Extending SRT for Parallel Applications in Tiled-CMP Architectures

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

Reliability has become a first-class consideration issue for architects along with performance and energy-efficiency. The increasing scaling technology and subsequent supply voltage reductions are increasing the susceptibility of architectures to soft errors. However, mechanisms to achieve full coverage to errors usually degrade performance in an unacceptable way for the majority of common users.

Simultaneous and Redundantly Threaded (SRT) [13] is a fault tolerant architecture in which pairs of threads in a SMT core redundantly execute the same program instructions. In this paper, we study the under-explored architectural support of SRT to reliably execute shared-memory applications. We show how atomic operations induce a serialization point between master and slave threads. This bottleneck has an impact of 34% in execution speed for several parallel scientific benchmarks. We propose an alternative mechanism in which the L1 cache is updated by master's stores before verification reducing the overhead up to 21%. Our approach also outperforms other recent proposals such as DCC with a decrease of 8% in execution speed.

1. Introduction

Nowadays, market trends are positioning Chip-Multiprocessors (CMPs) as the best way to use the large number of transistors we can fit into a chip. By increasing the number of cores per chip and the size of caches, we aim to improve the performance in an energy-efficient way, as well as keeping a manageable complexity to exploit the thread-level parallelism.

However, with the increase in the number of transistors per chip, the failure ratio continuously grows in every new scale generation [15]. Firstly, the mere fact of having a higher number of transistors per chip, increases the probability of a fault. Secondly, the growth of temperature, the decrease of the voltage supply and the subthreshold voltage in the chip, in addition to other non-desirable effects such as higher power supply noise and signal cross-talking compromise the system's reliability.

Hardware errors can be divided into three main categories: transient faults, intermittent faults and permanent faults [10], [3]. Both transient and intermittent faults appear and disappear by themselves. The differences between them are their duration and their causes: while transient faults are originated by external factors, like particle strikes over the chip, intermittent faults are caused by intra-chip factors, such as process variation combined with voltage and temperature fluctuations [20]. In addition, transient faults disappear much faster than intermittent faults, on average. Finally, permanent faults remain in the hardware until the damaged part is replaced. Thus, this paper is aimed at the study of fault tolerant techniques to detect and recover from transient and intermittent faults, also known as soft errors.

Although the fault ratio is still low for the majority of users, several studies show how soft errors can heavily damage industry [10]. For instance, in 1984 Intel had some problems delivering chips to AT&T because of alpha particle contamination in the manufacturing process. In 2000, a reliability problem was reported by Sun Microsystems in its UltraSparc-II servers as a result of an insufficient protection in the SRAM. A report from Cypress Semiconductor shows how a car factory is halted once in a month because of soft errors [23]. Another fact to take into account is that the fault ratio increases as altitude does. Therefore, reliability has become a major design problem in the aerospace industry.

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

There are many proposals to achieve fault tolerance in microarchitectures. RMT (Redundant Multi-Threading) is a group of techniques based on redundant execution in which two threads execute the same instructions. Simultaneous and Redundantly Threaded processors (SRT) [13] is one of them, implemented on Simultaneous Multithreading (SMT) architectures in which the two threads are executed with a delay respect to the other, called *slack*, which allows to speed up the trailing thread. This approach is attractive since it does not require many design changes in a traditional SMT processor, and it only adds some extra hardware for communication purposes between the threads. It also avoids inter-core communication, one of the major drawbacks of other proposals such as Chip Redundant Threaded

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

processors (CRT) [11], Reunion [16] and Dynamic Core Coupling (DCC) [5]. Although there are several proposals using SRT in uni-processor environments with sequential applications [19][4], the architectural support for redundancy with shared-memory workloads remains unexplored.

The specific contributions of this paper are:

- Building a fault tolerant architecture for shared-memory applications using SRT as a building block for providing reliable executions. At the same time we identify atomic operations as a serialization point which causes a significant performance degradation.
- Proposing a new design for SRT to reduce the bottlenecks of the architecture, which consists of updating L1 data cache before verification.
- Studying the behaviour of a speculative mechanism which allows sharing unverified data.

