A Log-Based Redundant Architecture for Reliable Parallel Computation

Daniel Sánchez, Juan L. Aragón and José M. García
Departamento de Ingeniería y Tecnología de Computadores
Universidad de Murcia, Spain
Email: {dsanchez, jlaragon, jmgarcia}@ditec.um.es

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 been already proposed to provide 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 main design goal is to provide transient fault detection and recovery while modifying hardware as less as possible.

To this end, we propose LBRA based on a Hardware Transactional Memory (HTM) architecture in which two redundant threads successfully detects and recovers from transient faults, assuring a consistent view of the memory by means of a pair-shared cacheable virtual memory log which keeps the computation results. Results show that our log-based mechanism introduces a small performance degradation of 5% in a non-faulty scenario. Additionally, we show that LBRA supports huge fault rates such as 100 faults per million of cycles with low additional performance degradation.

1. Introduction

Present and future CPUs designs still continue adapting themselves to Moore's law. The increasing transistor density allows designers to integrate a higher number of cores into the same chip, aiming at increasing the performance in an energy-efficient way, while keeping a manageable complexity to exploit Thread Level Parallelism. However, this trend, along with temperature fluctuations and process variation, increases the susceptibility of architectures to hardware errors. Furthermore, proposed techniques to decrease power consumption, such as DVFS, lower voltage margins exacerbating thus reliability problems.

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, which is nowadays applied in large CAM arrays, such as caches or RAM. However, ECC cannot be extensively used through all hardware structures. Instead, in shared-memory environments, dual or triple modular redundancy techniques are used. The main idea behind these approaches is to redundantly execute program instructions and then check the outputs. Among these works we can distinguish two different trends: (1) those in which memory is not updated until the values have been satisfactory checked [10, 4, 13, 19] and (2) those in which, once a fault is detected, the state of the architecture in rolled-back to a previous known-to-be-safe checkpoint [5, 12]. In the first case, performance is affected given the fact that forward progress can be stalled until verification is accomplished. In the second case, the major drawback is the synchronization between redundant executions, which includes stopping execution, sharing and comparing architectural states and, finally, resuming execution.

In contrast to previous approaches on reliable architecture designs, this paper explores the use of already existing log-based hardware transactional memory (HTM) systems as a novel way to provide fault tolerance.

Ideally, a fault tolerant architecture should degrade performance as less as possible and should require an area overhead not larger than 10% [14]. To this end, we present LBRA: A Log-Based Redundant Architecture for Reliable Parallel Computation. What we propose is to build a fault tolerant architecture by using a current state-of-the-art HTM system with slight modifications, something which, to the best of our knowledge, has not been previously studied. The main idea is to execute redundant copies of the same software thread in two different hardware contexts which are executed within the same SMT core. In this work we have chosen LogTM-SE [22], an elegant HTM design which performs both *eager version management*, by updating memory in place and keeping the old values in a virtual memory space called log, and *eager conflict detection*. In particular, we use the capabilities provided by LogTM-SE to assure:

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.

In our approach, the program instructions are executed in virtual execution groups that we call *pseudo-transactions* (p-XACTs) or chunks [3]. The master thread executes p-XACTs as regular instructions but, additionally, it keeps the results of its progress in a pair-shared log. By using 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. This verification is accomplished off the critical path by the slave thread. This high decoupling also allows to disable the redundant mode whenever it is required bringing opportunities such as partial coverage in applications which are not so sensitive to faults, such as multimedia. Experimental results show that our log mechanism introduces a small performance degradation of 5%. In addition, we show that LBRA supports huge fault rates such as 100 faults per million of cycles, with an additional performance degradation of 35% over a non-faulty environment.

The remainder of the paper is organized as follows: Section 2 introduces Transactional Memory and how it can be adapted to provide reliable computation. Section 3 details how our proposal is implemented. The evaluation setup and analysis are described in Section 4. Section 5 discusses the related work. Finally, Section 6 resumes the main conclusions of this work.

2. Reliable Computation by means of Transactional Memory

Our approach is built upon the top of a LogTM-SE [22] system, a hardware implementation of Transactional Memory. This section describes how we provide fault tolerance to the system by adding several modifications to its behaviour with a modest hardware overhead.

2.1. Version management

LogTM-SE provides an eager version management. This means that the values produced by transactions are stored in cache and are visible to the rest of the system. A lazy version management, on the contrary, does not expose updated values until commit. An eager mechanism performs better than a lazy one in case rollbacks are infrequent. Thus, given the fact that faults are still an uncommon event, the use of an eager version management is justified.

