Distrib. Comput. (2015) 28:75–90
DOI 10.1007/s00446-014-0215-6

Soft-error mitigation by means of decoupled transactional memory threads

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

Received: 18 April 2012 / Accepted: 8 April 2014 / Published online: 29 April 2014

Abstract CMOS scaling exacerbates hardware errors making reliability a big concern for recent and future microarchitecture designs. Mechanisms to provide fault tolerance in architectures must accomplish several objectives such as low performance degradation, power consumption and area overhead. Several studies have already proposed fault tolerance for parallel codes. However, these proposals are usually implemented over non-realistic environments including the use of shared-buses among processors or modifying highly optimized hardware designs such as caches. Our attempt to face this multiple challenge is an architectural design called LBRA (Log-Based Redundant Architecture). Based on a Hardware Transactional Memory architecture, LBRA executes redundant threads which communicate through a pair-shared virtual memory log allocated in cache. Our initial version of LBRA executes these redundant threads in SMT cores. To avoid the performance penalty inherent to this architecture, we propose to decouple their execution in different cores, solving the inter-core communication by means of a log buffer empowered by a simple prefetch strategy. Simulation results using a variety of scientific and multimedia applications show that the execution time overhead of our best design is less than 7 % over a base case without fault tolerance. Additionally, we show that LBRA outperforms previous proposals that we have implemented and evaluated in the same framework.

Keywords Reliability · Fault tolerance · Soft-errors · Hardware transactional memory

D. Sánchez (✉) · J. M. Cebrián · J. M. García · J. L. Aragón
Computer Engineering Department, Facultad de Informática, University of Murcia, 30100 Murcia, Spain
e-mail: dsanchez@ditec.um.es

J. M. Cebrián
e-mail: jcebrian@ditec.um.es

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

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

1 Introduction

Increasing device density offers designers the opportunity to place more functionality per unit area. Billions of transistors are available within a single chip but existing techniques cannot further exploit exploit the ILP (Instruction Level Parallelism). The solution manufacturers have adopted is to use this vast amount of transistors for the integration of large caches and many cores into the same chip or CMP (Chip Multi Processor) [22] to exploit the TLP (Thread Level Parallelism). These CMPs are usually implemented around a shared-memory environment in which some levels of the memory hierarchy are private but coherent (typically the L1 cache), while the rest of the levels are shared. First implementations connect the cores by means of shared-buses and/or crossbars. However, as the number of cores grows, these networks become non-scalable due to area and power constraints [15]. The most promising approach to provide both efficiency and scalability are directory-based cache coherence protocols [8,10] which operate in direct-network environments. Therefore, it seems that future architectures which include many cores in a single chip will be designed according to these parameters [32,33].

Unfortunately, the scaling of device area has been accompanied by, at least, two negative consequences: a slowdown of both voltage scaling and frequency increase, as a result of the decreased scaling of leakage current (*Dennard Scaling*) as compared to area scaling [5,12,31]; and a shift to

probabilistic designs and less reliable silicon primitives, due to static [6] and dynamic [7] variations.

These trends forecast that the performance and cost benefits from area scaling will be hindered unless scalable techniques are developed to address power and reliability challenges. In particular, it will no longer be possible to operate all on-chip resources, even at the minimum voltage for safe operation, due to power constraints (*Dark Silicon* [11]) without an exponential sensitivity increase to transient faults.

Being reliability a major concern for hardware architects, several mechanisms to detect and recover from faults have been already implemented in microarchitectures. This is the case of ECC (Error Correcting Codes), which is nowadays applied in large CAM (Content Addressable Arrays) arrays such as caches or RAM (Random Access Memory) memories. Unfortunately, ECC cannot be extensively used through all hardware structures. On the contrary, architectural-level mechanisms provide a more flexible framework in which multiple hardware structures are covered in comparison to cycle-level techniques which are focused on single units.

In this paper we present a new architectural-level proposal called LBRA: A Log-Based Redundant Architecture for reliable parallel computation. With LBRA, we explore the use of a HTM (Hardware Transactional Memory) system to build a fault tolerant architecture. The main idea is to execute redundant copies of the same software thread in two different hardware contexts which check for computation mismatches.

LBRA provides high flexibility, allowing the programmer to manually declare which areas of the program may be protected. Program instructions in these areas are divided into virtual execution groups that we call *pseudo-transactions* (p-XACTs) or chunks [9]. The master thread executes p-XACTs as regular instructions but, additionally, it keeps the results of its progress in a pair-shared log. By means of this log, the slave verifies that the results produced by the master are correct. We provide a highly decoupled environment since the master is allowed to execute multiple p-XACTs without verification, which is carried out off the critical path by the slave thread. This high decoupling allows the latencies due to memory or inter-core communication to be hidden.

A preliminary version of this work was presented in [26]. The major contributions of this paper are:

- An architecture design which, on top of Hardware Transactional Memory system and SMT (Simultaneous Multi Threaded) cores, includes fault tolerance measures in a parallel point-to-point network environment.
- In addition to [26], we study the implications of running redundant threads in different cores. Since most of the performance degradation is caused by the resource contention inherent to SMT architectures, with this measure we increase the efficiency of the proposed mechanism.
- A set of hardware mechanisms to reduce and/or hide the inter-core communication latency. These mechanisms include a log buffer combined with a simple prefetch strategy and slight modifications of coherence actions.
- A detailed comparison among the proposed architecture design and state-of-the art previous proposals within the same framework. For this evaluation, we make use of a great variety of parallel benchmarks executed in both SMT and non-SMT cores.

The remainder of the paper is organized as follows: Sect. 2 summarizes relevant previous approaches and motivates this work. Section 3 briefly introduces Hardware Transactional Memory and explains how it can be adapted for fault tolerance purposes. In Sect. 4 we discuss the implementation details of LBRA in 2-way SMT cores, whereas in Sect. 5 we extend it to redundant regular cores as a way to reduce the performance penalty inherent to the use of simultaneous multithreading. The evaluation setup and analysis are described in Sect. 6. Finally, Sect. 7 summarizes the main conclusions of this work.

2 Background and related work

As we move into the late CMOS technologies, system reliability is compromised. Among different events which can lead to unsafe computations we find: (i) process variability, which causes heterogeneous or erratic behavior of identical hardware components in the same chip; (ii) increased aging/wear-out due to extreme operating conditions, which leads to permanent hardware faults; and (iii) an increased sensitivity to soft/transient faults, which grows exponentially and is the subject of our study.

Transient faults are radiation-induced faults which can manifest themselves as transient errors. Radiation induced events include alpha-particles from packaging materials and neutrons from the atmosphere. It is well established that the strike of particles such as alpha particles or neutrons over a logical device can overwhelm the circuit inducing its malfunction. For instance, an SRAM (Static Random Access Memory) cell, which forms cache memories, could alter its value from 0 to 1 or vice versa due to one of these events. In the fault model we assume a system prone to transient faults which can affect all pipeline structures, which our mechanism would be able to detect and correct. These faults can cause either single or double bit flips. However, array structures such as the register file or caches are assumed to be fault-free because of the use of ECC codes.

While there are different approaches to provide both detection and correction of transient faults, one of the most studied group of techniques are those based on redundant execution.

Table 1 Main characteristics of several redundant architectures

	SoR	Synchronization	Input replication	Output comparison
SRT(R) [24,34] CRT(R) [13,21]	Pipeline, registers	Staggered execution	Strict (queue-based)	Instruction by instruction
Reunion [28]	Pipeline, registers, L1Cache	Loose coupling	Relaxed input replication	Fingerprints
DCC [16]	Pipeline, registers, L1Cache	Thousands of instructions	Consistency window	Fingerprints, checkpoints
HDTLR [23]	Pipeline, registers, L1Cache	Thousands of instructions	Sub-epochs	Fingerprints, checkpoints

2.1 Fault tolerance by means of redundant execution

Redundant execution is a traditional methodology to detect transient faults. Under this methodology a redundant core/thread which is fed with the same inputs as the checked core/thread should produce the same outputs. When the redundant one produces a different result, a transient fault is detected.

To classify different redundant approaches we can use four main characteristics: sphere of replication, input replication, output comparison and synchronization.

- **Sphere of replication (SoR)** [24]. The SoR determines the components in the architecture whose functionality is replicated, i.e. all faults that occur within the sphere will be detected.
- **Input replication**. In order to assure that redundant copies perform exactly the same work, they must be provided with the same view of the memory. If not, even if redundant executions perform correct computations, they may follow different paths due to data races.
- **Output comparison**. The output comparison defines error detection latency. Generally, lower latency increases the pressure over the hardware, e.g. comparing the register updates, while higher latency increases the recovery time after a fault.
- **Synchronization interval**. Which defines how often output comparison takes place.

2.1.1 Previous work

One of the first proposals for full redundant execution is Lockstepping [2], a mechanism 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) [24], SRTR [34], CRT [21] and CRTR [13] was proposed. A more recent study [25] showed that their ability to deal with faults in shared-memory environments seriously degrades performance even in a fault-free scenario.

