loads are allowed to bypass these buffered stores. As a measure to improve the performance, stores to the same cache block are coalesced in the store buffer. Finally, atomic instructions and memory fences stall retirement until the store buffer is drained.

In shared-memory applications such as those that can be found in either scientific SPLASH-2 [34], web server, or multimedia workloads, the access to critical sections is granted by acquisition primitives relying on atomic instructions and memory fences. We have noticed that, in this environment, CRTR could lead to a performance loss because of the constraints of the consistency model to ensure mutual exclusion.

The key point is that, in CRTR the master thread never updates memory. Therefore, when a master executes the code to access a critical section, the acquisition is not made visible until the slave executes and verifies the correctness of the instructions involved. This means that, for the rest of master threads, the 'lock' will remain free for a while, enabling two (or more) of these threads to access a critical section as illustrated in Fig. 3. To address this issue, which appears in CRTR without modifying the memory consistency model, we propose, implement and evaluate two different alternatives: *atomic synchronization* and *atomic speculation*.

4.1.1 CRTR with atomic synchronization

In order to preserve the underlying consistency model (TSO) and, therefore, the correct program execution, the most straightforward solution is to synchronize the master and slave threads whenever atomic instructions or memory fences are executed. This way, only when the slave thread catches up with the master and the SVQ drains, the instruction is issued to memory. Therefore, the master thread is not allowed to enter into a critical section without making the results of the acquisition mechanism visible.

Note that this is a conservative approach which introduces a noticeable performance degradation because the master stalls the retirement on every atomic/memory fence instruction. The duration of this stall 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.

4.1.2 CRTR with atomic speculation

One could argue that the previous alternative is not fair to competition. To this end, we have evaluated a mechanism, which we called *Atomic Speculation* to relax even more the consistency constraints imposed by TSO through the use of speculation. Memory ordering speculation has been previously studied in [2, 5, 33] in order to increase the performance of different consistency models.

What we try to accomplish with Atomic Speculation is to avoid the costly synchronizations that atomic instructions and memory fences impose over CRTR. For this, we allow loads and stores to bypass these instructions speculatively. In the same

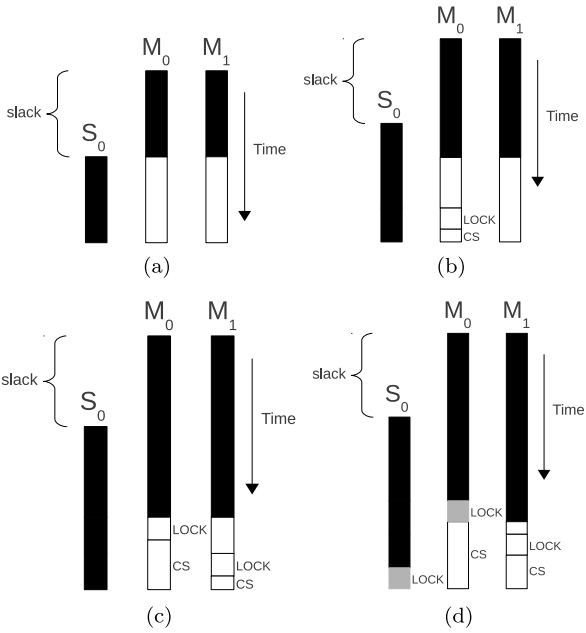

Fig. 3 Violation of the atomicity and isolation of a critical section without proper support. In the figure, two master threads M_0 and M_1, and one slave thread S_0 are presented (the corresponding slave for M_1 has been omitted for simplicity). Part (**a**) shows a snapshot of the program execution. M_0 runs ahead of S_0 by an amount of instructions determined by the slack. A striped portion of a bar means that updates to memory have not been performed yet. Part (**b**) shows the situation when M_0 acquires a lock and enters into the critical section it protects. None of the modifications are visible yet. Part (**c**) shows that M_1 also acquires the lock. This is because M_0 has not updated memory so the lock seems free for the rest of the nodes in the system. M_1 enters the critical section at the same time that M_0. Finally, Part (**d**) shows that when S_0 validates the execution of M_0 and updates memory values, it is too late since atomicity and isolation of the critical section has been violated

fashion as in [5], the list of speculated blocks is maintained in a hardware structure in the core.[1] A hit in the table upon a coherence message from other core indicates that the current speculation could potentially lead to a consistency violation. In this situation, a conflict manager decides whether to roll back the receiver or the requestor because of the miss-speculation. Eventually, if no violations have been detected, the slave thread will catch up with the master. Then, the speculation table is flushed and the speculative mode is finished.

In benchmarks with low to medium synchronization time this kind of speculative mechanism results a good approach. However in other scenarios with highly contended locks the frequency of rollbacks impacts severely on performance. Nonetheless, this mechanism comes at an additional cost such as the hardware needed to roll back the architecture upon a consistency violation. Additionally, this solution requires a change in the way atomicity is implemented since these accesses cannot

[1] The same goal could be accomplished by means of signatures as in certain hardware approaches of Hardware Transactional Memory.

perform the memory update to avoid fault propagation. Finally, there exists a power consumption overhead due to the need of checking the speculation table for every coherence request in speculative mode. Note, however, that we have not considered these overheads in the evaluation section.

5 REPAS architecture

At this point, we present *Reliable Execution for Parallel ApplicationS in tiled-CMPs* (REPAS) [26]. 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 but instead using 2-way SMT cores. This way, the architecture does not rely in the use of the expensive inter-core datapaths while it still offers fault tolerance to soft errors. An overview of the core architecture is depicted in Fig. 4. As in a traditional SMT processor, issue queues, register file, functional units and L1-cache are shared among the master and slave threads. The shaded boxes in Fig. 4 represent the extra hardware introduced by CRTR and REPAS as explained in Sect. 4.