The rest of the paper is organized as follows. Section 2 provides some background and reviews the related work. Section 3 details a reliable tiled-CMP architecture by using SRT as a building block. In Section 4 we describe a first design which achieves a noticeable performance degradation because of serialization in atomic operations, proposing a new alternative to reduce this bottleneck. Section 5 introduces the methodology employed in the evaluation and shows the experimental results. Finally, Section 6 summarizes the main conclusions of our work.

2. Related work

There are several proposals to detect and recover from soft errors. They can be classified attending to their operation mode, as we can see in Figure 1. In the first category, we find error detection and correction codes. Error-Correcting Codes (ECC) and Forward Error Correction (FEC) are created by specific rules of construction, in order to detect and correct any error caused in the transmission of data automatically. ECC codes are commonly used in dynamic RAM and caches.

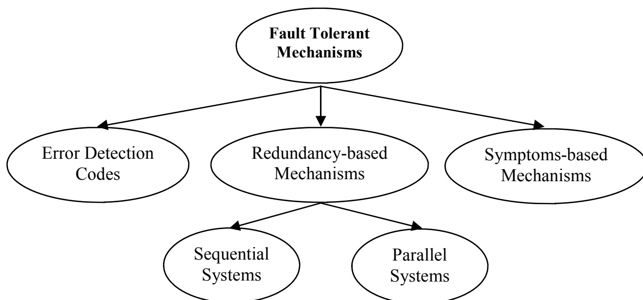

Figure 1. Fault tolerant mechanisms classification.

The second category is composed of redundancy-based mechanisms. The first approaches to full redundant execution started with Lockstepping [1], a proposal in which two statically bound execution cores receive the same inputs and execute the same instructions. Later, the family of techniques Simultaneous and Redundantly Threaded (SRT) [13], Simultaneous and Redundantly Threaded with Recovery (SRTR) [19], Chip Redundant Threading (CRT) [11] and Chip Redundant Threading with Recovery (CRTR) [2] were proposed, based on a previous approach called AR-SMT [14]. In all these studies, the fault tolerance is achieved by means of redundant execution of thread pairs in SMT cores.

The third category is a novel approach to fault detection which follows a scheme based on symptoms [6]. In this study, a characterization of how errors affect either application or OS behaviour with almost zero hardware overhead is presented. By looking for these unusual situations as fatal hardware traps, abnormal application exits or hangs in either the program or the OS, faults can be detected and rolled-back to a previous safe state by means of checkpoints. However, these techniques cannot provide a solution for those errors that do not modify the behaviour of applications. This is the case of errors in ALUs, which simply flip the value of a bit. This error will be left undetected, leading to an erroneous program execution.

When comparing different fault tolerance mechanisms, we can examine four main characteristics. Firstly, the *sphere of replication* [13] which determines the components in the microarchitecture that are replicated. Secondly, the *synchronization* between redundant copies. Thirdly, the *input replication* method, which defines how redundant copies always observe the same data. Finally, the *output comparison* method, which defines how the correctness of the computation is assured.

In SRT(R) the redundant threads are executed within the same core. The sphere of replication includes the entire SMT pipeline but the first level of cache. The threads execute in a *staggered execution* mode, using a strict input replication and output comparison on every instruction. Other studies have chosen to allocate redundant threads in separate cores. This way, if a permanent fault damages an entire core, a single thread can still be executed. Among these proposals it is worth mentioning CRT [11], CRTR [2], Reunion [16], Dynamic Core Coupling (DCC) [5] and Highly Decoupled Thread-Level Redundancy (HDTLR) [12]. In all these proposals the focus relies on how the redundant pairs communicate with each other.

In Reunion, the *vocal core* is responsible for coherently accessing and modifying shared-memory. However, the *mute core* only accesses memory by means of non-coherent requests called *phantom requests*, providing redundant access to the memory system. This approach is called *relaxed input replication*. In order to detect faults, the current architectural state is interchanged among redundant cores by using a compression method called *fingerprinting* [17] through a dedicated point-to-point fast bus. Since relaxed input replication leads to input incoherences masked as fault

detections, the checking interval must be slow (hundred of instructions). This induces a serialized execution (very similar to lock-stepped execution) between redundant cores affecting performance with a degradation of 22% over a base system when no faults are injected.