2.1.1. Input replication. Our approach could be classified as Redundant Multi-Threading (RMT). In RMT systems two hardware threads, called master and slave, redundantly execute the program instructions to provide fault tolerance within a SMT processor. Note that, unlike true SMT threads, each redundant thread pair appears to the operating system as a single one.

In RMT systems one of the most important issues is *input replication* which defines how redundant threads or executions observe the same data. Since master and slave thread execution is not lockstepped [1], the execution of redundant memory instructions would probably lead to input incoherences. In order to solve this problem, in our proposal we extend the functionality of the log (a memory space allocated in virtual memory already implemented in LogTM-SE) as follows.

For each load instruction, the master appends to the log both the virtual address and loaded value for that address. This way, slave memory instructions are served through the log where they obtain the same values as its master-pair, avoiding then input incoherences. Note that, as the log is written at instruction commit, it will only keep instructions of the right path and in program order.

2.1.2. Output comparison. The *output comparison* defines how the correctness of the computation is assured in RMT systems. In our approach, we define the output comparison granularity at p-XACTs level. 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.

The semantic and execution of a p-XACT is quite different from a regular XACT in LogTM-SE. Firstly, whereas traditional XACTs are manually coded in the application, p-XACTs are dynamically created in execution time and their length is variable, as we will see later. This allows a great flexibility, making redundancy easy to turn on and off. Secondly, a p-XACT does not ensure isolation and/or atomicity. This way, dirty memory blocks are shared as in a non-transactional environment, relying on other synchronization mechanisms such as locks or barriers to assure correction.

The execution of a program in our approach is as follows. First, the master starts the execution of a new p-XACT. This implies the allocation of a new section in the log and the initialization of the registers which hold write and read signatures. Eventually, a mechanism would trigger a signal indicating the end of the current p-XACT. This event is called as the *commit* of the p-XACT. The commit is completely local and it does not require any communication outside the actual core so this mechanism is very fast. However, unlike in the original LogTM-SE implementation, this mechanism does not reset the read and write signatures or the log pointer, something which is carried out by the slave thread. Then, the active transaction is considered as finished and the following program instruction is executed within a new p-XACT.

The task of the slave is to assure the correct execution of all the work done by the master. To accomplish this, the slave thread redundantly executes the p-XACTs committed by the master. At the end of the p-XACT, the slave performs the *consolidation*. In the consolidation process, the architectural state of master and slave threads are compared to assure that the produced values are correct. For that purpose, we follow a similar approach as in [5], where signatures are created and used to compare computations. The master thread creates an in-flight signature which is saved in the *Verification Signature* at commit for every p-XACT, see Figure 1. Then, in consolidation, the slave compares its own signature with the *Verification Signature*. If the check succeeds, the execution of the p-XACT is correct. Then, the signature registers and the log pointer are reset. Finally, the slave performs a backup of its register file which is now considered correct. If a mismatch is found in the consolidation process the recovery mechanism is triggered. Note that faults can be detected before consolidation. This happens when the slave detects a mismatch in the addresses accessed in the log by load or store instructions, as we will see in Section 3.1.3.

2.2. Dependence Tracking

p-XACTs rely on software mechanisms to ensure atomicity and isolation. As blocks are allowed to be shared, potential faults could be spread across the system, so we need to keep track of these sharings. Although conflict detection is not engaged in p-XACTs, we find this mechanism already implemented in LogTM-SE very suitable to keep track of potential faulty shared blocks.

LogTM-SE provides eager conflict detection by means of the coherence protocol, decoupling the mechanism from caches by using write and read signatures. External requests arriving to a core are checked through these signatures and, on a possible conflict, 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 Figure 1. The Producer and Consumer registers keep the transaction identifiers involved in the sharing of all the cores in the system.

The proposed mechanism is 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 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 requestor of the block, when obtaining a response, updates its *Consumer Register* indicating the core and transaction id which produced the obtained block.

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).

3. Implementation Details

One of the major drawbacks in previous RMT approaches is the synchronization between redundant threads. As a measure to amortize the latency of comparing redundant executions, long checkpoint intervals are needed. Our goal is to eliminate these latencies, independently of whether synchronizations are common or not. To achieve this, we use a decoupled approach by means of the capabilities of LogTM-SE and the ability to execute multiple p-XACTs before verification. The hardware additions needed to accomplish these goals are depicted in Figure 1.