5.1 Sphere of replication in REPAS

In benchmarks with high contention resulting from synchronization, the approaches described in Sect. 4.1 for CRTR may increase the performance degradation of the architecture due to atomic synchronizations or too frequent rollbacks because of miss-speculations. To avoid frequent master stalls derived from consistency, we propose an alternative management of stores in REPAS. Instead of updating memory only after verification, a more suitable approach is to allow updates in L1 cache without checking. This measure implies that unverified data could go outside the SoR while the master thread will not be stalled as a result of synchronizations.

Additionally, with this new behavior we effectively reduce the pressure on the SVQ queue. 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

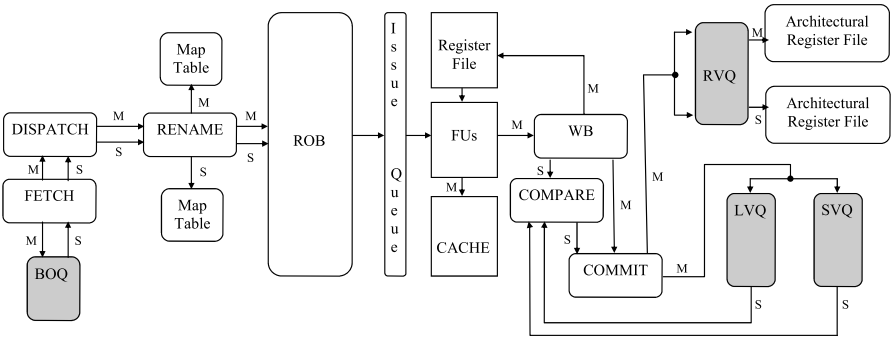

Fig. 4 REPAS core architecture overview

searches since the up-to-date values for every block are stored in L1 cache where they can be accessed as usual.

However, this change in the SoR with respect to CRTR entails an increase in the complexity of the recovery mechanism and the management of verified data. In our approach, in contrast to CRTR, when a fault is detected, the L1 cache may have unverified blocks. The recovery mechanism involves the invalidation of all the unverified blocks in L1. In order to maintain L2 updated with the most up-to-date versions of blocks, when stores are correctly checked by the slave, the values in the SVQ 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 the mechanism triggered because of a fault. To perform these writebacks we use a small coalescing buffer to mitigate the increase of the SVQ-to-L2 traffic in the same fashion as [21]. Despite the increasing SVQ-to-L2 traffic, there is no noticeable impact on performance.

5.2 The unverified bit

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. This way of buffering unverified data was previously introduced by DCC [11]. 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. This mechanism is controlled at the coherence protocol level by adding a new state (M_Unv) to the base MOESI[2] protocol as we can see in Fig. 5. Modified blocks remain in M_Unv state until a positive verification is performed by the slave. Upon this verification, the state of the block transitions from M_Unv to M state, where it can be shared or replaced as usual.

However, clearing the Unverified bit is not a trivial task. We might find a problem when a master thread updates a cache block several times before a verification takes

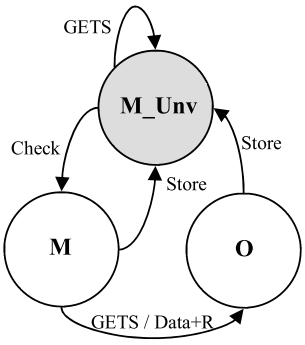

Fig. 5 Transition diagram with the states involved with Unverified blocks. **M_Unv** state: modified by the master and waiting for slave validation (Check). While in this state, the data sharing (GETS) is not allowed

[2]In a MOESI protocol, blocks are within one of the following states: Modified, Ownership, Exclusive, Shared and Invalid.

place. If the first check performed by the slave is successful, it means that the first memory update was valid. However, this does not imply that the whole block is completely verified since the rest of the updates have not been checked yet. We propose two different mechanisms in order to address this issue.

The first mechanism is based on counters per L1-cache block. Each time that the master thread updates a block it increments the counter which is eventually decremented when a verification is performed. When the counter rises 0, a transition from M_Unv to M is performed meaning that the block has been successfully verified. However, these counters have a deep impact in hardware overhead. With small 4-bit counters (which only can record up to 15 consecutive updates) the area overhead becomes around 6% with a 64 KB L1 cache and 64-byte blocks.

Thus, we have adopted a more lightweight mechanism based on the observation of the SVQ: we know if a block needs more slave checks before clearing the unverified bit by checking 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 for every slave's store. This results in a significant reduction of associative searches within the SVQ, given the fact 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 the L1 cache, we do not expect an increase in the L1-cache access latency.

5.3 Fetch and ROB occupancy policies

The most common fetch policy for SMT processors is *round-robin* in which each thread fetches instructions in alternative cycles. In REPAS, the fetch policy needs to interact with the slack mechanism, which significantly differs from the requirements in a typical SMT processor. As in CRTR [6], we have adopted a slightly different policy. When the distance between the two threads is below the threshold imposed by the slack, only the master thread is allowed to fetch new instructions. Contrarily, when the distance is above the threshold, the fetch priority is given to the slave. However, in order to use all the available bandwidth, if the slack is not satisfied but for some reason the master thread cannot fetch more instructions, we allow the slave thread to fetch. In the remaining stages of the pipeline such as decode, issue, execution and commit, the used policy is FIFO.

We can experience a noticeable performance degradation if the master thread fetches enough instructions to completely fill the shared ROB. This happens since the master thread runs some instructions ahead of the slave. In this scenario, the master thread cannot fetch more instructions because of the previously described fetch policy, neither the slave because the ROB (Re-Order Buffer) is full. So, until the ROB entries are released, the two threads are stalled and cannot fetch new instructions.

In order to solve this problem, our approach consists of keeping a percentage of free entries in the shared ROB for the slave. This way, we avoid both threads to stall due to ROB contention. Our experimental results show that 20% of total ROB's free entries is the best case in order to reduce this penalty.

An alternative approach would be to use a private ROB for each thread (or a static partitioning). However, the requirements of the master and slave threads are changing constantly due to the slack mechanism, branch mispredictions and long latency memory operations. In this scenario, a static partitioning is not able to maximize the use of all the available ROB entries. Therefore, a fully shared ROB is the best approach to the architecture presented in REPAS.