DCC [5] does not use a special communication channel and reduces the overhead of Reunion by providing a decoupled execution of instructions, making bigger comparison intervals (thousand of instructions) and reducing the network traffic. At the end of each interval, the state of redundant pairs are interchanged and, if no error is detected, a new checkpoint is taken. Input incoherences are avoided by a consistency window which forbids data updates, while the members of a pair have not observed the same value. However, DCC uses a shared bus as interconnection network, which is the best way to detect memory coherence and consistency violations since all cores are able to see every message. However, this kind of buses are not scalable as a result of area and power constraints. In [18], the authors show when a direct network (as a mesh) is used for DCC, the performance degradation rises to 10%, 19%, 39% and 42% for 4, 8, 16, and 32-core CMPs.

Recently, Rashid et al. [12] proposed a fault tolerant architecture in which decoupled threads in different cores redundantly execute the same program instructions. The recovery mechanism is based on checkpoints which reflect the architecture changes between *epochs*. Instead of a consistency window (as DCC), it employs extra hardware to buffer unverified stores (thousand of instructions). In this proposal, the memory system must be severely modified, including a new structure called PCB (Post-Commit Buffer).

3. SRT as a building block for reliability

In this section we describe the base SRT architecture to provide fail-safe execution for sequential applications.

SRT is a fault tolerance architecture proposed by Reinhardt and Mukherjee [13]. In SRT, two threads in a SMT core redundantly execute the same instructions providing transient fault detection. These threads are called *master* (or *leading*) and *slave* (or *trailing*) threads, since one of them runs ahead of the other by an amount of instructions called *slack*. As in a traditional SMT, each thread has its own PC register, renaming map table and register file, while all the other resources are shared.

The master thread is the one responsible for accessing memory to load data. When it does, it bypasses the data loaded, together with the accessed address to the slave thread by a circular FIFO structure called LVQ (Load Value Queue). This structure prevents the slave thread from observing different values from those the master did, a phenomenon called *input incoherence*. However, the master thread never updates memory. When a master's store commits, it records the address and value in a structure called SVQ (Store Value Queue). The slave will access the SVQ for comparison purposes. If the check succeeds, the memory update will be issued to L1 cache. Additionally, to avoid divergent path execution between thread pairs, the master, at the mercy of its prior execution, indicates the correct destination of branches through a structure called BOQ (Branch Outcome Queue).

SRTR [19] was proposed for recovery purposes. It extends the original SRT proposal by adding an additional structure called RVQ (Register Value Queue), as shown in Figure 2. The RVQ is used to bypass register values between threads in order to compare the results from non-memory instructions, before they are committed by the slave. This way, the register file is a safe point used to restart the execution of the master if a fault is detected. However, the pressure over this structure is high and, despite of the efforts made in SRTR by using Dependence-Based Checking Elision (DBCE) [19] to reduce it, it still remains high. An attractive alternative is the creation of regular small checkpoints of the register file as proposed in Cherry [9]. When a fault is detected, the checkpoint is used to recover the register file from both the master and the slave thread. Then, execution is restarted. The integrity of the checkpoints is preserved thanks to the fact that unverified data is not propagated to memory.

In SRT(R) the master thread runs ahead of the slave by an amount of instructions indicated by the *slack*. The benefit of using a slack (staggered execution) in comparison to lock-stepped execution, is twofold. Firstly, the slave avoids executing all the instructions in the wrong path since it obtains the destination for every branch through the BOQ. Secondly, the master thread acts as a prefetcher for the slave, since a master memory instruction, that caused a cache miss, will have been resolved (or almost resolved) by the time the slave thread executes it.

In this paper we propose to create a reliable architecture by adding small two-way SRT cores to form a tiled-CMP. Note that this differs from the original formulation of SRT, in which all the threads were executed within a unique and large SMT processor, being this one a non-scalable approach when the number of threads increases, due to the intrinsic limitations of SMT processors. On the other hand, CRT [11] extends SRT by using two-way SMT cores in a CMP processor, each core executing only two threads. However, in CRT master threads and their corresponding slaves are bound to different cores as an effort to maximize overall throughput. This implies wide data paths between cores to communicate data and to control messages needed by structures like the LVQ, BOQ or the SVQ.