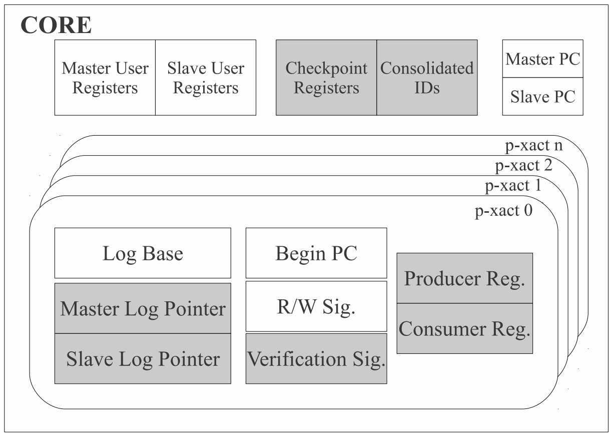

Figure 1. LBRA Hardware Overview. Shadowed boxes represent the added structures.

3.1. Accessing the Log

To provide access to the same log, both master and slave threads should share memory space. To this end, unlike in true SMT threads, master and slave threads appear to the OS as a single one as in [13].

3.1.1. Master access. The master thread writes in the log through the *Master Log Pointer* as in a traditional LogTM-SE system, a pointer which is local to every p-XACT. At every memory operation, the master generates a new store instruction whose destination address is indicated by this pointer. This new store allows the system to satisfy input replication and output comparison as explained in Section 2.1. As memory operations are logged in commit, the content of the log is structured in program order.

3.1.2. Slave access. The slave accesses to the log are more complicated and require a special treatment. The goal is to redirect all the memory references through the log to satisfy input replication and, eventually, detect any arising fault. Therefore, at memory access time the destination address of the load is switched with the *Log Slave Pointer* by means

	Address	Value	Old-value	Facilitates
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

Table 1. Alternatives in log content for loads and stores.

of a multiplexer. Then, the memory access is performed as usual.

The treatment of stores presents two alternatives. In the first case, for each store executed by the slave, a read access to the log is generated through the Slave Pointer. The aim of this access is to assure that the destination address of master and slave stores match. This way, a miscalculation could be detected and the recovery mechanism would be initiated. Despite the log access for every slave store, this approach requires the check of the address by means of a comparator, increasing the pressure in the slave commit phase. Alternatively, we propose a simpler yet not less correct mechanism: avoiding store checking. Then, instead of detecting faults at store granularity, we delay it until consolidation process, in which both registers and execution signatures are checked. This approach is beneficial for two reasons: first, it does not increase cache pressure since no access is performed to the log (the memory access in the slave store is simple discarded), and secondly, it does not require an additional comparison, which does not complicate slave commit phase.

3.1.3. Log content. The size of the log is a major concern in our approach since its growth affects the available cache space for the application. In this section we discuss how to control the size of the log by reducing its contents. The different alternatives can be seen in Table 1.

The contents to write in the log could be adjusted depending on the desired detection mechanism granularity. As a first approach, the data logged by the master thread in every memory operation are the address and value for each operation, as explained in Section 2.1. For loads, the log of address and read value is used to satisfy input replication and to detect faults concerning the calculation of the destination address. A measure to reduce the log size is to avoid logging the address. In this case we rely on the consolidation process to detect any fault regarding a miscalculated destination address.

For stores, it is mandatory to log the address and old value in order to perform the recovery process, as we will see later. Nonetheless, there exist two options with the current value to store. If this value is logged, a fault in the calculation of this value could be detected here. In order to reduce the size of the log we avoid logging the value to be stored. Then, as in the case of loads, we rely on the consolidation process to detect faults derived from a miscalculation of the value of a store instruction.

3.2. Circular Log

In LBRA the execution and verification of p-XACTs is decoupled. For this, the master thread is allowed to execute and commit several p-XACTs while, as a background mechanism, the slave consolidates its correct execution. This way, the forward progress of the master is never interrupted by any verification process.

To allow the buffering of up to n p-XACTs each transaction needs its own hardware as depicted in Figure 1, which is very similar to the hardware added in LogTM-SE. However, one of the major differences between LogTM-SE and our proposal is how the commit and consolidation of a p-XACT handle the log. In our approach, the commit of a p-XACT does not affect the log, in the sense that none of the data stored is eliminated after commit. As the log is used by the slave as a way to verify the results of the master, the contents of the log are not reset until a correct verification is performed in the consolidation process. This way, the log grows circularly through all the virtual space reserved for it.