5.4 Reliability in the forwarding logic

In our design, the integrity of the information within structures as caches or additional buffers is protected by means of ECC codes. We assume SECDED (Single Error Correction, Double Error Detection) with an additional hardware cost of 12.5% (1 ECC byte per each 8 data bytes). However, a traditional issue derived from the use of queues to bypass data is the potential problems arising from errors in the forwarding logic. An error in the LSQ forwarding logic in the master executing a load instruction, might cause an incorrect bypass to the corresponding slave's load. If this happens, the slave thread would consume a wrong values from the LVQ leading to a SDC (Silent Data Corruption).

To address this potential problem, in REPAS we use a double check: the slave thread compares the load values obtained by means of its own LSQ with the corresponding values in the LVQ. This way, if either the forwarding logic of the master or the slave fail, this check will detect a mismatch in the values signaling a fault. This mechanism result is appropriate to ensure the correction of the data forwarding in the LSQ. Nevertheless, there are some environments in which the coverage could not be considered good enough. In those cases, another mechanism at microarchitecture level as proposed in [3] could be applied, achieving almost a 100% AVF (Architectural Vulnerability Factor) reduction while affecting performance in just 0.3%.

6 Evaluation

6.1 Simulation environment

The methodology used in the evaluation of this paper is based on full-system simulation. We have implemented all the previously described proposals by extending the multiprocessor simulator GEMS [15] from the University of Wisconsin. GEMS is an execution-driven simulator based on Virtutech Simics [14] which we have used to run several parallel applications.

Our study has been focused on a 16-core in which 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 Total Store Order (TSO). The coherence protocol is directory-based MOESI. The main parameters of the architecture are shown in Table 2(a). Among them, it is worth mentioning the 2D-mesh topology used as well as the 256-instruction slack fetch as a result of the sensitivity analysis performed in Sect. 6.2.

For the evaluation, we have used a selection of scientific applications: Barnes, Cholesky, FFT, Ocean, Radix, Raytrace, Water-NSQ and Water-SP are from the

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
Execution mode	Out-of-order		**L1 cache**
Max. Fetch / retire rate	4 instructions / cycle	Size	64 KB
ROB	128 entries	Associativity	4 ways
FUs	6 IALU, 2 IMul	Hit time	1 cycle
	4 FPAdd, 2 FPMul		
Consistency model	Total Store Order (TSO)		
Memory parameters		Shared L2 cache	
Coherence protocol	Directory-based MOESI	Size	512 KB/tile
Write buffer	64 entries	Associativity	4 ways
Memory access time	300 cycles	Hit time	15 cycles
Network parameters		Fault-tolerance parameters	
Topology	2D mesh	LVQ	64 entries
Link latency (one hop)	4 cycles	SVQ	64 entries
Flit size	4 bytes	RVQ	80 entries
Link bandwidth	1 flit/cycle	BOQ	64 entries
		Slack Fetch	256 instructions

(b) SPLASH-2 + Scientific Benchmarks

Benchmark	Size	Benchmark	Size
Barnes	8192 bodies, 4 time steps	Raytrace	10 Mb, teapot.env scene
Cholesky	tk16.0	Tomcatv	256 points, 5 iterations
FFT	256 K complex doubles	Unstructured	Mesh.2K, 5 time steps
Ocean	258 × 258 ocean	Water-NSQ	512 molecules, 4 time steps
Radix	1M keys, 1024 radix	Water-SP	512 molecules, 4 time steps

(c) ALPBench + Web Servers

Benchmark	Size	Benchmark	Size
FaceRec	ALPBench training input	Speechrec	ALPBench training input
MPGDec	525_tens_040.mv2	Apache	100,000 HTTP transactions
MPGEnc	Output from MPGDec	SpecJBB	8,000 transactions

SPLASH-2 [34] benchmark suite. Tomcatv is a parallel version of a SPEC benchmark and Unstructured is a computational fluid dynamics application. Additionally, we have run several multimedia applications: Facerec, MPGDec, MPGEnc, Speechrec from ALPBench benchmark suite [12]. Finally, we have studied two well known web server applications such as Apache and SpecJBB. For the studies, each application has been executed with 16 software threads, each one bound to a different processor core. This means 16 hardware threads for the base case and 32 hardware threads for the rest (16 master threads plus 16 slave threads). An alternative base case would consist of executing 32 software threads for every application (as we have a 16-core

CMP, being each core a 2-way SMT). However, because of the low scalability of some applications, specially the scientific ones, evaluated results are not homogeneous when you compare a 16-threaded application (fault-tolerant machine having 16 masters) with a 32-threaded application. Therefore, in order to isolate the non-scalability issues and perform a fairer comparison, we have chosen as base case the one which uses the same number of threads (16) as the evaluated fault-tolerant architectures which is, in fact, the common approach followed in other previous proposals [6, 9, 31].

The sizes and parameters for the studied applications are reflected in Table 2(b) and Table 2(c), respectively. We have performed all the simulations with different random seeds for each benchmark to account for the variability of multithreaded execution. This variability is represented by the error bars in the figures, enclosing the confidence interval of the results.

For comparison purposes we have implemented several previous proposals. As explained in Sect. 3 DCC incurs in an additional performance degradation when it is ported from a shared-bus to a direct network. The use of shared-buses will be no longer possible in future CMP architectures due to area, scalability and power constraints issues. Therefore, we compare our proposed REPAS against DCC when a direct network such as a 2D-mesh is used. Additionally, we compare REPAS against the performance of SMT-dual and DUAL. SMT-dual models a coarse-grained redundancy approach which represents a 16-core 2-way SMT architecture executing two copies (A and A′) of each studied application. Within each core, one thread of A and one thread of A′ are executed. As mentioned in [22], this helps to illustrate that the performance degradation occurred within a SMT processor when two copies of the same thread are running within the same core. DUAL represents a 16-core non-SMT architecture executing two copies of the same program. In the case of 16-threaded applications it means that each processor executes 2 threads (1 thread of every application). In DUAL, the OS is the responsible of the schedule of the different software threads among the different cores.