Another set of studies has been directly applied to a multiprocessor domain. Reunion [28] describes a mechanism in which redundant threads access memory independently (relaxed input replication). Due to data races, relaxed input replication leads to divergences in the memory values observed by the redundant threads (input incoherences). These divergences are treated as faults, inducing a serialized execution (very similar to lock-stepped execution) between redundant cores, and degrading the overall performance of the architecture. Dynamic Core Coupling (DCC) [16] reduces the overhead of Reunion by providing a decoupled execution of instructions, having larger comparison intervals (thousand of instructions) and reducing the network traffic. At the end of each interval, the architectural state of redundant pairs is interchanged and, if no error is detected, a new checkpoint is taken.

In the same fashion, Highly-Decoupled Thread-Level Redundancy (HDTLR) [23] is proposed, also using a shared bus. HDTLR architecture is similar to DCC in the sense that the recovery mechanism is based on checkpoints which reflect the architecture changes between *epochs*. In addition, memory updates, which are buffered in a PCB (Post Commit Buffer), are not visible to the L2 until verification. However, in HDTLR each redundant thread is executed in a different hardware context (*computing wavefront* and *verification wavefront*), maintaining coherency independently. This way, the consistency window is avoided. Unfortunately, the asynchronous progress of the two hardware contexts may lead to memory races, which result in different execution outcomes. These events are masked by the architecture as transient faults. 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. Our LBRA approach provides decoupled execution as DCC and HDTLR while using a more scalable network. Furthermore, our mechanism does not require modifications on optimized structures such as caches. Table 1 summarizes the characteristics of all these previous proposals.

Another piece of work is based on checkpointing techniques. Rebound [1] proposes a coordinated local check-

pointing scheme for recovery purposes in order to solve the scalability problems inherent to global checkpointing approaches. Finally, another approach towards fault detection follows a scheme based on symptoms [17] which is inspired by ReStore [35]. This study presents a characterization of how errors affect either application or OS behaviour 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. If a fault is detected, the execution is rolled-back to a previous safe state. However, this approach cannot provide a solution for those errors which do not modify the behaviour of applications, such as those affecting values but not control flow. Furthermore, it still requires the use of rollback mechanisms such as those previously cited.

Similar to our approach, FaultTM [37] is based on hardware transactional memory. In FaultTM, programs are executed redundantly following a lazy update policy. This means that only after the execution is proved to be correct, the memory updates are made visible. This has two main advantages: (i) the redundant thread perceives memory as its checked pair thread since no modifications were made, satisfying the input replication; (ii) faults are not propagated to the rest of the system, making recovery a very fast process. Nonetheless this presents additional problems that LBRA aims to solve: (i) redundant executions are tightly coupled and, frequently, execution is stalled because one execution thread must wait until the execution of its pair; (ii) after a successful verification, buffered values must be made visible to the rest of the system. This involves an increased pressure over the memory hierarchy affecting system performance; (iii) the mechanism is not suited for parallel, shared-memory applications, since lazy update policies of big transactions lead to transaction conflicts which are solved by aborts. Based on this pros and cons, in LBRA we chose to implement an eager update policy which allows for decoupled execution and low-overhead for parallel applications.

2.1.2 Some previous work in detail

In this section we present a detailed analysis of two of the previous approaches we will use for comparison purposes: REPAS [25] and DCC [16].

Fault tolerance by means of SMT cores

REPAS is an architecture based on the SRTR/CRTR [13,34] family. In this approach, the input replication is assured by means of the LVQ (Load Value Queue) and output comparison is achieved by means of the SVQ (Store Value Queue), in which the master thread keeps memory values loaded and stored, respectively. This way, the delay between master and slave thread is given by the buffering capabilities of these queues (usually, in the order of hundred of instructions), which allows to partially hide the latencies of cache misses, among others. The major problem addressed by REPAS is the potential violation of the consistency when executing parallel benchmarks in SRTR/CRTR. These inconsistencies may arise when two different master threads try to acquire the access to a shared area by means of locks. Since master threads do not update memory until verification, with no additional support, two threads may enter the same critical section without noticing. To avoid this, REPAS proposes to update memory before verification ruling out this possibility. To avoid the propagation of potentially faulty blocks, unverified data is marked in the cache and cannot be shared. The major drawback of REPAS is precisely the complexity it introduces in the pipeline, e.g. the LVQ and SVQ queues.

Fault tolerance by means of redundant cores

DCC (Dynamic Core Coupling), on the other hand, 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. At the end of a checkpoint interval (in the order of 10,000 cycles), redundant cores interchange their architectural state, which are checked to detect errors. If so, the architectural state of the whole machine is restored to a previous checkpoint and the execution is restarted from that point.

In DCC, both cores are allowed to access the memory subsystem. Therefore, several measures are introduced to assure both coherence and consistency. At the coherence level, only one of the two cores is responsible for sharing unverified data, something which involves several changes in the coherence protocol. But the major obstacle is found at the consistency level. Since both cores access memory, an intervening store from another core could cause the value loaded by the same dynamic load to be different. To overcome this issue, DCC implements a consistency 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. When an invalidation or upgrade request reaches a core, it interchanges its age with its pair. If a mismatch is found, it means that the request could cause an incoherence. Therefore, the request is nacked.

The fundamental issue in DCC is that it is based on a shared-bus as interconnection network. As we noted previously, the area required by a shared-bus or a crossbar as the number of cores grows, increases to the point of becoming impractical [15]. A direct-network introduces an indirection with undesirable impacts over the checkpoint creation and, foremost, over the age table mechanism. Specifically, in a shared-bus environment, the mechanism implies a three-way interchange, while in a direct-network the number of hops increases to four, as we can see in Fig. 1.

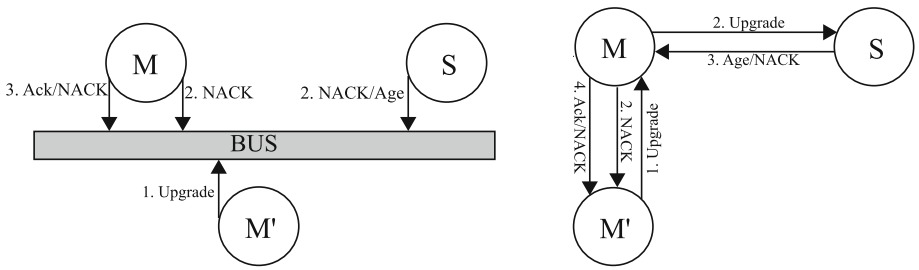

Fig. 1 DCC master-slave consistency

(a) Shared-bus **(b)** Direct-network

3 HTM support for reliable computation

3.1 Brief summary

Hardware Transactional Memory (HTM) is an alternative to traditional lock-based synchronization mechanisms. The main idea is to generalize LL/SC (Load Link/Store Conditional) primitives to provide atomic accesses not to one but to several independent memory locations. This allows the programmer to produce parallel code more easily which, hopefully, is also more efficient. To this end, instead of relying on fine-grain and difficult-to- program critical sections, HTM introduces the concept of coarse-grain units of work called *transactions*. Within a transaction, multiple load and store instructions are executed atomically and in isolation with instructions of other transactions. When two or more concurrent transactions access the same data block and, at least, one of the accesses is a write, a transactional conflict occurs. This means that the atomicity of the transactions cannot be assured. To preserve the atomicity, HTM systems must implement mechanisms to detect transactional conflicts and to solve them, which usually implies the abort of one or more of the involved transactions.

In this paper we propose to build a redundant, fault-tolerant system based on a HTM implementation called LogTM-SE [38]. The first thread (*master thread*) executes the program instructions divided into different transactions. But, additionally, it also performs three actions: (i) tracking producer/consumer dependencies with other threads in the system by means of the conflict detection support provided by LogTM-SE; (ii) maintaining a log of the memory actions it has taken; and (iii) generating a signature (*verification signature*) which summarizes the results of the performed work. The redundant thread re-executes the transactions, fetching memory values from the log to satisfy the input replication. Store instructions, though, do not update memory since they were already exposed by the master thread. Finally, the redundant thread checks for a positive match of the verification signature. In case of a negative match (meaning that the signatures are not identical), a fault is detected. This event triggers a fault recovery mechanism which involves: (i) a rollback to a previous safe state of the affected core; and (ii) the rollback of potentially affected consumers of the faulty computation, which were tracked previously.

In the rest of the section we will make an overview of how we adapt LogTM-SE to fulfill the required reliability requirements such as the input replication and the output comparison (Sect. 3.2. Finally, we will discuss how to track inter-thread dependencies (Sect. 4.4.1).

3.2 Enforcing reliability requirements

3.2.1 Input replication

In RMT (Redundant Multi-Threading) approaches like LBRA the input replication defines how redundant threads observe the same data. Since master and slave thread execution is not lockstepped [2], the execution of redundant memory instructions would probably lead to input incoherences. To solve this issue, we extend the functionality of the log already proposed in LogTM-SE. The log is simply a memory space allocated in virtual memory which contains the history related to memory operations in the format $< address >< value >$.