With our new design, we accomplish two requirements. Firstly, we avoid the use of expensive cross-core buses since master and slave threads are executed in the same SMT core. Secondly, we provide high scalability because, in contrast to the original SRT proposal, we use separate dual SMT

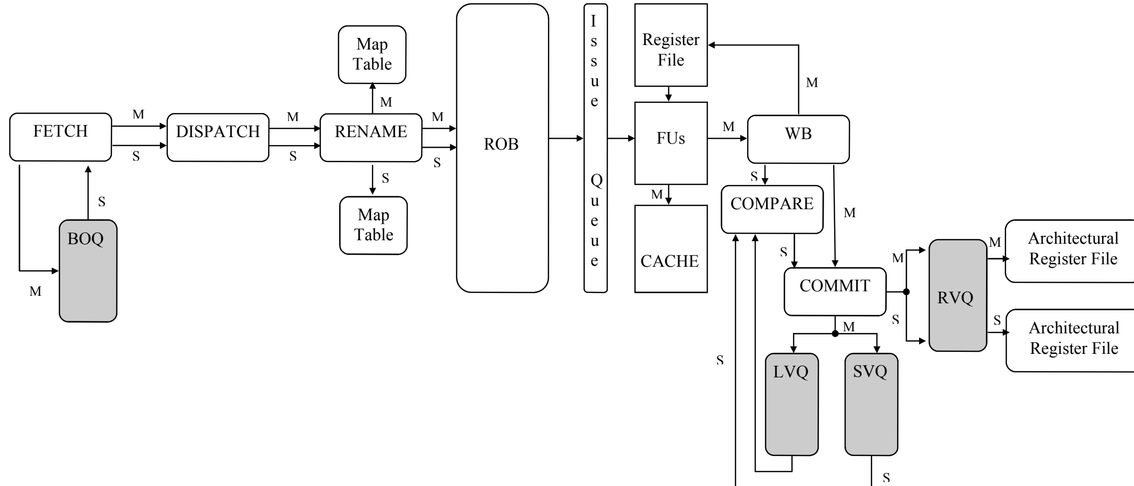

Figure 2. SRT architecture overview.

cores to execute multiple thread-pairs from the same parallel application.

4. SRT in a shared-memory environment

Although SRT was originally proposed and evaluated for sequential applications and the authors argue that it could be used for parallel applications as well, we have found that, with no additional restrictions, SRT can lead to wrong program execution for parallel workloads in a CMP scenario even with the absence of transient faults.

In parallel benchmarks, such as those we can find in SPLASH-2 [21], the access to critical sections is granted by primitives which depend on atomic instructions. In SRT, the master thread never updates memory. Therefore, when a master executes the code to access a critical section, the value of the lock variable will not be updated until the slave executes and verifies the same instructions. This behaviour might lead two (or more) master threads to access a critical section at the same time, which potentially leads to a wrong program execution.

4.1. Synchronizing atomic operations

In order to preserve sequential consistency, and therefore, the correct program execution, one option is to synchronize master and slave threads when executing atomic instructions.

This conservative approach introduces a noticeable performance degradation. The duration of the stall of the master thread depends on two factors: (1) the size of the slack, which determines how far the slave thread is and, (2) the number of write operations in the SVQ which must be written in L1 prior to the atomic operation to preserve consistency.

4.2. Exposing writes to cache

In benchmarks with high contention resulting from synchronization, the constraint previously described can increase the performance degradation of the architecture dramatically. To avoid master stalls deriving from consistency, we propose an alternative design to SRT in which a master thread is able to update L1 cache.

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

To successfully recover from a fault, additional actions must be taken. As L1 cache might have unverified blocks, when a fault is detected all the unverified blocks are invalidated. In order to avoid losing the correct updates performed in a block, when a store is correctly checked by the slave, the word is written-back in L2 cache which is consistent if the block in L1 is invalidated.