6.2 Slack size analysis

The slack fetch mechanism maintains a constant delay between master and slave threads. This delay results in a performance improvement (due to thread-pairs cooperation) because of factors such as the reduction of the stall time for the L1 cache misses in the slave and the better accuracy in the execution of slave's branches thanks to the BOQ. From this perspective, we would choose to use a slack as big as possible.

However, a larger size of the slack also requires an increase in the size of structures like the SVQ or the LVQ to avoid stalls. Furthermore, in a shared-memory environment, a large slack causes that the average life latency of a store (the time spent between the execution of the store and its validation) is increased.

This negatively affects performance because unverified blocks cannot be shared or replaced from cache. Figure 6 shows a sensitivity analysis for different sizes of the slack. The slack is measured in number of fetched instructions between the master and the slave thread. The bars are normalized with respect to the 32 slack size. As is shown, the increase of the slacks size to 256 instructions obtains a noticeable performance improvement. However, further increasing the slack size, is counterproductive.

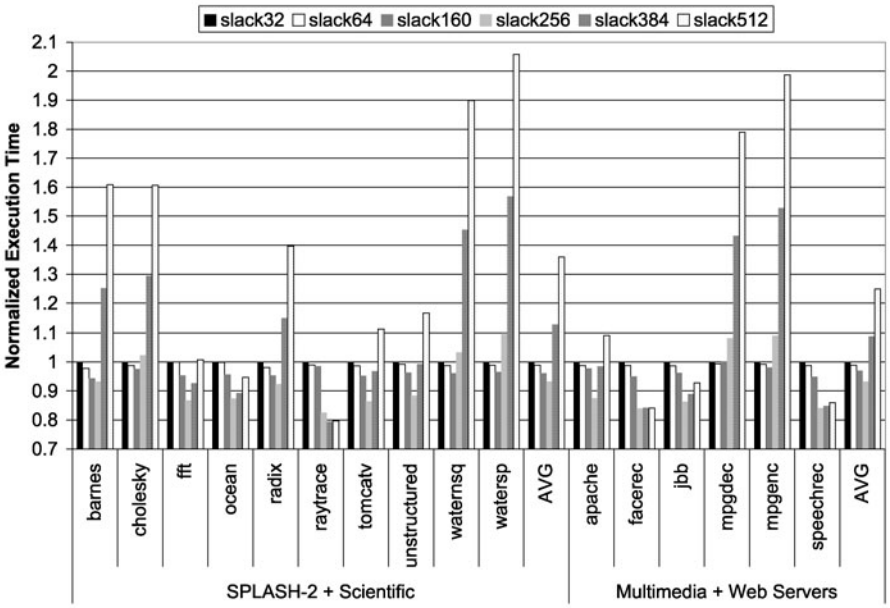

Fig. 6 Sensitivity analysis for the optimal size of the slack

On average, a slack of 256 instructions is 7% better than a slack of 32. Therefore, for subsequent experiments we will use 256 as our target slack.

6.3 Overhead of the fault-free case

We compare our proposed REPAS architecture against CRTR with the alternative mechanisms, atomic synchronization and atomic speculation, as explained in Sect. 4.1. As many other previous proposals [11, 19, 29], we initially present the results of our mechanism in a fault-free environment in order to quantify the execution time overhead for the common case.

Figure 7 plots the results of REPAS normalized with respect to a 16-core system in which there is not any fault-tolerant mechanism. CRTR_sync refers to the atomic synchronization mechanism for CRTR and CRTR_spec refers to the atomic speculation mechanism. As derived from Fig. 7, REPAS outperforms CRTR_sync for both groups of benchmarks (scientific and multimedia/web) by 13% and 6%, respectively, while the execution time overhead rises a 25%, on average, for all the studied benchmarks.

The main source of degradation in CRTR_sync comes from the frequent synchronizations between master and slave threads because of the execution of atomic instructions and memory fences. This effect can be better observed in those benchmarks with more synchronizations such as Ocean, Raytrace and Unstructured, in which the performance exhibited by CRTR_sync is even worse.

As was expected, CRTR_spec outperforms CRTR_sync because of the effectiveness of the speculative mechanism. However, in benchmarks with highly contended

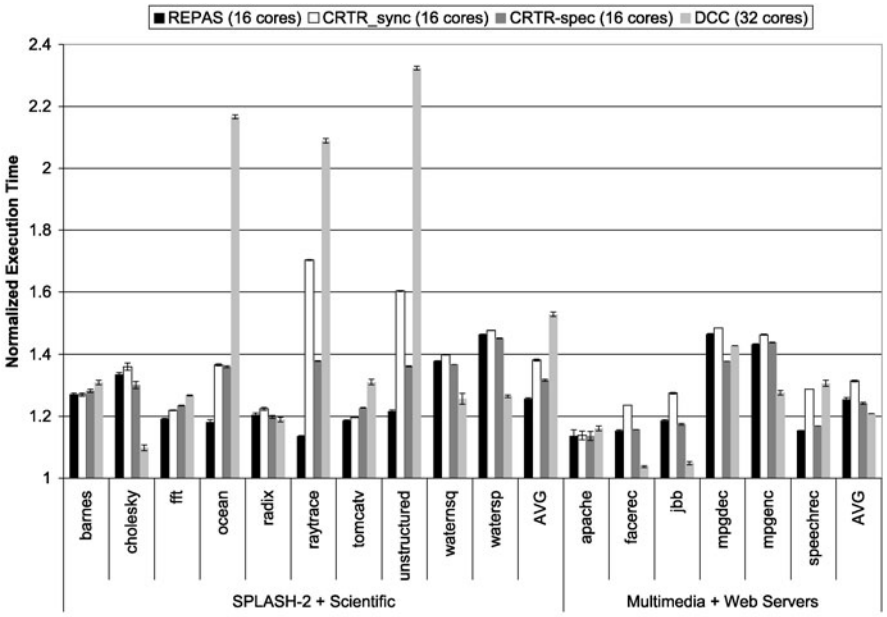

Fig. 7 Execution time overhead over a non fault-tolerant 16-core architecture

locks such as Ocean, Raytrace and Unstructured the number of rollbacks due to miss-speculation have a significant impact on performance in relation to REPAS. On average, REPAS is a 6% faster than CRT_spec for SPLASH-2 benchmarks, although for Multimedia and Web Server applications CRTR_spec shows a performance similar to REPAS, benefited from the low synchronization exhibited by these applications.