In LBRA we use this log to keep track of the data values that the master thread accesses. This way, slave data load instructions are served through the log where they obtain the same values as its master-pair, thus avoiding input incoherences. Notice that, as the log is written at instruction commit, it will only keep instructions of the correct execution path and in program order.

3.2.2 Output comparison

In our LBRA approach, we define the output comparison granularity at pseudo-transaction (p-XACT) level.[1] A p-XACT defines the unit of work which is considered to be either incorrect or correct, depending on whether faults have been detected within its execution or not.

[1] We could increase fault detection granularity to memory operations as well, but this requires a bigger log. Refer to Sect. 4.1.1 for more details.

The semantic and execution of a p-XACT differ from a regular transaction in LogTM-SE. Firstly, p-XACTs are dynamically created at execution time, whereas traditional transactions are manually coded in the application. Secondly, as opposed to LogTM-SE transactions, p-XACTs do not ensure isolation and/or atomicity of the executed instructions. This means that, in our approach dirty memory blocks are shared as in a non-transactional environment, relying on other synchronization mechanisms (such as locks or barriers) to ensure the correct behavior.

During the execution of the p-XACT, the master thread computes what we call the *Verification Signature*. This signature summarizes the computation performed during the execution of the p-XACT. When the p-XACT finishes, the Verification Signature is allocated in a special hardware structure. We call this step the *commit* of the p-XACT. In the same way, the slave thread computes its own signature. When the p-XACT ends, both signatures are checked, a step which we call *consolidation*. If both signatures match, the p-XACT execution is considered fault free. If not, a fault is detected triggering the recovery mechanism.

The Verification Signature is a CRC-32 hash code similar to the one proposed in Fingerprinting [29] and employed in DCC [16] and Reunion [28]. The signature is obtained by hashing the instruction results as they complete. Based on Hamming codes, these signatures have an aliasing ratio, and therefore a probability of undetected or masked faults, of 2^{-32} ($2.3 * 10^{-10}$), which has been traditionally considered above commercial requirements [16,23,28].

3.2.3 Sphere of replication (SoR)

In LBRA, the SoR includes the processor pipeline as well as first level data caches (L1). This means that all faults affecting other parts of the chip must be protected by other mechanisms. Additionally, we enforce cache data integrity by assuming SECDED (Single Error Correction, Double Error Detection).

3.2.4 Synchronization interval

Redundant threads in LBRA are highly decoupled. We support this feature by allowing redundant threads to run up to thousands of instructions apart in time. To this end, we allow the execution of several p-XACTs in-flight without the need of verification

3.2.5 I/O events

Regarding I/O events, due to its unrecoverable nature (e.g. print screen), a mechanism which detects the faults before propagation must be used. In our approach, we rely on a mechanism like lockstepped execution [2]. Before an I/O event or system call is about to be executed, we need to make sure that the current architectural state is correct. To this end, master and slave threads have to be synchronized and all the inflight transactions have to be consolidated. After that, the execution of redundant threads is changed to lockstep mode. In lockstep or cycle-by-cycle mode, both master and slave threads execute the same program instructions at the same cycle being their outputs compared by means of a checker circuit. In case of mismatch, the affected instructions are reissued. In case of a successful check, the instructions results are committed.

There are different ways to implement lockstep execution mode, but we refer the reader to the description of two commercial products such as the Compaq Nonstop Series [3] and the IBM S/390 G5 microprocessors [30] for further details. Finally, after the I/O event or system call is solved the execution mode is changed to the normal one described along this paper.

4 LBRA implementation details

One of the major drawbacks in RMT approaches is the penalty resulting from the time spent on the synchronization between redundant threads. This penalty includes both the effect of stalling execution and comparing architectural states. As a measure to pay off this penalty, previous approaches make use of long checkpoint intervals. However, this increases detection latency. Our approach eliminates this penalty independently of whether synchronizations are common or not. To achieve this, we use a decoupled approach and the ability to execute multiple p-XACTs before verification. Each p-XACT has a small hardware context which includes a Master and a Slave Log pointers, a Verification Signature, and the Producer and Consumer registers. These

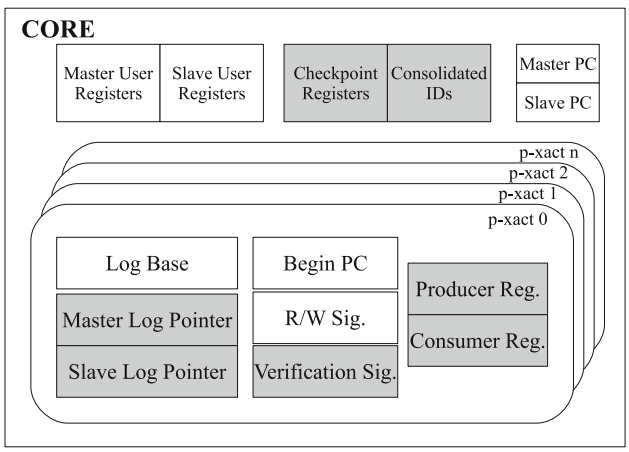

Fig. 2 LBRA hardware overview. *Shadowed boxes* represent the added structures

Table 2 Alternatives in log content for loads and stores

	Address	Value	Old-value	Provides
Loads	Yes	Yes	–	Input replication; Fault detection in address calculation
	No	Yes	–	Input replication
Stores	Yes	Yes	Yes	Fault detection in address calculation and value, fault recovery
	Yes	No	Yes	Fault detection in address calculation, fault recovery

hardware additions are depicted in Fig. 2 and explained in following sections. All these memory structures are assumed to be especially protected by means of cycle-level techniques such as 10-T transistors [14].

4.1 Accessing the log

To provide access to the log, both master and slave threads should share the same memory space. To this end, unlike in true SMT threads, master and slave threads appear to the OS as a single one.

The master thread writes in the log through the *Master Log Pointer*. For every memory operation, the master generates a new write instruction whose destination address is indicated by this pointer. This new write enables the redundant thread to satisfy the input replication and output comparison as explained in Sect. 3.2. Memory operations are logged at commit stage, therefore the content of the log is structured in program order.

The slave accesses to the log require a special treatment. In order to ensure the input replication, each load access must be redirected to the log. For that purpose, at memory access time, destination addresses of loads are switched with the *Log Slave Pointer* which indicates the location of the memory value previously read by the master thread. Then, the memory access is performed as usual and the log pointer is set to the next entry in the log.

In the case of stores, the mechanism differs slightly. Since slaves do not update memory, their stores become reads to the log. For this reason, the destination address is switched with the Log Slave Pointer one and the data value is retrieved from the log to perform the corresponding checks.

These pointers are assumed to be protected by means of circuit-level mechanisms such as the use of 10-T transistors.

4.1.1 Log content and fault detection granularity

The size of the log is a major concern in our approach since if it grows too much, it decreases the effective cache space damaging the performance of the application. In this section we discuss how to decrease the size of the log by reducing the amount of data to store, something which also affects the detection granularity. The different alternatives can be seen in Table 2.

Faults in load addresses

To satisfy input replication, it is mandatory to include in the log every data value read by the master thread. However, the address of the load is optional. If we include it, we could detect faults affecting address calculation. But this presents two major drawbacks. First the log size increases. Second, we increase the hardware pressure by adding an additional master-slave check on every load. Since our first goal is to reduce performance penalty, we choose to store the minimum information possible, i.e. only data values, and to rely in the consolidation process to determine the correct execution of all the p-XACT.

Faults in store addresses

Likewise, we try to reduce the information we keep from stores to decrease the log size as much as possible. In order to recover from a fault, we rely on the LogTM-SE handler which restores modified memory values. For this purpose, we need to keep the address and the old value for every memory update in the log. Additionally, we could keep the current value to be stored. If so, a fault in the calculation of this value could be detected when the slave thread accesses the log. However, for the same reasons as for loads, we avoid keeping the new value, waiting for faults to be detected at consolidation phase.

4.2 Circular log

LBRA provides a high decoupled execution of redundant threads. This is achieved because the forward progress of the master is rarely interrupted since the latencies inherent to the verification process are virtually hidden. For this purpose, the master thread is allowed to execute and commit several p-XACTs without verification. In parallel with this, the slave thread checks the correct execution of already committed p-XACTs and, as a final step, performs consolidations.

In order to allow multiple p-XACTs to be committed without verification, each one needs its own architectural support. This support includes its own R/W and verification signatures together with log pointers among others, as depicted in Fig. 2. The amount of extra hardware is determined by the maximum number of in-flight p-XACTs allowed.