3.3. In-order Consolidation

In our approach, memory blocks are updated in place (L1 cache) and allowed to be shared even before consolidation takes place. This eager approach allows fast commits and appropriate results, since faults can be considered as the uncommon case. 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, because a faulty block produced in p_j would be silently consolidated in p_i. To take track of these dependencies we introduce the Consumer and Consolidated-Ids registers, as explained in Section 2.2, which gather the information provided by the coherence protocol. To achieve this, memory coherence messages in our approach are extended to include the p-XACT identifier providing the data, which are used by the requestor 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

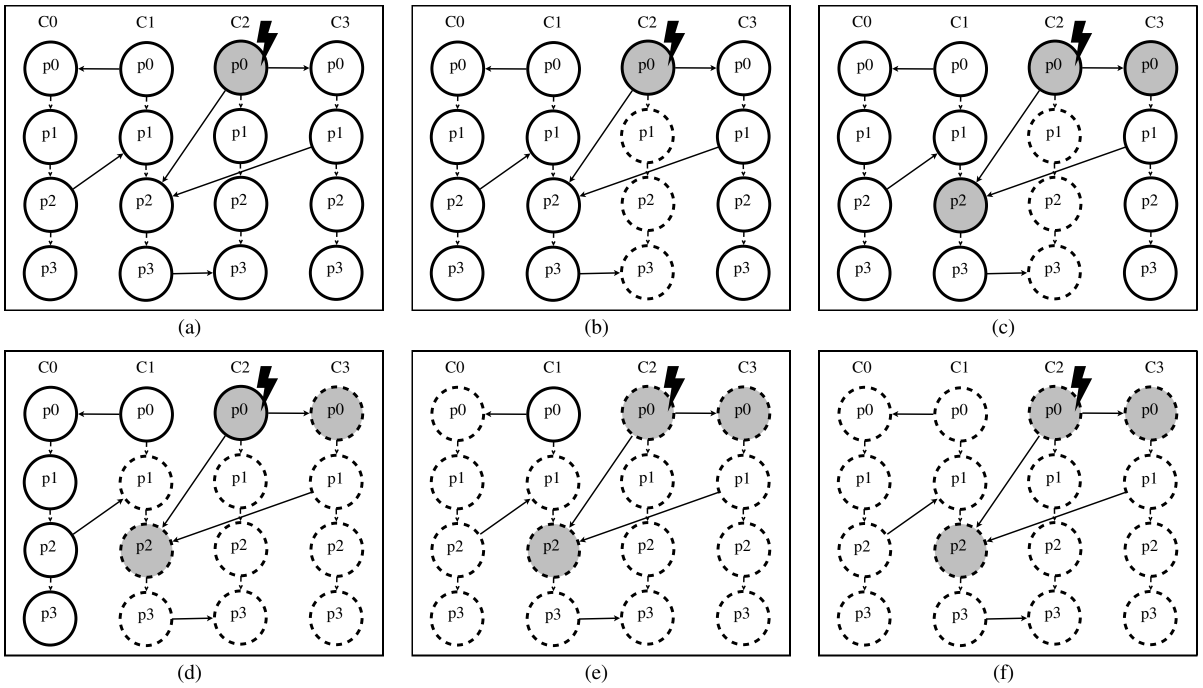

Figure 2. **Fault recovery mechanism.** In this example, four processors C0, C1, C2 and C3 have executed four p-XACTs p0, p1, p2 and p3. In the figures, the data sharing is represented by a solid line and an arrow, while precedence order is indicated by a dotted line. In (a), a fault is detected in p0 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. Since p2 recovery was requested by C2, it acks to inform 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 register checkpoint is also recovered and finally, C3 acks C2. In (e) , C2 have received all the ACKs so it performs the rollback of p0 and restore the registers checkpoint. Meanwhile, C1 sends an abort request to C0 since p0 is its producer. C0 recovers from p3, p2, p1, p0, restores the registers checkpoint and acknowledges C1. Finally, in (f), C1 receives the acknowledgement from C0 and performs the rollback from p0 and recovers the registers checkpoint.

from any other p-XACT, so the consolidation process can take place without any additional checks. In the case 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 lookup mechanism. The slave thread requests its producers to provide the last consolidated id until all the dependencies are satisfied.

3.3.1. 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 provide 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.

3.4. Fault Recovery

Upon 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.

3.4.1. 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 perform transactional aborts. This software writes back the old values to their appropriate address from the log. After that, the transactional hardware of the current p-XACT is reset. Additionally, if this mechanism was triggered by an external request, it will acknowledge the requestor.

3.4.2. 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 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 the final step, the register checkpoint is written back to both master and slave, and the execution is resumed. 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 Figure 2.