The performance degradation reported for DCC when evaluated within a shared-bus is roughly 5% for several parallel applications [11]. However, as explained in Sect. 3, this overhead is increased when a direct network is used. As explained, the major source of degradation is related to the mechanism to assure the master–slave consistency which allows to avoid input incoherences.

As we can see in Fig. 7, REPAS is able to outperform DCC by 27% for scientific applications. However, for multimedia and web servers benchmarks, the performance exhibited by DCC is better than the performance of REPAS by 4%. The reason of this behavior is that, while in scientific benchmarks the cores share a lot of data, multimedia and web servers applications are multithreaded applications in which the sharing of data is reduced to some extent, so that the degradation because of the consistency window affects less the performance. In any case, we have to recall that while REPAS uses SMT cores to provide fault tolerance, DCC uses twice the number of cores than REPAS. This reduces the overall throughput of a system implementing DCC in more than a 100% over a non fault-tolerant base case.

Finally, as we can see in Table 3, REPAS is 20% faster than SMT-dual on average which, at the same time, is slower than CRTR_ sync and CRTR_spec by 10% and 17%, respectively. The performance degradation of SMT-dual is because of the bad interaction of different threads in the same core. While in REPAS and CRTR threads

Table 3 Average normalized execution time for the studied benchmarks

	REPAS (16 cores)	CRTR_sync (16 cores)	CRTR_spec (16 cores)	DCC (32 cores)	SMT-dual (16 cores)	DUAL (16 cores)
Normalized execution time	1.25	1.35	1.28	1.40	1.45	1.88

collaborate (LVQ, SVQ, BOQ) in SMT-dual threads compete against each other for the resources of the core affecting performance. In the same way, DUAL affects performance noticeably. This is because in DUAL, threads must be re-scheduled by the OS to be executed in each core (remind that we have 16 cores but 32 threads, 16 threads for every application). This adds an extra overhead of almost $2\times$ in the computation. As a final remark we can conclude that SMT approaches could benefit from a better performance than non-SMT approaches.

6.4 Performance in a faulty environment

We have shown that REPAS introduces an overhead in a fault-free scenario although outperforming several previous proposals. Nonetheless, REPAS guarantees the correct execution of shared-memory applications even in the presence of soft errors. The failures and the necessary recovery introduce an additional overhead that we study now.

Figure 8 shows the execution time overhead of REPAS under different fault rates normalized with respect to a non-faulty environment case. Failure rates are expressed in terms of faulty instructions per million of cycles per core. For a realistic fault ratio, the performance of REPAS is barely affected so, for this experiment, we have used fault rates which are extremely higher than expected in a real scenario in order to show the kindness of the proposed architecture.[3]

As we can see, REPAS is able to tolerate rates of 100 faulty instructions per million cycles per core with an average performance degradation of 1.6% in the execution time in comparison to REPAS in a non-faulty environment. Only when the fault ratio is increased to the huge (and unrealistic) amount of 1000 failures per million cycles, the performance shows a noticeable degradation of 8.6%. As expected, the performance degradation rises almost linearly with the increase of the fault ratio although it still allows the correct execution of all the studied benchmarks.

The time spent on every recovery varies across the executed benchmark. This time includes the invalidation of all the unverified blocks and the rollback (bypass the safe state of the slave thread to the master) of the architecture up to the point where the fault was detected. On average this time is 80 cycles. In contrast, other proposals such 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.

[3] As an example, a ratio of 10 failures per million cycles per core is equivalent to a MTTF of $3,125 \times 10^{-6}$ s for the proposed architecture.

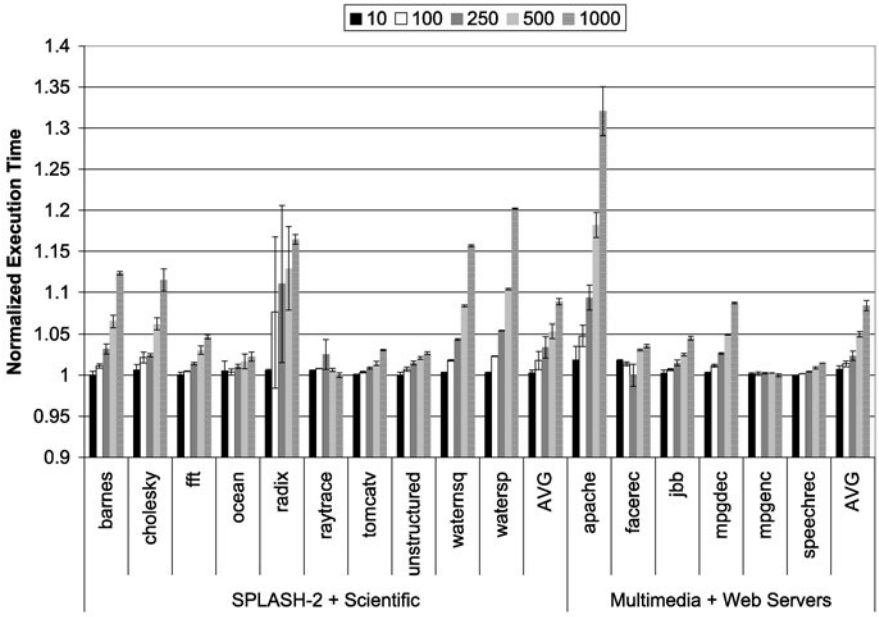

Fig. 8 REPAS overhead under different fault rates (in terms of faulty instructions per million per core)