However, the major implication derived from this support does affect the management of the log. LogTM-SE only allows one transaction to be executed at a time. After

commit, signatures are cleared and log pointers are reset. Thus, the next XACT will start writing the log from the beginning of the reserved memory space, as we can see in Fig. 3. However, in LBRA the log must be preserved for the slave thread. Therefore, instead of resetting log pointers, after the commit of a p-XACT, the following one starts writing from the last entry used by the previous p-XACT, as we can see on the right in Fig. 3. When the pointer reaches the limit of the memory reserved for the log, it is set to the beginning of the latter. In case the master thread runs out of p-XACTs and/or the log space due to the slow execution of the redundant thread, its execution must be stalled. We will see the performance impact of these stalls in the evaluation section.

In case the slave thread crashes due to a soft error, the corresponding p-XACT would never be released and the master thread could be indefinitely stalled. To avoid such a problem we make use of a watchdog timer which triggers the recovery mechanism in case of a lack of forward progress.

4.3 In-order consolidation

In our approach, memory blocks are updated in place (L1 cache) and are allowed to be shared even before consolidation takes place. This eager approach allows fast commits in the common case (a fault free environment). However, this mechanism affects the consolidation order of p-XACTs since, if additional mechanisms are not implemented, faults could be spread all around the system.

It is clear that if a p-XACT p_i has consumed data produced from another p-XACT p_j, the consolidation of p_i cannot take place before the consolidation of p_j. Otherwise, a faulty block produced in p_j would be silently consolidated in p_i. To keep track of these dependencies we introduce the Consumer and Consolidated-Ids registers, as explained in Sect. 4.4.1, which gather the information provided by the coherence protocol. To achieve this in our approach, memory coherence messages are extended to include the p-XACT identifier providing the data, which are used by the requester to fill the Consumer register, and the last consolidated p-XACT identifier.

The in-order consolidation process works as follows. After completing the verification of state, the slave thread checks the Consumer vector for the current p-XACT. If it is empty, it means that this p-XACT has not consumed data from any other p-XACT, so the consolidation process may take place without any additional checks. If the Consumer register is not empty, then, for every dependence, the slave checks if the producer p-XACT has already been consolidated by checking the Consolidated register. If all the dependencies satisfy this condition, then the p-XACT is finally consolidated. If not, we initiate a look-up mechanism. The slave thread requests its producers to supply the last consolidated *id* until all the dependencies are satisfied.

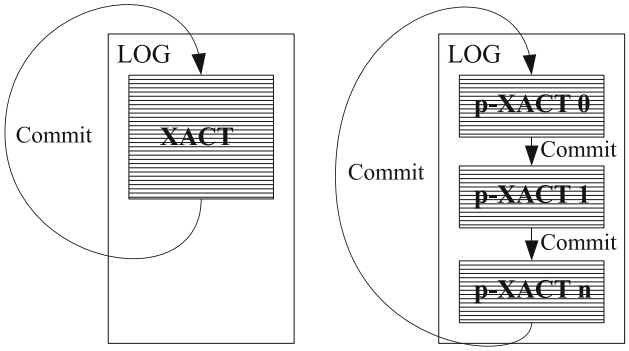

Fig. 3 LogTM-SE and LBRA log management

4.3.1 Cycle avoidance

There exists a danger of deadlock in the consolidation process if we allow cycles to be formed. For example, let us consider the case in which p_i is the producer of p_j which, at the same time, is the producer of p_k and, finally, p_i consumes data from p_k. In this case, none of the three p-XACT could be consolidated since a cycle has been created. Although this case is rare, we need to present a mechanism to avoid it.

Our goal is to create a DAG (Directed Acyclic Graph). DAGs assure that a topological order exists although this order, in general, is not unique. Therefore, we implement a simple policy: we disallow situations in which a p-XACT is both producer and consumer of other p-XACTs at the same time. When a master thread which is already a producer receives data produced by a p-XACT, the active p-XACT is forced to commit and a new one is started before consuming these data. Likewise, if a consumer p-XACT is requested to provide data (becoming a producer), it is forced to commit and the dependence is created in a new p-XACT. This guarantees that no cycles can be created avoiding consolidation deadlocks.

4.4 Fault recovery in LBRA

Upon a fault detection the recovery mechanism is triggered. In our approach, this mechanism is taken by a combination of both software and hardware processes for local and global recovery which act on the youngest p-XACT of the core. The correctness of the proposed mechanism is proved since dependencies form a DAG, so a topological order can be established.

4.4.1 Tracking inter-thread dependencies

As memory values are allowed to be shared despite of modifications, potential faults could be spread across the system. To keep track of shared blocks among different p-XACTs we

use the conflict detection mechanism provided by LogTM-SE.

LogTM-SE provides eager conflict detection by means of the coherence protocol and decoupling the mechanism from caches by using R/W signatures. External requests arriving at a core are checked through these signatures and, on a possible conflict[2], requests are NACKed. What we propose is to use these signatures to maintain a pair of per-transaction registers called *Producer Register* and *Consumer Register*, see Fig. 2. The Producer and Consumer registers keep the transaction identifiers involved in the data sharing of all the cores in the system.

The proposed mechanism works as follows. A core receiving a forward request checks its write signatures from all active p-XACTs (those which have been already committed by the master or are still in execution). For a positive match in an active p-XACT, the core updates the *Producer Register* storing the transaction *id* for the involved core. In the same way, the requester of the block, when obtaining a response, updates its *Consumer Register* indicating the core and transaction *id* produced by the previously-obtained block. All the required information is obtained from memory request messages.

The functionality of these registers is twofold. First, when a fault is detected, the *Producer Register* is used in the recovery process to abort all the p-XACTs involved, since their states are potentially corrupted, as we will see later. Secondly, the *Consumer Register* is used to provide an order in the consolidation mechanism, needed to avoid SDCs (Silent Data Corruptions), as we saw in Sect. 4.3.

4.4.2 Local recovery

The local recovery is the rollback to a safe state previous to the execution of a faulty p-XACT in a processor. For this process we rely on the software approach proposed in LogTM-SE to abort transactions. This software writes back the old values to their appropriate addresses from the log. After that, the transactional hardware of the current p-XACT is reset. Additionally, if this mechanism were triggered by an external request, it would acknowledge the requester.

4.4.3 Global recovery

Given the fact that blocks are shared before consolidation, potential faults could be spread among cores. In case that a p-XACT is detected as faulty, the recovery mechanism is also responsible for notifying its consumers (including the lower p-XACTs of the same node). Thus, upon fault detection, the mechanism carries out different actions, depending on whether the affected p-XACT is either a consumer or a producer:

- **Consumer**. If the current p-XACT is a consumer, the produced values were not previously shared, therefore potential faults were not spread outside the core. In this case, a local recovery of the current p-XACT is performed. If the recovery process is initiated by an external request, an ACK is sent back to the source of the request. Likewise, the mechanism is repeated for the upper p-XACT.
- **Producer**. In this case, the process sends a rollback request to all the consumers of the current p-XACT (indicated by its Producer Register). When all the ACKs are collected, a local recovery of the current p-XACT is initiated and this mechanism is repeated for the upper p-XACT.

The recovery process finishes when all the p-XACTs in a core have been recovered. As as a final step, the register checkpoint is written back to both master and slave, and the execution is resumed. Hence, on the one hand, the described method assures that, for a faulty core, a younger p-XACT is "undone" before an older one. On the other hand, consumers are restored before producers, in case of dependencies among different cores. We can see an example of a fault recovery in Fig. 4.

5 Performance enhancements via spatial thread decoupling

So far we have focused our discussion on the execution of redundant pair threads in 2-way SMT cores. The benefits of having both threads in the same core include a better use of the resources, the hiding of latencies (e.g., cache misses) and the avoidance of additional hardware. However, the major problem this design entails is the performance degradation due to resource contention associated with SMT architectures. To avoid this penalty and to provide a low-overhead solution, we propose to execute redundant threads in different cores rather than in different SMT contexts.

5.1 Decoupling thread execution into different cores

The support for the execution of redundant threads in different cores is pretty straightforward given the software nature of threads. This support includes the creation of the redundant thread by the OS and the initialization of all the registers and related hardware structures including the log pointers.

However, the major drawback we must face in this new scenario is inter-core communication. Proposals like Reunion [28] rely on the use of dedicated fast lines to communicate redundant cores since the latency of the messages

[2] A conflict occurs when an address appears in the write-set of two transactions or the write-set of one and the read-set of another [38].