4. Evaluation

4.1. Simulation Environment

To evaluate the proposed architecture, we have simulated a tiled-CMP by means of Virtutech Simics [8] and GEMS [9]. 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, provides a hardware implementation of a transactional memory model called LogTM-SE [22]. 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 Section 3, together with all the mechanisms needed to detect and recover from a fault.

16-way Tiled-CMP	
Processor Speed	2GHz
Memory and Cache	
Mem. Size	4GB
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	
Signatures	Perfect

Table 2. Simulation parameters.

Table 2 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 [21] (barnes, fft, radix, raytrace, waternsq and watersp), ALPBench [7] (facerec, mpgdec and mpgenc) and PARSECv2.1 [2] (blackscholes, canneal and swaptions) benchmark suites. The experimental results reported here correspond to the parallel phase of each program. Each experiment has been performed with several random seeds in order to take into account the variability of the multithreaded execution.

4.2. p-XACT Size Analysis

The size of a p-XACT is a key parameter of 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

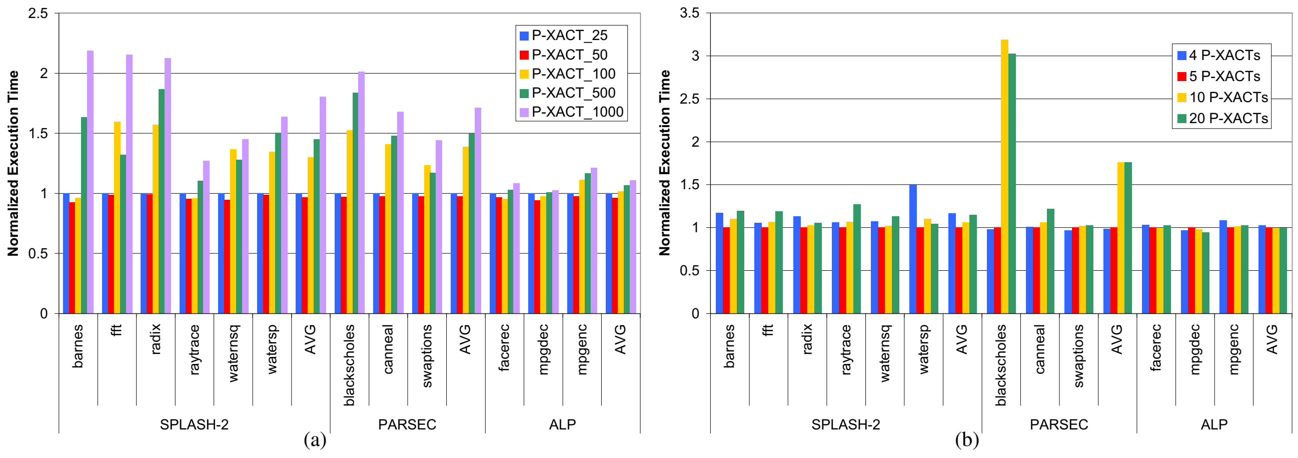

Figure 3. Sensitivity analysis for p-XACT size and number of in-flight p-XACTs.

cache. Figure 3(a) 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 1000 instructions to 100 instructions achieves 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 p-XACT sizes, the performance is even worse (not shown here for clarity), for the overhead incurred in every p-XACT creation (register initializations mostly) becomes more valuable.

		Miss Increase	Consold. Stalls	Master Stalls
SPLASH-2	**barnes**	7.93%	0.02%	0.03%
	fft	20.22%	0.06%	0.02%
	radix	3.12%	0.01%	0.01%
	raytrace	35.18%	0.03%	0.08%
	waternsq	1.93%	0.00%	0.01%
	watersp	0.85%	0.01%	0.01%
PARSEC	**blackscholes**	6.20%	0.00%	0.03%
	canneal	22.10%	0.07%	0.00%
	swaptions	44.31%	0.74%	0.04%
ALP	**facerec**	5.08%	0.14%	0.00%
	mpgdec	1.67%	9.79%	0.01%
	mpgenc	2.35%	0.00%	0.02%

Table 3. L1-Cache Miss Overhead, Consolidation and Master Stalls for 50-instructions p-XACTs.

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 in Figure 1. 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 pace of the master (because of dependencies in consolidations, for example), this turns in a bottleneck since the master must be stalled. This behaviour can be observed in Figure 3(b). For 4 in-flight p-XACTs, the stalls of the master execution are responsible of 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 as it can be seen in Table 3.