4.2.1. Implementation details. We propose to allow the master thread to update L1 cache before checking whether the generated value is correct or not. However, to avoid error propagation due to a wrong result stored in cache, we must identify unverified blocks in cache. In order to do this, we add an additional bit per L1 cache block called *Unverified bit*, activated on any master write. When the Unverified bit is set on a cache block, it cannot be displaced or shared with other nodes. Eventually, the Unverified bit will be cleared when the corresponding slave thread verifies the correct execution of the memory update.

Clearing the Unverified bit is not a trivial task. A master thread can update a L1 cache block several times with-

out checking. If the first check performed by the slave is successful, it means that the first memory update was valid. However, it does not imply that the whole block is completely verified since the rest of the updates have not been checked yet. One simple way of knowing if a L1 block needs more checks before clearing the unverified bit is by checking if the block appears more than once within the SVQ. If it does, more checks need to be performed. Yet, this measure implies an associative search in the SVQ. However, as we said before, we eliminate much of the pressure produced by master loads. In quantitative terms, with the original SRT proposal we had an associative search every master's load, and now we need an associative search every slave's store, which results in a significant reduction of associative searches in the SVQ.

Alternatively, we can use a small counter for every L1 cache block to indicate the number of updates performed by the master. The number of bits per counter depends on the size of the SVQ which indicates the maximum number of unverified stores at a given moment. Experimental results show that 4 bits per L1 cache block are enough, which in a 64KB L1 cache with 64-byte blocks represents an overhead of 4KB. We believe that the first option is better because the pressure over the SVQ is lower than in the original proposal whereas 4KB is an expensive hardware cost.

4.2.2. Speculative sharing. We have studied the effect of sharing unverified blocks. The basic idea is to allow reads on data that have not been verified yet, in essence the mechanism proposed in DCC [5].

In order to avoid an unrecoverable situation, the data speculatively delivered block the commit stage of the requester. This way, we introduce speculative data in the pipeline to operate with (similar to conventional speculative execution due to branch prediction). Furthermore, a block can be speculatively shared with two or more requesters. When the producer validates the block, it sends a signal to all the speculative sharers confirming that the acquired block was correct and the commit stage is re-opened in its pipelines. If a fault is detected in a core which speculatively shared data, an invalidation message is sent to all the sharers which flush their pipeline, undoing the previous work.

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

5. Evaluation

5.1. Simulation environment

We have implemented the proposed extensions to SRT for supporting the execution of parallel applications in tiled CMPs, evaluating the performance results by using the functional simulator Virtutech Simics [7] extended with Wisconsin GEMS [8].

Table 1. System parameters.

Processor Parameters	
Max. fetch/retire rate	4 inst./cycle
Processor Speed	2 GHz.
Consistency model	Sequential consistency
Write-read reordering	Allowed
Cache Parameters	
Line Size	64 bytes
L1 Cache:	
Size	64 KB
Associativity	4 ways
Hit time	1 cycles
Shared L2 Cache:	
Size	512 KB/tile
Associativity	2 ways
Hit time	6+9 cycles (tag+data)
Memory Parameters	
Coherence Protocol	MOESI
Write Buffer	64 entries
Directory Hit Time	15 cycles
Memory Access Time	300 cycles
Network Parameters	
Topology	2D-Mesh
Link Latency (one hop)	4 cycles
Routing Time	2 cycles
Flit Size	4 bytes
Link bandwidth	1 flit/cycle
Fault Tolerance Parameters	
LVQ	64 entries
SVQ	64 entries
Slack Fetch	256 instructions

Our study has been focused on a tiled CMP where each core is a 2-threaded SMT which has its own private L1 cache, a portion of the shared L2 cache and a connection to the on-chip network. The architecture follows the sequential consistency model with the write-read reordering optimization. The main parameters of the architecture are shown in Table 1. For the evaluation, we have used a selection of scientific applications: Barnes, FFT, Ocean, Radix, Raytrace, Water-NSQ and Water-SP are from the SPLASH-2 benchmark suite [22]. Tomcatv is a parallel version of a SPEC benchmark and Unstructured is a computational fluid dynamics application. The experimental results reported here correspond to the parallel phase of each program only. Problem sizes are shown in Table 2.