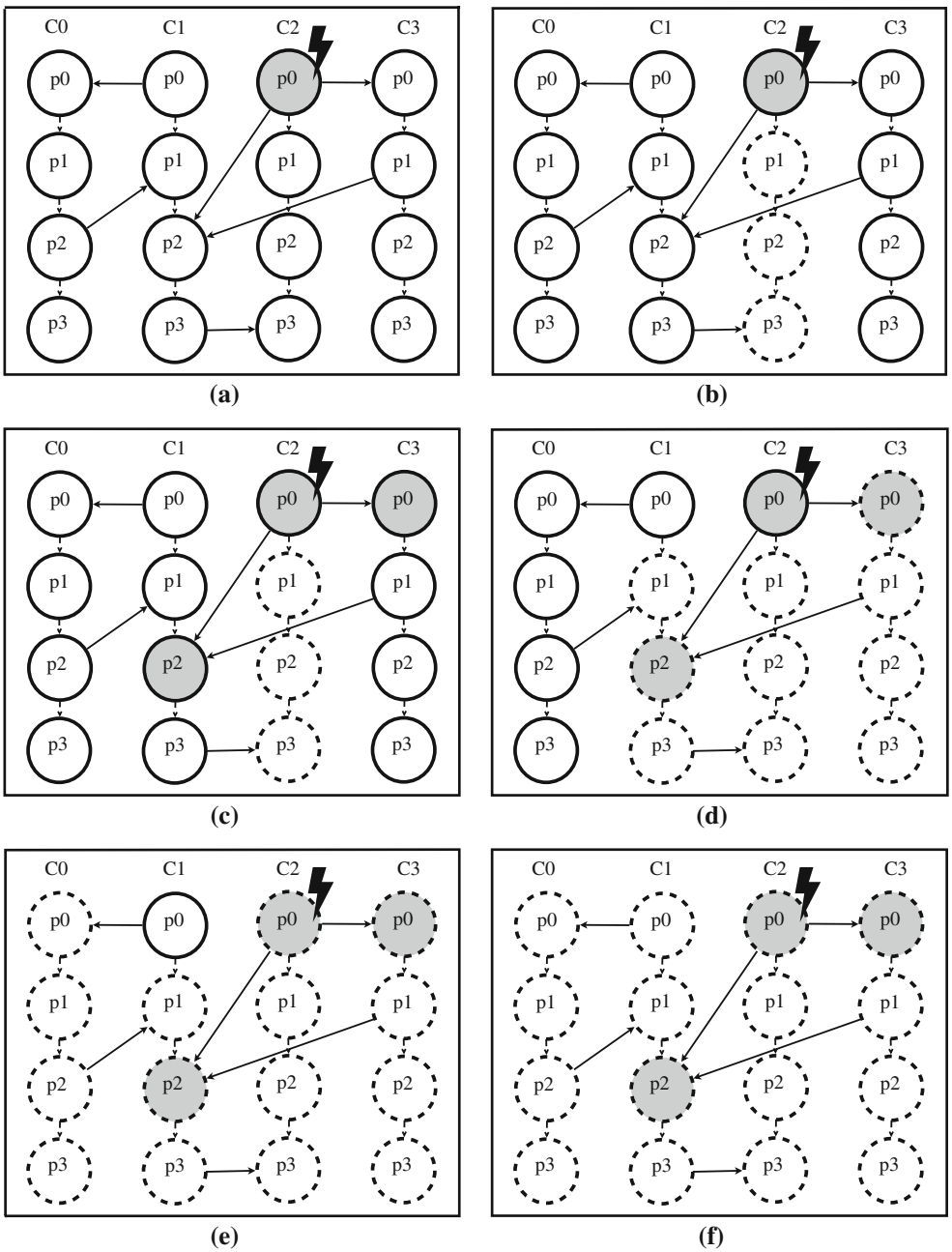

Fig. 4 Fault recovery mechanism. In this example, four cores C0, C1, C2 and C3 have executed four p-XACTs p0, p1, p2 and p3. In the figures, the data sharing is represented by a *solid line* and an *arrow*, while the order of precedence is indicated by a *dotted line*. In (**a**), a fault is detected in p0 from C2, which initiates the recovery mechanism. In (**b**) the recovery mechanism proceeds with p3 from C2, the youngest p-XACT of the core. As p3 has no consumers, it is locally recovered. C2 repeats the same process with p2 and p1, which are also locally recovered. As p0 in C2 is producer, it cannot be recovered yet. In (**c**), C2 sends invalidations to its consumers C1 and C3 and waits for the corresponding ACKs. In (**d**), C1 performs the recovery for p3. As p3 is producer of C2, it sends a rollback request which is acknowledged by C2 since p3 has already been recovered. Then, p3 and p2 from C1 are rolled-back. Given the fact that p2 recovery was requested by C2, C1 informs it that the recovery for the affected p-XACT has been performed. In the same way, C3 recovers from p3, p2, p1 and p0. As p0 is the oldest p-XACT, the registered checkpoint is also recovered and finally, C3 acks C2. In (**e**), C2 has received all the ACKs, thus performing the rollback of p0 and restoring the backup of the register file. Meanwhile, C1 sends an abort request to C0 since p0 is its producer. C0 recovers from p3, p2, p1, p0, restores the register file checkpoint and acknowledges C1. Finally, in (**f**), C1 receives the acknowledgement from C0 and performs the rollback from p0 and recovers the register file backup

is crucial for its performance. Nonetheless, this measure is not desirable given the fact that it reduces the flexibility and adds a considerable hardware overhead.

Instead of that, our approach uses the interconnection network to communicate redundant cores. The major implication of this approach, thus, is the increase of the latency

when accessing the log. This additional delay results from the travel across the chip of log blocks from master to slave cache. Eventually, if the latency is too high, the performance of the slave thread may be affected considerably. Nonetheless, this decreased slave thread performance slightly impacts on the master thread forward progress because of the temporal decoupling inherent to LBRA, which allows the master thread to commit several p-XACTs without the need of slave consolidations. If this buffering results insufficient, we can still adopt measures like allowing a higher number of in-flight p-XACTs, something which requires more hardware, or increasing the p-XACT size, which impacts directly on the total log size and reduces the effective capacity of the cache.

6 Evaluation

6.1 Simulation environment

To evaluate the proposed LBRA architecture, we have simulated a tiled-CMP by means of Virtutech Simics [19] and GEMS [20]. Simics is a functional simulator executing a Solaris 10 Unix distribution simulating the UltraSPARC-III ISA. GEMS is a timing simulator which, coupled to Simics, supplies a hardware implementation of a transactional memory model called LogTM-SE [38]. We have performed several modifications to the simulator to provide the redundant execution of software threads, as well as other modifications related to the way in which the log is accessed. Furthermore, we have implemented all the hardware additions as described in Sect. 4, together with all the mechanisms needed for the detection of transient faults.

Table 3 shows the main parameters of the evaluated architecture. Each core of our 16-core CMP is a dual-threaded SMT with private L1 cache and a shared portion of the L2 cache. We conduct our experiments by executing several applications from SPLASH-2 [36] (barnes, fft, radix, raytrace, waternsq and watersp), ALPBench [18] (facerec, mpgdec and mpgenc) and PARSECv2.1 [4] (blackscholes, canneal and swaptions) benchmark suites. The experimental results reported here correspond to the parallel phase of each program. Each experiment has been run with several random seeds as to take into account the variability of the multi-threaded execution. Although LBRA allows the programmer to explicitly activate the redundancy in specific program parts, in this work we assume full protection. Thus, every program instruction is redundantly executed.

Table 3 Simulation parameters

16-way Tiled-CMP	
Processor speed	2 GHz
Memory and cache	
Mem. size	4 GB
Mem. latency	300 cycles
Cache line size	64 bytes
L1 cache	32KB, 1 cycle/hit
L2 cache	512KB/core, 15 cycles/hit
Network	
Topology	2D-Mesh
Protocol	MESI directory
Link latency	4 cycles
Flit size	4 bytes
Link bandwidth	1 fit/cycle
LogTM-SE	
R/W signatures	Variable size (see Sect. 6.4)
Verification latency	10 cycles
Log contents	Loads: data (4 bytes)
	Stores: data + address (8 bytes)

6.2 Area overhead

LBRA support increases the hardware overhead with respect to a common LogTM-SE implementation, as we have seen in Sect. 4. Nonetheless, this hardware addition is moderated. As depicted in Table 4 the largest structures are the register file checkpoint, which is used to rollback to a safe state in case of a fault, and the Verification Signature, which is used to compare the state of redundant threads. The impact of the rest of structures is negligible. It is also worth noting that for every in-flight p-XACT, new hardware structures are needed.

6.3 p-XACT size analysis

The size of a p-XACT is a key parameter in the architecture. A bigger size helps to increase the decoupling between master and slave threads. Unfortunately, this also increases the size of the log, incurring in a greater occupancy of the cache. Figure 5a shows a sensitivity analysis of the p-XACT base size in terms of memory instructions. The bars are normalized with respect to the case in which p-XACT length is 25 memory instructions. As we can see, decreasing p-XACT size from 1,000 instructions to 100 instructions brings performance gains. However, further decreasing the p-XACT size below 50 instructions it is not worthwhile. On average, 50-instruction size performs 3, 2 and 3 % better than 25 instruction size for SPLASH-2, PARSEC and ALP studied benchmarks, respectively. For smaller sizes (under 25 instructions), performance decreases because it is not possible to pay off the cost of the p-XACT creation.