6.5 Sharing unverified blocks

As initially implemented, REPAS does not allow the sharing of unverified blocks. This conservative constraint avoids the propagation of errors among cores. However, it is not expected that it imposes a high performance degradation, since the verification of blocks is quite fast (in the order of hundred cycles). On the contrary, DCC [11] 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. The mechanism is straightforward to implement: accept forward requests for blocks in unverified state. However, since we do not support checkpointing capabilities as DCC, to avoid unrecoverable situations, cores obtaining speculative data cannot commit. This way, if a fault is detected by the producer of the block, all the consumer cores can recover by flushing their pipeline in a similar way as is done when a branch is mispredicted. An additional disadvantage is that the producer of the block must send a message indicating whether the shared block is faulty or not, increasing the network traffic. Luckily, the sharing information is gathered from the sharers list as in a conventional MOESI protocol, so we do not need additional hardware to keep track of speculative sharings.

Finally, 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 the original value of the data block (for recovery purposes).

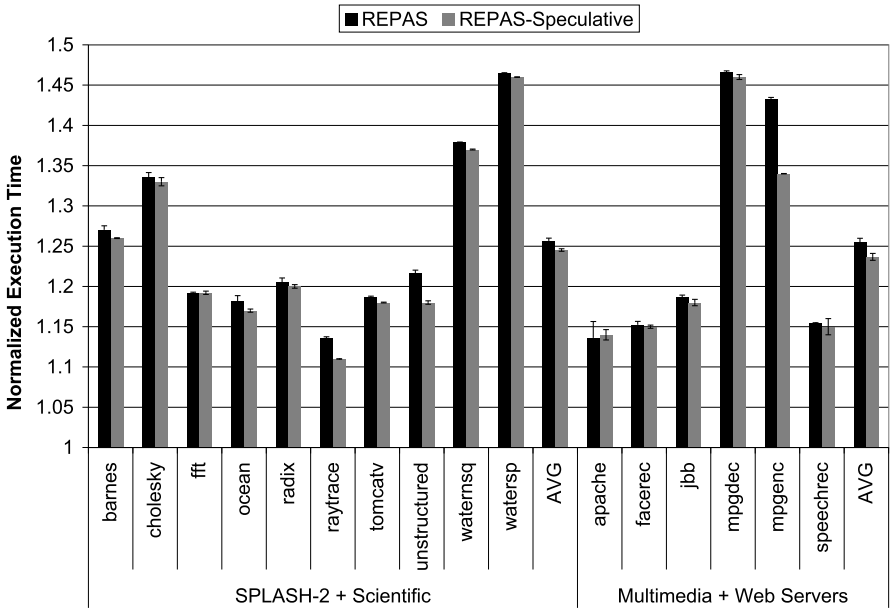

Fig. 9 Normalized execution time with and without the speculative mechanism

As we can see in Fig. 9, the performance improvement for the speculative mechanism is not noticeable. Just for benchmarks such as Ocean, Raytrace, Unstructured and MPGEnc, speculation obtains a slight improvement. Table 4 reflects that speculations are highly uncommon. Furthermore, if we consider the time to verification of speculative blocks it can be seen that, on average, we could benefit from around 100 cycles, although they cannot be fully amortized because the pipeline is closed at commit. This explains why speculative sharings do not obtain much benefit in REPAS. Overall, the speculative sharing mechanism seems inadequate for the studied benchmarks, since it is not worth the incremented complexity in the recovery mechanism of the architecture.

6.6 L1 cache size stress

An unverified block cannot be evicted from L1 cache since potentially faulty blocks would go out of the SoR (Sphere of Replication). In an environment with high pressure over the L1 cache, this can cause a performance degradation due to the unavailability of replacements to be completed. In this section, we study how REPAS behaves with different configurations.

It could be expected that the stress of cache size would impact negatively in the performance of REPAS. However, the results show that this forecast is not fulfilled. Figure 10 represents the execution time of REPAS for different L1 cache configurations. Each set of bars is normalized with respect to the case base with the same configuration.

Contrarily to expected, smaller caches do not degrade performance in REPAS but even improve it in comparison with the base case (1 KB, 2 KB and 4 KB perform

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

BENCHMARK	Speculations	Time to verification
Barnes	12860	92.5
Cholesky	5758	161.5
FFT	128	94.5
Ocean	13786	94.5
Radix	710	82
Raytrace	37031	92
Tomcatv	250	91
Unstructured	223524	107
Water-NSQ	1585	98
Water-SP	339	89.5
Apache	135	99.5
Facerec	0	–
JBB	877	94.5
MPGDec	0	–
MPGEnc	48997	123.5
Speechrec	0	–
AVG	–	101.875

better than the 64 KB configuration in comparison with the base case with the same configuration). The reason for this behavior is subtle but it can be easily explained if we attend to the REPAS mechanism. As we said before, a smaller cache penalize REPAS because of the increased latency of the L1 replacements. However, a smaller cache also penalizes the architecture due to the increased L1 cache miss ratio. The key point here is that, while in the base case the processor is stalled on a cache miss, in REPAS L1 misses (or master stalls in general) are used by the slave thread to continue executing program instructions, thus making forward progress.

Finally, an approach to allow the eviction of unverified blocks from L1 to L2 is to use a small VB (Victim Buffer). With this mechanism, L1 cache replacements of these blocks are performed out of the critical path. As we can see in Fig. 10, the VB improves the performance for 1 KB, 2 KB and 4 KB configurations. For the rest of them, there are not noticeable gains because the number of unverified blocks to replace from L1 cache is very low. Our experimental analysis states that, on average for all studied benchmarks, the optimal size for the VB is 14 entries, which we consider acceptable without spending too much hardware. Beyond that point there are no noticeable performance gains.