4.3. 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 applications mentioned in Section 4.1. We quantify the performance in a fault-free scenario which can be considered the common case. 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 is limited because of the use of the log 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, as we can see in Table 3. Furthermore, the master thread is rarely stalled as a result of the proposed mechanism which allows to execute several p-XACTs without consolidation.

As shown in Figure 4, the average performance degradation of LBRA is 18% for the studied benchmarks. Note, however, that most of it is due to the cost of the redundancy (the first factor cited above). A way of isolating the real

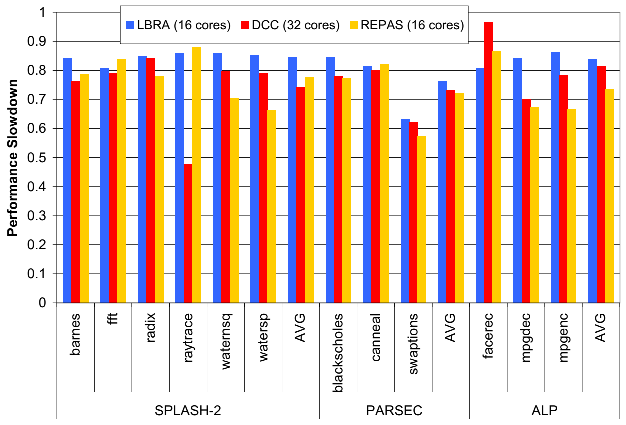

Figure 4. Performance Slowdown Comparison.

overhead introduced by LBRA implies simulating a configuration without resource contention between master-slave threads. By doing this, we have measured that the actual performance overhead of LBRA is just 5% on average.

4.4. Comparison Against Previous Work

DCC [5] is built with a shared-bus as interconnection network. However, the use of shared-buses will no longer be possible in future CMP architectures due to area, scalability and power constraints issues. Therefore, we have studied the behaviour of DCC whenever a direct network such as a 2D-mesh is used.

The performance degradation exhibited by DCC is caused by two main reasons: the consistency window and the checkpoint creation. In contrast to our log mechanism, DCC uses a consistency window to manage input incoherences. When a node performs a load or a store, it opens a read or a write window, respectively, for that address. This window is closed when the redundant thread executes the same operation. To assure coherence between redundant nodes, a sufficient condition is that one node cannot open a write window for an address which is already opened. In a shared-bus environment this mechanism is easily implemented since all the requests are seen for all the nodes. However, in a direct-network scenario, this mechanism should be implemented by broadcast requests, which lead to both performance degradation due to indirection, and network traffic increase. Indirection is also responsible for the increase in the checkpoint creation, since the multi-phase synchronization protocol must be augmented to face the new direct-network scenario [17].

As we can see in Figure 4, LBRA is able to outperform DCC in a 10% on average for SPLASH-2 applications. However, in PARSEC and ALP benchmarks the performance gain is reduced. The reason for this behaviour is that in PARSEC and ALP applications data sharing is reduced to the extent that the degradation resulting from the consistency window affects performance less. In any case, we have to bear in mind that while LBRA makes use of SMT cores to provide fault-tolerance, DCC employs twice the number of cores. This reduces the overall throughput of a system implementing DCC in more than a 100% over a non fault-tolerant base case.

We also compare our proposal to REPAS [18], another RMT-based approach based on SMT-dual execution. REPAS uses two queues to bypass data between master and slave thread, the LVQ for load values and SVQ for store values, providing input replication and output comparison, respectively. In LBRA we avoid the use of these queues allowing a more decoupled environment in which slave threads can run thousands of instructions ahead of the master thread, if necessary. This decoupling also allows to easily turn off the redundant mechanism. As shown in Figure 4, LBRA outperforms REPAS in a 7%, 4% and 10% for SPLASH-2, PARSEC and ALP benchmarks.

4.5. Performance in a Faulty Environment

LBRA introduces an overhead in a fault-free scenario guaranteeing the correct execution of shared memory applications, even in the presence of soft errors. Fault detection and recovery introduce an additional overhead that we study below.