Another interesting parameter is the maximum number of p-XACTs which the master can commit without consolidation. At one end, a higher number of in-flight p-XACTs facilitates decoupling, but it also adds more hardware as seen

Table 4 Storage overhead in LBRA

Structure	Checkpoint registers	Consolidated IDs	Log pointers	Verification signature	Producer register	Consumer register
Size (bytes)	4	8	10	128	8	8

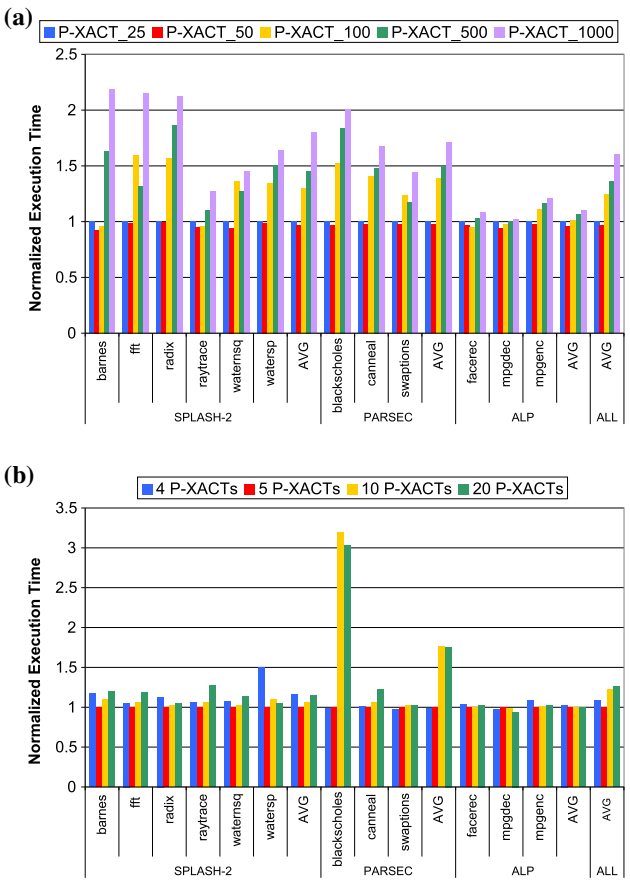

Fig. 5 Sensitivity analysis for p-XACT size and number of in-flight p-XACTs

in Fig. 2. In addition, the size of the log grows, increasing thus the cache miss ratio of the architecture. At the other end, if the number of in-flight p-XACTs is low, in situations in which the slave thread is unable to keep up with the master (e.g. because of dependencies in consolidations), this turns in a bottleneck since the master must be stalled. This behaviour can be observed in Fig. 5b. For 4 in-flight p-XACTs, the stalls of the master execution are responsible for a performance degradation of 16 % in SPLASH-2 and 2 % for ALP (almost no degradation for PARSEC benchmarks) in relation to 5 in-flight p-XACTs, which is the best configuration for the studied benchmarks. For a higher number of in-flight p-XACTs, the overhead specially increases in benchmarks such as blackscholes and canneal, in which the cache miss ratio raises significantly.

6.4 Signature size analysis

LBRA makes use of R/W signatures to keep track of the interactions among p-XACTs. To compute these signatures we implement the double-bit-select (DBS) algorithm as previously proposed in LogTM-SE [38]. A smaller signature size reduces the hardware overhead whereas, in general, the number of false positives increases, something which may lead to performance drawbacks. Specifically, false positives lead to an increase of premature commits due to producer/consumer constraints as explained in Sect. 4.3.1. Therefore the number of used p-XACTs increases and may lead to stalls due to lack of resources.

In Fig. 6 we can see the performance of different signatures sizes (from 64 bits to 2,048 bits) compared to the use of perfect signatures. As depicted, small signature sizes perform comparably to perfect signatures for most of the benchmarks. However, in benchmarks such as watersp and facerec the use of small signatures impacts dramatically on performance due to the large number of false positives. As we can see, 64 bit signatures, which lead to an average of 50 % of false positives, incur in a 11 % of performance degradation. On the other hand, 2,048 bit signatures obtain virtually no performance degradation. While the use of perfect signatures is not feasible in real hardware, using 1,024–2,048 bit signatures provides a good tradeoff between area overhead and performance.

6.5 Overhead of the fault-free case

In this section we compare our proposed architecture to a base case composed by a 16-core CMP running the 16-threaded

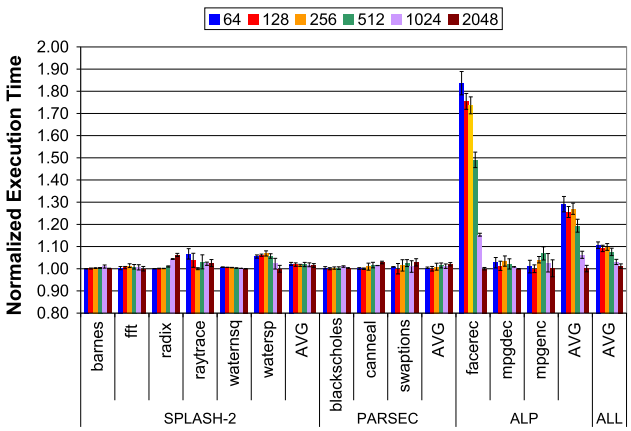

Fig. 6 Impact of signature size on LBRA performance

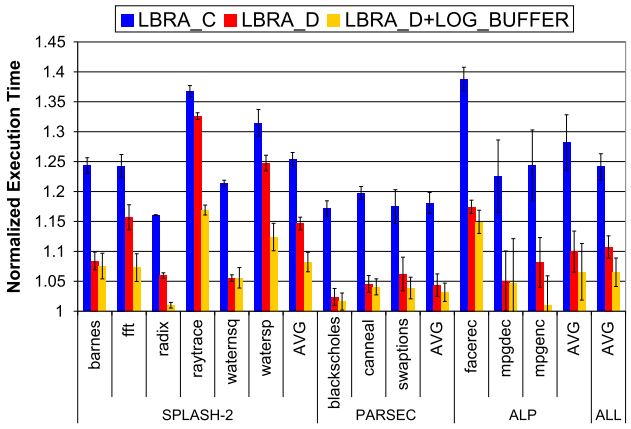

Fig. 7 LBRA performance in a fault-free scenario

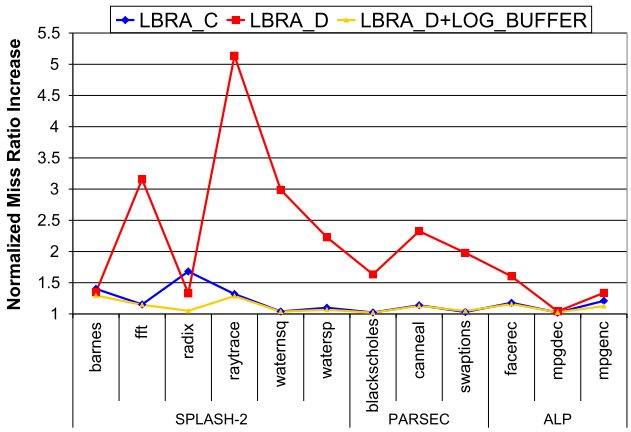

Fig. 8 LBRA L1 miss rate increase

applications mentioned in Sect. 6.1. We quantify the performance in a fault-free scenario which can be considered the common case. Although LBRA allows the programmer to select specific program areas to protect while leaving the rest unprotected, for the evaluation, we provide redundant execution for all of the program instructions.

Three different factors are responsible for the performance overhead of LBRA. First and foremost, the cost of redundancy itself (note that the use of dual SMT cores aggravates this overhead as a result of the higher resource contention of master-slave pair threads). Second, the capacity of the L1 cache, which is reduced by the log used to bypass data between master and slave thread and to provide a backup. This way, smaller p-XACTs normally achieve better performance. And finally, the stalls in the consolidation phase due to dependencies among two or more p-XACTs. Fortunately, these consolidation stalls are uncommon (virtually non existent). Furthermore, the master thread is rarely stalled as a result of the proposed mechanism which allows to execute several p-XACTs without consolidation.

Figure 7 depicts the behaviour of the coupled LBRA approach labeled as LBRA_C. As we can see, the performance degradation in our first approach ranges between 38 % (facerec) and 16 % (radix) with an average of 24 %. Although the experimented degradation is noticeable across all the studied benchmarks, it is worth noting the impact over ALP benchmarks because of the inherent SMT degradation. These results are consistent with related studies such as Sasanka et al. [27], where CMP architectures obtain a better performance when compared to SMT architectures for multimedia workloads.