7 Conclusions and future work

Processors are becoming more susceptible to transient faults due to several factors such as technology scaling, voltage reduction, temperature fluctuations, process vari-

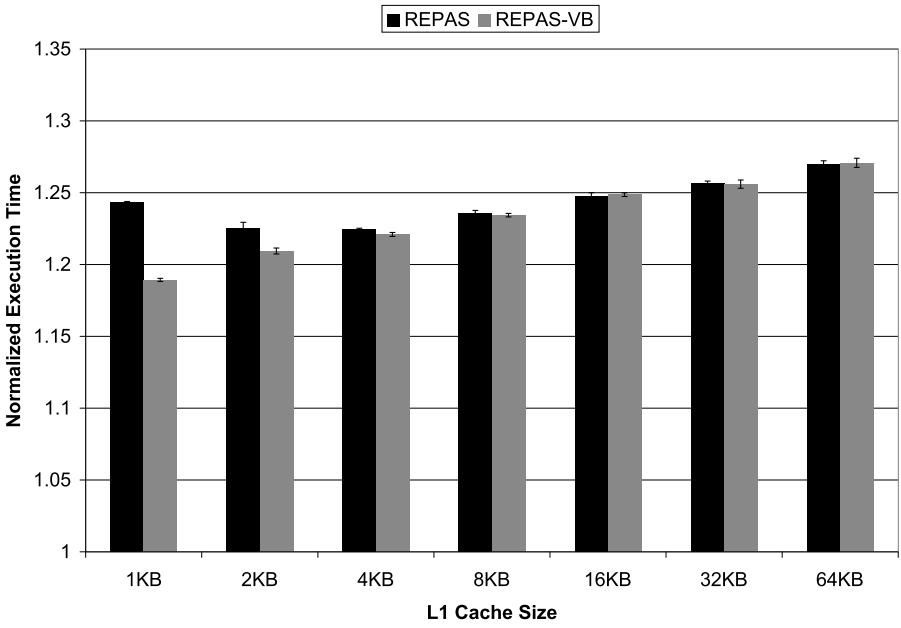

Fig. 10 Normalized execution time for different L1 cache sizes with and without Victim Buffer

ation or signal cross-talking. Although there are many approaches exploring reliability for single-threaded applications, shared-memory environments have not been thoroughly studied.

Proposals like DCC or Reunion use DMR (Dual Modular Redundancy) to provide fault tolerance in microarchitectures. However, they impose a $2\times$ hardware overhead, an unacceptable result for manufacturers which claim for a 10% maximum extra area impact. Hence, in this paper we propose REPAS: Reliable Execution for Parallel ApplicationS in tiled-CMPs, a novel RMT approach to provide transient fault detection and recovery in parallel and shared-memory applications.

While other proposals use large amounts of extra hardware, RMT architectures perform reliable computation by redundant thread execution (master and slave) in SMT cores. Therefore, the hardware overhead is kept low. However, the architectural support for shared-memory applications has remained under-explored so far. In our study, we show that atomic operations induce a serialization point between master and slave threads, a problem which may be minimized by means of speculation in the consistency model. Although this solution requires both a change in the way atomicity is implemented and a hardware increase to support the speculation, the degradation in low to medium contention benchmarks remains moderated. However, in scenarios with high contention the performance is severely affected. In REPAS we effectively avoid this overhead due to synchronization or miss-speculations by eager updates of the L1 cache.

We have implemented our solution in a full-system simulator and presented the results compared to a system in which no fault-tolerant mechanisms have been introduced. We show that, in a fault-free scenario, REPAS reduces the overall execu-

tion time down to 25%, outperforming CRTR, a traditional RMT implementation. We have also compared REPAS with DCC, showing some winnings in certain applications but losings in others. Nonetheless, REPAS uses half the number of cores than DCC, providing a better throughput. We have also evaluated the performance of REPAS in a faulty environment, showing an increase of just 2% of execution time with a huge fault ratio of 100 faults per million of cycles per core. This ratio is much higher than expected in a real scenario, so negligible slowdown is reported in a realistic faulty environment.

Finally, we have performed a L1 cache size stress in order to study the behavior of REPAS due to its inability to evict blocks from cache until verification. Results show that even with smaller cache sizes, the performance degradation of REPAS is kept in acceptable margins. Additionally, a Victim Buffer to hold unverified blocks has been used in REPAS showing slight performance improvement (up to 4%) for configurations which highly stress the L1 cache.

As part of our future work, we are studying new mechanisms to improve the collaboration between master and slave threads. One of our main ideas is to detect the execution of critical path instructions in order to increase the priority of the affected thread. This way, we could even improve the performance of REPAS.

Acknowledgements The authors would like to thank the anonymous reviewers for their detailed comments and valuable suggestions, which have increased the quality of this paper. This work has been jointly supported by the Spanish MEC and European Commission FEDER funds under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2009-14475-C04-02".

References

1. Bartlett J, Gray J, Horst B (1987) Fault tolerance in tandem computer systems. In: The evolution of fault-tolerant systems. doi:10.1.59.6080
2. Blundell C, Martin MM, Wenisch TF (2009) Invisifence: performance-transparent memory ordering in conventional multiprocessors. In: Proc of the 36th annual international symposium on computer architecture (ISCA '09), Austin, TX, USA, pp 233–244
3. Carretero J, Vera X, Chaparro P, Abella J (2008) On-line failure detection in memory order buffers. In: IEEE international test conference, ITC 2008, pp 1–10
4. Francisco J, Villa MEA, Garcýa JM (2016) Toward energy-efficient high-performance organizations of the memory hierarchy in chip-multiprocessors architectures. J Comput Sci Technol 6:1–7
5. Gniady C, Falsafi B (2002) Speculative sequential consistency with little custom storage. In: Proc of the 2002 international conference on parallel architectures and compilation techniques (PACT '02), pp 179–188
6. Gomaa M, Scarbrough C, Vijaykumar TN, Pomeranz I (2003) Transient-fault recovery for chip multiprocessors. In: Proc of the 30th annual int' symp on computer architecture (ISCA'03), San Diego, California
7. González A, Mahlke S, Mukherjee S, Sendag R, Chiou D, Yi JJ (2007) Reliability: fallacy or reality? IEEE MICRO 27(6). doi:10.1109/MM.2007.107
8. International VS, Weaver DL, Germond T (1992) The sparc architecture manual. doi:10.1.1.106.2805
9. Kumar S, Aggarwal A (2008) Speculative instruction validation for performance-reliability trade-off. In: Proc of the IEEE 14th int' symp on high performance computer architecture (HPCA'08), Salt Lake City
10. Kumar R, Zyuban V, Tullsen DM (2005) Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling. In: Proc of the 32th int'l symp on computer architecture (ISCA'05), Madison, Wisconsin