Table 2. Simulated benchmarks.

Benchmark	Size
Barnes	8192 bodies, 4 time steps
FFT	256K complex doubles
Ocean	258x258 ocean
Radix	1M keys, 1024 radix
Raytrace	teapot
Tomcatv	256 elements, 5 iterations
Unstructured	Mesh.2K, 5 time steps
Water-NSQ	512 molecules, 4 time steps
Water-SP	512 molecules, 4 time steps

5.2. Performance analysis

We have implemented all the SRT mechanisms described in Section 4. SRT-base corresponds to the SRT implementation where atomic operations became a synchronizing point as explained in 4.1. With SRT-FT we refer to the implementation where stores are exposed to cache as seen in Section 4.2, and finally, SRT-Speculative is the implementation of the speculative sharing in SRT as seen in Section 4.2.2.

We have noticed that SRT-base is very sensitive to write-read reordering. If this capability is not activated, every load must wait for the verification and issue of all the pending stores in the SVQ before being issued, which implies a huge performance degradation. Thus, in fairness to the study, we have allowed for this optimization.

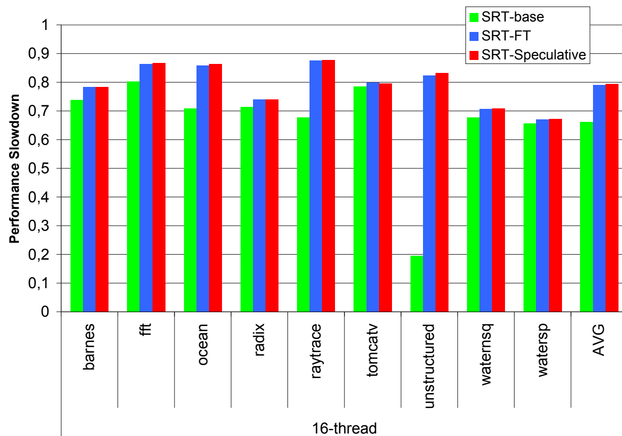

Figure 3. Performance impact of SRT mechanisms over a non fault-tolerant architecture.

We have simulated the benchmarks listed in Table 2 in a tiled CMP with 16 cores. The performance results are showed in Figure 3. Overall, these results suggest that SRT-FT and SRT-Speculative perform better than SRT-base. This tendency is more noticeable in Unstructured or Raytrace which have many more atomic synchronizations than the rest of the studied benchmarks, as we can see in Table 3.

Table 3. Atomic synchronizations per 100 cycles.

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

SRT-base obtains an average degradation of 34% in performance with regards to a non-redundant system. On the contrary, SRT-FT is able to reduce this degradation up to 21%. Atomic operations damage SRT-base in two ways. Firstly, they act as a serialization point: the slave thread must catch up with the master. In SRT-FT there are no such serialization points. Secondly, to preserve the sequential consistency, all the stores in the SVQ must be issued to memory before the actual atomic operation. Unlike SRT-base, in SRT-FT the stores are issued to L1 cache without verification. Therefore, it is unusual for an atomic instruction to wait for consistency risks.

However, the speculative mechanism implemented does not increase the execution speed. In SRT, the verification of a memory update takes a short time (in the order of hundred of cycles), which creates a small opportunity window to obtain a benefit from speculation. A larger slack would benefit from this mechanism, but it would also require bigger sizes in critical structures like the SVQ or LVQ.

5.3. Comparison against DCC

DCC [5] provides a fault tolerant framework based on dynamic binding of cores for re-execution, relying on the use of a shared bus. However, for current and future CMP architectures, more efficient designs are tiled CMPs, which are organized around a direct network, since area, scalability and power constraints make impractical the use of a bus as the interconnection network.

We have evaluated DCC on the top of a direct network using the same parameters as seen in Table 1. As studied in [18], DCC performs poorly in this environment due to the latency imposed for the *age table* introduced to maintain the master-slave consistency. Figure 4 shows the execution speed for SRT-FT and DCC normalized to a non-redundant tiled CMP with 16 cores. As we can see, SRT-FT is 8% faster than DCC on average. Furthermore, DCC uses twice as much hardware as SRT-FT, because the redundant threads are executed in different cores. This represents another advantage of SRT-FT over DCC.