To overcome this issue we propose to enhance LBRA by using decoupled redundant threads in different non-SMT cores, as explained in Sect. 5. Thus, we avoid degradation due to resource sharing but we increase the hardware requirements. Specifically, the number of cores increases to 32 distributed in a 4×8 2D-mesh in which master and slave cores are placed at a 1 link distance. LBRA_D in Fig. 7 shows the overhead of this approach which ranges from 32 (raytrace) to 2 % (blackscholes) with an average of 11 %, clearly outperforming LBRA_C (24 % on average). But, besides the hardware overhead, LBRA_D results in a noticeable increase of the L1 cache miss ratio as we can see in Fig. 8. However, the size of the log remains constant with an average of 1 KB (up to 2.4 KB) for all the studied benchmarks. Thus, it is clear that the miss ratio increase is due to the conflicts between master and slave cores when accessing the log.

6.6 Leveraging coherence actions

In a coupled environment the log is allocated in the L1 cache and, virtually, all its accesses result in hits for both master and slave threads due to temporal locality. In a decoupled environment, though, the log blocks are written by the master in its private cache and requested by the slave thread. This implies a cache-to-cache request and a transfer of the block permissions from *M* to *S*.[3] This extra latency only affects slave performance as we noted before. But the major problem results when the master thread eventually reuses a portion of the log. Since the last state of the log block is *S*, the master thread must re-acquire the write permissions, something which implies an invalidation message to the sharers (the slave thread in this case) and a subsequent acknowledgement. This results in an increase of log latency in the master which may affect the forward progress of the application.

In order to avoid this issue and given the singularity of this producer-consumer pattern, we have added a small hardware structure on the slave side together with the use of non-coherent requests to access the log. We call this structure the *log buffer*, a FIFO queue which stores log blocks. Slave accesses are performed through it. When the requested data

[3] Provided that the cache coherence protocol is MESI.

is not present in the log buffer, the slave thread performs a cache-to-cache request which does not alter the coherence state of the memory subsystem. The capacity of the log buffer can be as small as one entry. Nonetheless, our experimental analysis shows that with a capacity of three blocks, we obtain the optimum performance by using a simple prefetch mechanism. The prefetch strategy we follow consists of requesting the next logical log block whenever there is, at least, one free entry in the buffer. This approach is labeled in Fig. 7 as LBRA_D+LOG_BUFFER. In this case, the time overhead is reduced to the range between 17 (raytrace) and 1 % (radix) with an average of 6.5 % for all the studied benchmarks. The performance improvement in comparison to LBRA_D is easily explained due to the reduction in the cache misses, as we can see in Fig. 8. In conclusion, the benefit of LBRA_D+LOG_BUFFER is twofold. First, we avoid performance degradation thanks to the redundant threads in the same SMT core. And second, we decrease the impact over the miss ratio by leveraging the coherence for log blocks.

6.7 Comparison against previous work

Now, let us evaluate the performance impact of previous approaches in a common framework, i.e. a) a direct-network environment and, b) using 2-way SMT core redundancy or simply core redundancy. LBRA_C, LBRA_D and LBRA_D+LOG_BUFFER are the presented mechanisms in this paper. Finally, REPAS [25] executes redundant threads in SMT cores whereas DCC [16] use redundant cores which communicate through the network to detect faults.

While LBRA_C uses the cache to communicate log values between redundant threads, REPAS uses specialized hardware, something which imposes a considerable complexity overhead. Besides, as we can see in Fig. 9, LBRA_C reduces the execution time overhead of REPAS for the majority of studied benchmarks with a 8 % on average with respect to the base case. For applications that lay more pressure over the LVQ and SVQ queues such as watersp and swaptions, REPAS incurs in noticeable overheads, since the leading thread must be stalled until resources are available. LBRA_C does not suffer from this problem since all the data communication is performed through the cache.

In the same Fig. 9, we can see the execution time overhead of DCC and LBRA_D implementing the log buffer. In both cases, redundant threads are executed in different cores, eliminating thus the degradation inherent to the SMT execution. The counterpart, however, is that the number of cores is increased 2× in both approaches. The consistency mechanism is the major degradation source in DCC, as we explained in Sect. 2.1.2, increasing time overhead in 31 % on average with respect to the base case.

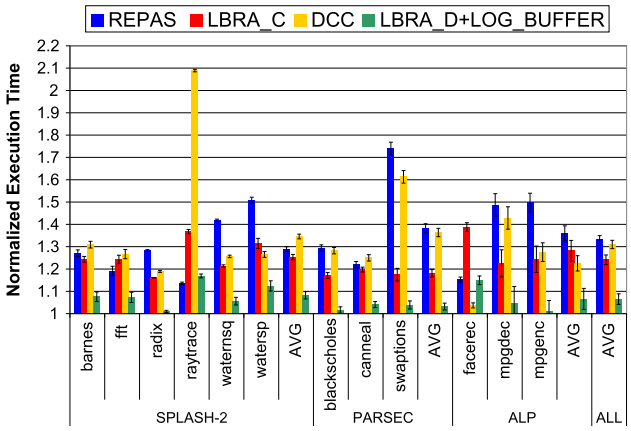

Fig. 9 Performance comparison of LBRA versus REPAS and DCC

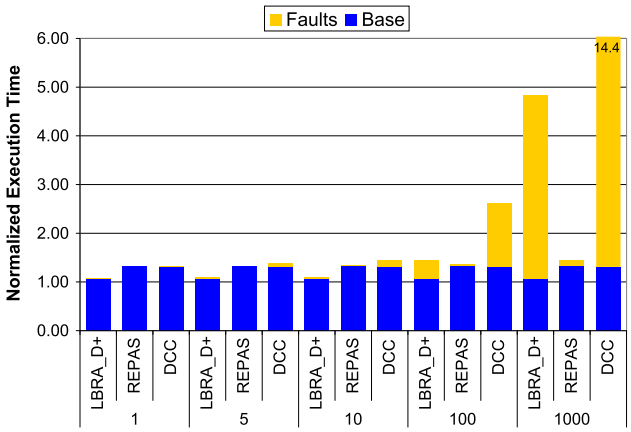

Fig. 10 Execution time overhead for several fault rates

Finally, in Fig. 10 we can see the execution time overhead for LBRA_D (with log buffer), REPAS and DCC in a faulty environment. Rather than injecting faults all across different areas of the processor (which would require extensive simulation), we simply mark faulty transactions based on a random probability. The evaluated fault rates are expressed in number of faults per million of execution cycles. Since realistic rates barely affect the performance of mechanisms, in this experiment we have used rates which are higher than those expected in a real scenario so to characterize the behavior of the different approaches. The time to recover from a fault depends on the speed to detect the fault and to undo all potentially affected work. REPAS is the fastest mechanism to detect and correct an error because of the small delay between redundant threads (less than 200 cycles). On the contrary, DCC spends roughly 10,000 cycles to recover because of its long checkpoint intervals. The overhead of LBRA_D stays in the middle of the other two. As depicted in Fig. 10, while the fault rate is less than 10 faults per million of cycles, LBRA_D is still the best approach but, when the fault rate increases to 100, the small overhead introduced

by REPAS makes it the best choice. DCC's performance is always the worst in a faulty environment because of its long checkpoint intervals.

To sum up, we have shown that both LBRA_C and LBRA_D are able to outperform previous approaches in the same environment with several advantages. First, we avoid the use of queues to bypass data values between threads as in REPAS, something which augments considerably the complexity of the design. Second, the modifications over the memory system are minimum whereas DCC induces virtually to the creation of a specific and new coherence protocol [16] and requires special measures to deal with master-slave inconsistencies. In the same way, in a faulty environment with realistic fault rates, LBRA_D is also the best approach in terms of performance degradation.

7 Conclusions and future work

CMOS scaling exacerbates hardware errors making reliability a big concern for present and future microarchitecture designs. However, mechanisms to provide fault tolerance in architectures must accomplish several objectives such as low performance degradation, power consumption and area overhead.

In this work, we present a novel low-overhead mechanism to deal with transient faults in present and future architectures. To this end, we introduce LBRA, an architecture design based upon LogTM-SE, a well-established hardware implementation of Transactional Memory. LBRA executes redundant threads which communicate through a virtual memory log placed in cache. The goal of the log is twofold. First, it provides input replication for both threads and, second, it is used to recover the architectural state of the system after a fault is detected.

LBRA main features include a) a consistent view of the memory between master and slave thread, avoiding input incoherences; b) both transient fault detection and recovery; c) more scalability and higher decoupling than previous proposals; d) low-performance overhead.

LBRA is presented in two flavours. In the first approach, redundant threads are executed in the same dual-threaded SMT core. This provides a low-hardware overhead but imposes a noticeable performance degradation as a counterpart. To solve this issue, we have proposed a second approach in which redundant threads are executed in different cores, increasing, thus, hardware requirements. To address the inter-core communication latency, we rely on the use of a simple yet effective mechanism comprised by a log buffer, a prefetch strategy and slight modifications of specific coherence actions.

We have compared and evaluated the proposed designs using full system simulation to measure the performance degradation in a fault-free environment with parallel benchmarks. We have shown that our proposals address the fault-tolerant goal imposing 20 and 7 % execution time overhead for the coupled and decoupled mechanism, respectively. Furthermore, we have shown that LBRA presents several advantages with respect to state-of-the-art approaches that we have also evaluated in the same framework.