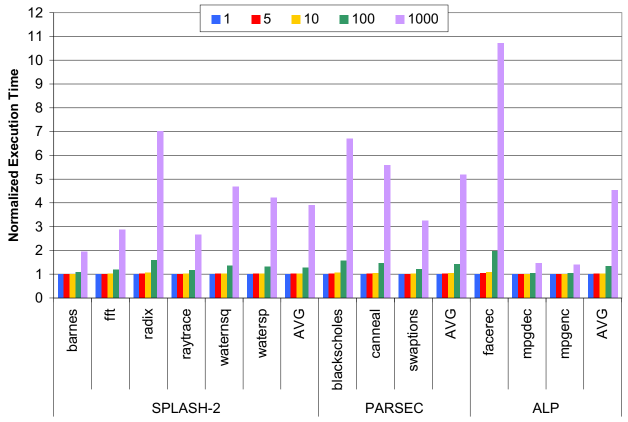

Figure 5. Execution Time Overhead for several fault rates.

Figure 5 shows the execution time overhead of LBRA under different fault rates normalized with respect to a non-faulty environment. Failure rates are expressed in terms of faults per million of cycles. For a realistic fault ratio, the performance of LBRA is barely affected. Therefore, in this experiment, we have used rates which are extremely higher than those expected in a real scenario, so as to show the kindness of the proposed architecture.

As we can see, LBRA is able to tolerate rates of 100 faults per million cycles with an average performance degradation of 29%, 41% and 35% for SPLASH-2, PARSEC and ALP benchmarks, respectively, in comparison to LBRA in a non-faulty environment. When the fault ratio is further increased to the high (and unrealistic) amount of 1000 fault per million cycles, the performance shows a noticeable performance loss. 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 detection of the fault, notification to all the potential consumers of the affected p-XACTs and, finally, the restoration of the log. Applications such as radix, blackscholes and facerec are more affected by faults since, on average, the length of their p-XACTs is greater. As a consequence, the detection of a fault takes place later, and the number of blocks and dependencies with other processors is usually higher.

5. Related Work

There is a large body of literature on mechanisms to deal with the increasing reliability problems in current and future architectures. One of the first techniques that was proposed to mitigate the impact of hardware faults was error coding techniques. Error detection and correction codes are based on the use of extra bits appended to data. In microarchitectures, the most studied mechanisms so far are based on redundant execution in which outputs are compared in order to detect faults. One of the first approaches to full redundant execution is Lockstepping [1], a proposal in which two statically bound execution cores receive the same inputs and execute the same instructions step by step. Later, the family of techniques Simultaneous and Redundantly Threaded processors (SRT) [13], SRTR [19], CRT [10] and CRTR [4] were proposed. A more recent study [18] 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 are directly applied to a multiprocessor domain. In Reunion [15], it is described a mechanism in which redundant threads access memory independently (relaxed input replication) which, unfortunately, leads to divergences in the memory values obtained (input incoherences). These divergences are treated as faults, inducing a serialized execution (very similar to lock-stepped execution) between redundant cores, degrading the performance of the overall architecture. Dynamic Core Coupling (DCC) [5] 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. Input incoherences are avoided by a consistency window which forbids data updates as long as the members of a pair have not observed the same value. However, DCC is built upon a non-scalable shared bus as interconnection network, and whenever a direct network is used as in [17], the performance degradation grows as a result of the consistency window mechanism. In the same fashion, Highly-Decoupled Thread-Level Redundancy (HDTLR) [12] 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 made visible to 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 approach provides a 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.

Similar to our logging mechanism, we find in literature approaches such as SafetyNet [16] and ReVive [11], in which efficient checkpoint mechanisms for recovery are proposed, relying on other mechanisms to accomplish fault detection. Our approach, however, provides an integral solution for both detection and recovery. To maintain checkpoint consistency ReVive uses a global mechanism in which all processors must be synchronized to take a new checkpoint. SafetyNet, however, employs a coordinated local mechanism, in which each processor may create its own checkpoint but, if interactions are found, the processors involved are also will be forced to create their own checkpoint. Our approach falls within this category. With this mechanism we improve the performance of the fault-free execution, avoiding communication latencies while in the critical path (master thread execution). This also creates opportunities such as the concurrent execution of both applications, requiring high reliability, and those, such as multimedia, which not.

Finally, another approach towards fault detection follows a scheme based on symptoms [6] which is inspired on ReStore [20]. 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.

6. 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.

Several studies have been already proposed to provide fault tolerance for parallel codes. However, these proposals have usually been implemented over non-realistic environments, including the use of shared-buses among cores or modifying highly optimized hardware designs such as caches. Proposals like DCC or Reunion use DMR (Dual Modular Redundancy) to provide fault tolerance in microarchitectures. However, they impose a 2X hardware overhead, an unacceptable result for manufacturers which claim for a 10% [14] maximum extra area impact.