6. Conclusions

Processors are becoming more and more unreliable due to several factors such as technology scaling, voltage reduction and signal cross-talking. This makes processors more susceptible to transient faults also known as *soft errors*. Although there are many approaches exploring reliability for single-threaded applications, the multithreaded, shared-memory environment has not been completely studied.

In this paper, we study the under-explored architectural support for SRT to reliably execute shared-memory applications. First, we have shown how atomic operations induce a serialization point between master and slave threads. This bottleneck has an impact of 34% on average in the execution speed over several parallel scientific benchmarks.

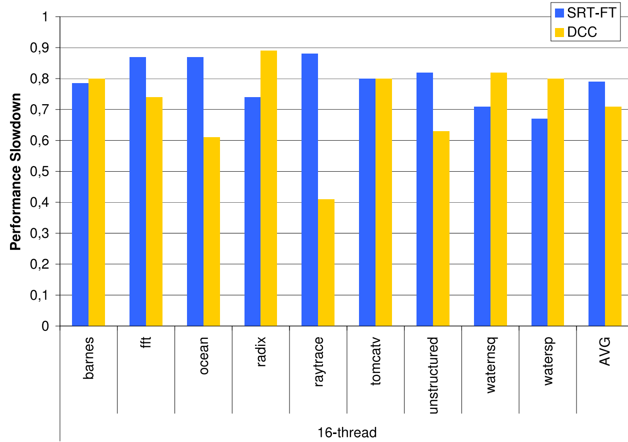

Figure 4. Performance impact comparison between SRT-FT and DCC.

Besides, to decrease this performance degradation, we propose allowing for updates in L1 cache before verification. Thus, we obtain a more decoupled execution reducing the stall time due to synchronization. To avoid fault propagation among cores, unverified data reside in L1 cache in which sharing is not allowed as a conservative measure. With this mechanism, we can reduce the performance degradation to 21%. This approach is better than the one obtained with other proposals like DCC, which has a negative impact of 29% in a direct network while using twice the number of cores than SRT.

In addition, as a way to improve the execution speed, we have studied a simple speculative mechanism for sharing unverified data in the same fashion as DCC. However, the results show that this mechanism does not obtain a better performance. This is due to the fact that in DCC, the cores which obtain unverified data can commit, since the old state is preserved by checkpoints, unlike in SRT, in which the commit stage is stalled to avoid an unrecoverable error.

As future work, we plan to study sharing patterns in order to predict future shared blocks so as to speed up their verification. Furthermore, to increase slave thread speedup we will also study out-of-order fetch models obtained from the master thread execution to accelerate critical path instructions.

Acknowledgements

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

References

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

[2] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proc. of the 30th annual Int' Symp. on Computer architecture (ISCA'03)*, San Diego, California, USA, 2003.

[3] A. González, S. Mahlke, S. Mukherjee, R. Sendag, D. Chiou, and J. J. Yi. Reliability: Fallacy or reality? *IEEE Micro*, 27(6), 2007.

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

[5] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proc. of the 37th Annual IEEE/IFIP Int' Conference on Dependable Systems and Networks (DSN'07)*, Edinburgh, UK, 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' Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, Seattle, WA, USA, March 2008.

[7] 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.

[8] 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.

[9] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proc. of the Int' Symp. on Microarchitecture (MICRO'02)*, Istanbul, Turkey, Nov. 2002.

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

[11] S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proc. of the 29th annual Int' Symp. on Computer architecture (ISCA'02)*, Anchorage, Alaska, USA, 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 Performance Computer Architecture (HPCA'08)*, Salt Lake City, USA, 2008.

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

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

[15] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on soft error rate of combinational logic. In *Proc. of the Int' Conference on Dependable Systems and Networks (DSN'02)*, Bethesda, Maryland, USA, 2002.

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

[17] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Fingerprinting: Bounding soft-error-detection latency and bandwidth. *IEEE Micro*, 24(6), 2004.

[18] 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 (WDDD'08). In conjunction with ISCA'08*, Beijing, China, 2008.

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

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

[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] 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.

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