11. LaFrieda C, Ipek E, Martinez JF, Manohar R (2007) Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In: Proc of the 37th annual IEEE/IFIP int' conference on dependable systems and networks (DSN'07), Edinburgh, UK. doi:10.1109/DSN.2007.100
12. Li ML, Sasanka R, Adve SV, Chen KY, Debes E (2005) The alpbench benchmark suite for complex multimedia applications. In: Proc of the IEEE int symp on workload characterization, pp 34–45
13. Li ML, Ramachandran P, Sahoo S, Adve S, Adve V, Zhou Y (2008) Understanding the propagation of hard errors to software and implications for resilient system design. In: Proc of the 13th int' conference on architectural support for programming languages and operating systems (ASPLOS'08), Seattle, WA
14. Magnusson PS, Christensson M, Eskilson J, Forsgren D, Hallberg G, Hogberg J, Larsson F, Moestedt A, Werner B, Werner B (2002) Simics: a full system simulation platform. Computer 35(2). doi:10.1109/2.982916
15. Martin MMK, Sorin DJ, Beckmann BM, Marty MR, Xu M, Alameldeen AR, Moore KE, Hill MD, Wood DA (2005) Multifacet's general execution-driven multiprocessor simulator (gems) toolset. SIGARCH Comput Archit News 33(4). doi:10.1.1.109.5362
16. Martínez JF, Renau J, Huang MC, Prvulovic M, Torrellas J (2002) Cherry: checkpointed early resource recycling in out-of-order microprocessors. In: Proc of the int' symp on microarchitecture (MICRO'02), Istanbul, Turkey. citeseer.ist.psu.edu/martinez02cherry.html
17. Mastipuram R, Wee EC (2004) Soft error's impact on system reliability. Electronics Design, Strategy, News (EDN) pp 69–74. URL http://www.edn.com/article/CA454636.html
18. Mukherjee S (2008) Architecture design for soft errors. Morgan Kauffman, San Mateo
19. Mukherjee S, Kontz M, Reinhardt SK (2002) Detailed design and evaluation of redundant multithreading alternatives. In: Proc of the 29th annual int' symp on computer architecture (ISCA'02), Anchorage, Alaska
20. Olukotun K, Nayfeh BA, Hammond L, Wilson K, Chang K (1996) The case for a single-chip multiprocessor. In: Proceedings of the 7th international conference on architectural support for programming languages and operating systems. ACM Press, New York, pp 2–11. doi:10.1145/237090.237140. http://doi.acm.org/10.1145/237090.237140
21. Rashid M, Huang M (2008) 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
22. Reinhardt SK, Mukherjee S (2000) Transient fault detection via simultaneous multithreading. In: Proc of the 27th annual int' symp on computer architecture (ISCA'00), Vancouver, British Columbia, Canada
23. Ros A, Acacio ME, García JM (2010) A scalable organization for distributed directories. J Syst Archit 56(2–3):77–87
24. Rotenberg E (1999) 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, Wisconsin
25. Sánchez D, Aragón JL, García JM (2008) 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'08, Beijing, China
26. Sánchez D, Aragón JL, García JM (2009) Repas: reliable execution for parallel applications in tiled-cmps. In: Proc of the 15th int European conference on parallel and distributed computing (Euro-Par 2009), Delft, Netherlands, pp 321–333
27. Selse (2006) Selse ii final remarks. In: The 2nd workshop on system effects of logic soft errors
28. Smolens JC, Gold BT, Kim J, Falsafi B, Hoe JC, Nowatzyk AG (2004) Fingerprinting: Bounding soft-error-detection latency and bandwidth. IEEE MICRO 24(6). doi:10.1109/MM.2004.72
29. Smolens JC, Gold BT, Falsafi B, Hoe JC (2006) Reunion: Complexity-effective multicore redundancy. In: Proc of the 39th annual IEEE/ACM int' symp on microarchitecture (MICRO 39), Orlando, Florida, p 42. doi:10.1109/MICRO.2006.42
30. Taylor MB, Kim J, Miller J, Wentzlaff D, Ghodrat F, Greenwald B, Hoffman H, Johnson P, Lee JW, Lee W, Ma A, Saraf A, Seneski M, Shnidman N, Strumpen V, Frank M, Amarasinghe S, Agarwal A (2002) The raw microprocessor: a computational fabric for software circuits and general-purpose programs. IEEE MICRO 22(2):25–35
31. Vijaykumar T, Pomeranz I, Cheng K (2002) Transient fault recovery using simultaneous multithreading. In: Proc of the 29th annual int' symp on computer architecture (ISCA'02), Anchorage, Alaska
32. Wang NJ, Patel SJ (2006) Restore: Symptom-based soft error detection in microprocessors. IEEE Trans Depend Secure Comput 3(3). doi:10.1109/TDSC.2006.40

33. Wenisch TF, Ailamaki A, Falsafi B, Moshovos A (2007) Mechanisms for store-wait-free multiprocessors, pp 266–277
34. Woo SC, Ohara M, Torrie E, Singh JP, Gupta A (1995) The SPLASH-2 programs: characterization and methodological considerations. In: Proc of the 22th int' symp on computer architecture (ISCA'95), Santa Margherita Ligure, Italy
35. Ziegler J, Lanford WA (1981) The effect of sea level cosmic rays on electronic devices. J Appl Phys 52:4305–4312
36. Zielger JF, Puchner H (2004) SER-History, Trends and Challenges. Cypress Semiconductor Corporation