Our main design goal is to provide transient fault detection and recovery modifying hardware as less as possible. To this end, we base our proposal upon LogTM-SE, a well-established hardware implementation of Transactional Memory. We propose LBRA, an architecture design in which two redundant threads are able to detect and recover from transient faults, by means of a pair-shared cacheable virtual memory log which keeps the results of the computation. Our evaluation shows that our log mechanism introduces a small performance degradation of 5% in a non-faulty environment, while the overall degradation is mostly due to redundancy itself. We have also shown that LBRA outperforms previous proposals such as DCC in a 14% across all the studied benchmarks.

Besides, we have evaluated the performance of LBRA in a faulty environment, showing an increase of just 35% of execution time with a huge fault ratio of 100 faults per million of cycles. Note that this ratio is much higher than expected in a real scenario, whereas negligible slowdown is reported in a realistic faulty environment.

In the future, we plan to study the performance of LBRA in a scenario in which master and slave threads are executed in different cores. This way, we could improve performance since no degradation would be occasioned because of the redundancy, although the decoupling of the threads should be higher to support the extra latency of the interconnection network. Finally, we plan to extend the functionality of our approach to support Transactional Memory applications.

Acknowledgments

This work was supported by the Spanish MEC and MICINN, as well as European Comission FEDER funds, under Grants CSD2006-00046 and TIN2009-14475-C04. The authors would like to thank the anonymous reviewers for their insightful suggestions and comments. We would also like to thank Rubén Titos for his technical support and Antonio González who provided good suggestions of an earlier version of the manuscript.

References

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

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proc. of the 17th Int' Conf. on Parallel Arch. and Comp. Tech.*, 2008.

[3] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. Bulksc: Bulk enforcement of sequential consistency. In *Proc. of the 34th Int' Symp. on Comp. Arch.*, pages 278–289, 2007.

[4] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proc. of the 30th Int' Symp. on Comp. Arch.*, San Diego, California, USA, 2003.

[5] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proc. of the 37th Int' Conf. on Dep. Sys. and Networks.*, Edinburgh, UK, 2007.

[6] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proc. of the 13th Int' Conf. on Arch. Supp. for Prog. Lang. and Oper. Sys.*, Seattle, WA, USA, March 2008.

[7] M.-L. Li, R. Sasanka, S. V. Adve, Y. kuang Chen, and E. Debes. The alpbench benchmark suite for complex multimedia applications. In *In Proc. of the IEEE Int. Symp. on Workload Characterization*, pages 34–45, 2005.

[8] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2), 2002.

[9] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4), 2005.

[10] S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proc. of the 29th Int' Symp. on Comp. Arch.*, Anchorage, Alaska, USA, 2002.

[11] M. Prvulovic, J. Torrellas, and Z. Zhang. Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proc. of the 29th Int' Symp. on Comp. Arch.*, Anchorage, Alaska, 2002.

[12] M. Rashid and M. Huang. Supporting highly-decoupled thread-level redundancy for parallel programs. In *Proc. of the 14th Int' Symp. on High-Perf. Comp. Arch.*, Salt Lake City, USA, 2008.

[13] S. K. Reinhardt and S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proc. of the 27th Int' Symp. on Comp. Arch.*, Vancouver, British Columbia, Canada, 2000.

[14] S. I. F. Remarks. Selse ii reverie. 2006.

[15] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proc. of the 39th Int' Symp. on Micro.*, Orlando, Florida, USA, 2006.

[16] D. J. Sorin, M. M. Martin, M. D. Hill, and D. A. Wood. Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proc. of the 29th Int' Symp. on Comp. Arch.*, Anchorage, Alaska, 2002.

[17] D. Sánchez, J. L. Aragón, and J. M. García. Evaluating dynamic core coupling in a scalable tiled-cmp architecture. In *Proc. of the 7th Int. Workshop on Duplicating, Deconstructing, and Debunking.*, Beijing, China, 2008.

[18] D. Sánchez, J. L. Aragón, and J. M. García. 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)*, pages 321–333, Delft, Netherlands, August 2009.

[19] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient fault recovery using simultaneous multithreading. In *Proc. of the 29th Int' Symp. on Comp. Arch.*, Anchorage, Alaska, 2002.

[20] N. J. Wang and S. J. Patel. Restore: Symptom-based soft error detection in microprocessors. *IEEE Trans. on Dependable and Secure Comp.*, 3(3), 2006.

[21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the 22th Int' Symp. on Computer Architecture (ISCA'95)*, Santa Margherita Ligure, Italy, 1995.

[22] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *Proc. of the 19th Int' Symp. on High-Perf. Comp. Arch.*, pages 261–272, 2007.