As future work, we plan to study the behaviour of lazy policies in a similar approach to Reunion. In that case, memory updates are not effective until blocks are verified. The major advantage is that slave threads can access memory independently. However, the major drawback to confront is that external writes between the same dynamic memory operation could create input incoherences. This would only be detected at consolidation time when all the executed work have been discarded.

Acknowledgments Thanks to the anonymous reviewers for their comments and suggestions which definitely improved this work. This work was jointly supported by the Spanish MINECO and Spanish MEC, as well as European Commission FEDER funds under grant numbers TIN2012-38341-C04-03 and TIN2012-31345.

References

1. Agarwal, R., Garg, P., Torrellas, J.: Rebound: scalable checkpointing for coherent shared memory. In: Proceeding of the 38th Annual International Symposium on Computer Architecture, ISCA '11, pp. 153–164. ACM, New York (2011)
2. Bartlett, J., Gray, J., Horst, B.: Fault tolerance in tandem computer systems. In: The Evolution of Fault-Tolerant Systems (1987)
3. Bernick, D., Bruckert, B., Vigna, P. D., Garcia, D., Jardine, R., Klecka, J., Smullen, J.: Nonstop advanced architecture. In: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05), pp. 12–21. Yokohama, Japan (2005)
4. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 72–81 (2008)
5. Borkar, S.: Design challenges of technology scaling. IEEE Micro **19**(4), 23–29 (1999)
6. Borkar, S., Karnik, T., Narendra, S., Tschanz, J., Keshavarzi, A., De, V.: Parameter variations and impact on circuits and microarchitecture. In: DAC '03: Proceedings of the 40th Annual Design Automation Conference, pp. 338–342, ACM, New York (2003)
7. Bowman, K., Tschanz, J., Wilkerson, C., Lu, S.-L., Karnik, T., De, V., Borkar, S.: Circuit techniques for dynamic variation tolerance. In: DAC '09: Proceedings of the 46th Annual Design Automation Conference, pp. 4–7. ACM, NY (2009)
8. Censier, L.M., Feautrier, P.: Readings in computer architecture. chapter a new solution to coherence problems in multicache systems, pp. 576–582. Morgan Kaufmann Publishers Inc., San Francisco (2000)
9. Ceze, L., Tuck, J., Montesinos, P., Torrellas, J.: Bulksc: bulk enforcement of sequential consistency. In: Proceedings of the 34th International Symposium on Computer Architecture, pp. 278–289 (2007)
10. Culler, D., Singh, J.P., Gupta, A.: Parallel Computer Architecture: A Hardware/Software Approach (1998)

11. Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. In: Proceedings of the 38th Annual International Symposium on Computer Architecture, pp. 365–376 (2011)
12. Frank, D.J.: Power-constrained CMOS scaling limits. IBM J. Res. Dev. **46**(2/3), 235–244 (2002)
13. Gomaa, M., Scarbrough, C., Vijaykumar, T.N., Pomeranz, I.: Transient-fault recovery for chip multiprocessors. In: Proceedings of the 30th International Symposium on Computer Architecture, pp. 98–109. San Diego, California (2003)
14. Kim, T.-H., Liu, J., Keane, J., Kim, C.: A 0.2 v, 480 kb subthreshold sram with 1 k cells per bitline for ultra-low-voltage computing. IEEE J. Solid-State Circuits **43**(2), 518–529 (2008)
15. Kumar, R., Zyuban, V., Tullsen, D.M.: Interconnections in multicore architectures: understanding mechanisms, overheads and scaling. In: Proceedings of the 32th International Symposium on Computer Architecture (ISCA'05), pp. 408–419. Madison, Wisconsin (2005)
16. LaFrieda, C., Ipek, E., Martinez, J.F., Manohar, R.: Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In: Proceedings of the 37th International Conference on Dependable Systems and Networks, pp. 317–326. Edinburgh, UK (2007)
17. Li, M.-L., Ramachandran, P., Sahoo, S., Adve, S., Adve, V., Zhou, Y.: Understanding the propagation of hard errors to software and implications for resilient system design. In: Proceedings of the 13th International Conference on Architectural Support for Programming Language and Operating Systems, pp. 265–276. Seattle, WA, USA (2008)
18. Li, M.-L., Sasanka, R., Adve, S.V., Kuang Chen, Y., Debes, E.: The alpbench benchmark suite for complex multimedia applications. In: Proceedings of the IEEE International Symposium on Workload Characterization, pp. 34–45 (2005)
19. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B., Werner, B.: Simics: a full system simulation platform. Computer **35**(2), 50–58 (2002)
20. Martin, M.M.K., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's general execution-driven multiprocessor simulator (gems) toolset. SIGARCH Comput. Archit. News **33**(4), 92–99 (2005)
21. Mukherjee, S., Kontz, M., Reinhardt, S.K.: Detailed design and evaluation of redundant multithreading alternatives. In: Proceedings of the 29th International Symposium on Computer Architecture, pp. 99–110. Anchorage, Alaska, USA (2002)
22. Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K., Chang, K.: The case for a single-chip multiprocessor. SIGPLAN Not. **31**, 2–11 (1996)
23. Rashid, M., Huang, M.: Supporting highly-decoupled thread-level redundancy for parallel programs. In: Proceedings of the 14th International Symposium on High Performance Computer Architecture, pp. 393–404. Salt Lake City, USA (2008)
24. Reinhardt, S.K., Mukherjee, S.: Transient fault detection via simultaneous multithreading. In: Proceedings of the 27th International Symposium on Computer Architecture, pp. 25–36. Vancouver, British Columbia, Canada (2000)
25. Sánchez, D., Aragón, J.L., García, J.M.: Repas: reliable execution for parallel applications in tiled-cmps. In: 15th International European Conference on Parallel and Distributed Computing (Euro-Par 2009), pp. 321–333 (2009)
26. Sánchez, D., Aragón, J.L., García, J.M.: A log-based redundant architecture for reliable parallel computation. In: 17th International Conference on High Performance Computing (HiPC), pp. 1–10. Goa (India) (2010)
27. Sasanka, R., Adve, S.V., Chen, Y.-K., Debes, E.: The energy efficiency of cmp vs. smt for multimedia workloads. In: Proceedings of the 18th Annual International Conference on Supercomputing, ICS '04, pp. 196–206. ACM, NY (2004)
28. Smolens, J.C., Gold, B.T., Falsafi, B., Hoe, J.C.: Reunion: complexity-effective multicore redundancy. In: Proceedings of the 39th International Symposium on Microarchitecture, pp. 223–234. Orlando, Florida, USA (2006)
29. Smolens, J.C., Gold, B.T., Kim, J., Falsafi, B., Hoe, J.C., Nowatzyk, A.G.: Fingerprinting: bounding soft-error-detection latency and bandwidth. IEEE Micro. **24**(6), 22–29 (2004)
30. Spainhower, L., Gregg, T.A.: Ibm s/390 parallel enterprise server g5 fault tolerance: a historical perspective. IBM J. Res. Dev. **43**, 863–873 (1999)
31. Taur, Y.: CMOS design near to the limit of scaling. IBM J. Res. Dev. **46**(2/3), 213–222 (2002)
32. Taylor, M.B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffman, H., Johnson, P., Lee, J.-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., Agarwal, A.: The raw microprocessor: a computational fabric for software circuits and general-purpose programs. IEEE Micro **22**(2), 25–35 (2002)
33. Vangal, S., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Iyer, P., Singh, A., Jacob, T., Jain, S., Venkataraman, S., Hoskote, Y., Borkar, N.: An 80-tile 1.28tflops network-on-chip in 65nm cmos. In: Solid-State Circuits Conference, 2007, ISSCC 2007. Digest of Technical Papers. IEEE, International, pp. 98–589 (2007)
34. Vijaykumar, T., Pomeranz, I., Cheng, K.: Transient fault recovery using simultaneous multithreading. In: Proceedings of the 29th International Symposium on Computer Architecture, pp. 98–109. Anchorage, Alaska (2002)
35. Wang, N.J., Patel, S.J.: Restore: symptom-based soft error detection in microprocessors. IEEE Trans. Dependable Secur. Comput. **3**(3), 188–201 (2006)
36. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of the 22th International Symposium on Computer Architecture (ISCA'95), pp. 24–36. Santa Margherita Ligure, Italy (1995)
37. Yalcin, G., Unsal, O., Hur, I., Cristal, A., Valero, M.: Faultm: fault-tolerance using hardware transactional memory. In: The 3nd Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA 2010), pp. 34–47 (2010)
38. Yen, L., Bobba, J., Marty, M. R. Moore, K.E., Volos, H., Hill, M.D., Swift, M.M., Wood, D.A.: Logtm-se: decoupling hardware transactional memory from caches. In: Proceedings of the 19th International Symposium on High-Performance Computer Architecture, pp. 261–272 (2007)