Adapting Dynamic Core Coupling to a direct-network environment

Daniel Sánchez, Juan L. Aragón and José M. García[1]

Abstract— **To obtain benefit of the increasing transistor count in current processors, designs are leading to CMPs that will integrate tens or hundreds of processor cores on-chip. However, scaling and voltage factors are increasing susceptibility of architectures to transient, intermittent and permanent faults, as well as process variations.**

A very recent solution found in literature consists of Dynamic Core Coupling (DCC) [1]. DCC provides a fault tolerant framework based on dynamic binding of cores for re-execution. This technique relies 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. In this work, we present the changes needed in the original DCC proposal to be used for a direct network environment. These changes are mostly due to the replacement of a bus for a mesh as interconnection network, the coherence protocol and the consistency window. Our evaluations show that, for several parallel scientific applications, the performance overhead with this new environment rises to 10%, 19%, 42.5% and 47% for 4, 8, 16 and 32 core pairs, respectively, compared to the 5% performance degradation as previously reported for 8 core pairs in the original DCC proposal.

I. Introduction

Nowadays, market trends are positioning CMPs as the best way to use the big number of transistors that we can accommodate in a chip. However, due to the raise of the number of transistors per chip, the failure ratio is increasing more and more in every new scale generation. On one hand, the actual larger number of transistors in a chip enlarges the probability of fault. On the other, the increase of the temperature and the decrease of the voltage in the chip leads to a higher susceptibility to transient faults. A transient fault is a flip in one or more bits. It may be caused as a result of the impact of an alpha particle on the chip or causes such as power supply noise and signal cross talking. All zones in the chip are vulnerable to this kind of faults, therefore, fault tolerance mechanisms must be designed to avoid incorrect program executions. Moreover, these techniques always have both a hardware cost because of the additional extra hardware required to re-execute instructions, and a performance cost because of the actions needed to assure a correct execution.

The family of techniques SRT [2], SRTR [3], CRT [4] and CRTR [5] are based on a previous proposal called AR-SMT [6], in which redundant threads execute the same instructions in an SMT processor with a performance degradation between 10-30%. In all these studies, the fault tolerance is achieved by redundant execution in two different execution cores (or threads) called *leading/master* and *trailing/slave*. The master core runs some instructions ahead of the slave and they communicate with each other by some different structures, like the LVQ, StB or RVQ. Although applicable to sequential programs, these techniques are not directly valid for executing parallel programs due to incoherences in memory values called *input incoherences* [7].

An input incoherence is a phenomenon that occurs when two dynamic loads do not obtain the same value from memory. This problem is very common in parallel programs when a redundant core executes the same instruction few cycles later. Reunion [7] addresses this problem with a new paradigm called *relaxed input replication*, in which the master issues non-coherent accesses to memory (*phantom request*) while the slave core issues real coherent accesses. If a difference is detected because of an input incoherence, it is marked as a transient fault when indeed it is not.

In order to avoid the need of intermediate structures to communicate the leading and the trailing cores, another option could be the periodic creation of checkpoints. To detect any fault between two checkpoints, the master and the slave interchange a signature or a hash resuming the current state, detecting a fault if they differ. The recovery mechanism is as easy as going back to the last successfully verified checkpoint, which establishes a safe point. A very recent study on this fashion is made by LaFrieda *et al.* in DCC [1]. DCC is a promising approach to achieving fault tolerance in multiprocessors, based on a shared bus.

In the present work, we analyse and evaluate how DCC behaves in a scalable tiled-CMP architecture. We show that the DCC execution time overhead is more noticeable than previously reported when considering direct networks. We have evaluated in detail the scalability, the influence of cache associativity, the delay of L1 replacements and the total network traffic. We have found that the main cause for this increasing execution time overhead is the mechanism used by DCC to assure the memory consistency between master and slave cores.

The rest of the document is organized as follows: Section II reviews how DCC operates and points out its major weaknesses. Section III presents how to migrate DCC to work under a direct network instead of a shared-bus. Section IV introduces the methodology employed in the evaluation. Section V shows the performance results. Section VI summarizes the

[1]Dpto. de Ingeniería y Tecnología de Computadores, Univ. de Murcia, e-mail: {dsanchez, jlaragon, jmgarcia}@ditec.um.es

main conclusions of our work and, finally, Section VII indicates our future work.

II. Background

A. Understanding DCC

DCC is a fault tolerance mechanism for sequential and parallel applications too. To achieve fault tolerance, DCC re-executes instructions in a redundant core and, after a variable number of cycles, cores exchange their state with their pairs by means of a hash. If the hashes match with each other, the actual state of the architecture constitutes a safe point and it is saved as a checkpoint.

In DCC, a node is formed by the master core and the slave core. Both, master and slave, access memory to bring back new data to private cache, but just the master is permitted to writeback data to shared memory. However, if the data to be back-written in the master core has not been checked against that data in the slave core yet, the operation is aborted. To identify those data blocks, L1 cache must be modified to add an *unverified bit* that indicates that a block has been modified by a core but has not been verified by its redundant core.

The unverified marks are cleared at the end of a checkpoint interval when the state of both, master and slave cores, have been verified. A checkpoint interval is set to 10,000 cycles in DCC [1]. However, there are some causes that induce to the creation of a new checkpoint before the interval is over. These causes are: interrupts, I/O instructions, context switches, but above all, overflows in cache lines. As said before, a block marked as unverified cannot be replaced from private cache. Therefore, the replacement policy should be modified to avoid replacing blocks marked as unverified. However, all blocks in a cache line are sometimes marked as unverified which is called a *cache buffering overflow*. When such an overflow appears, DCC needs to create a new checkpoint in order to have the chance of replacing an unverified block.

B. DCC in a parallel environment

DCC is a fault-tolerant mechanism that also works for parallel architectures. In order to facilitate that, DCC needs to guarantee the memory coherence and consistence.

B.1 Coherence

As cited before, in DCC, both master and slave cores issue requests to shared memory although just master cores can update it. This behaviour implies several modifications in the coherence protocol in order to avoid coupled cores to fight for data blocks, resulting in performance degradation and incorrect program output. To solve this issue, coherence actions from requests between coupled cores are ignored at destination. This also implies that requests from external cores can cause invalidations in slave cores which will never provide values since this task is reserved just for master cores.

B.2 Consistency

In DCC, the consistency problem is solved by using a structure called *age table*. Its aim is to prevent an external write from modifying a value between the time that the leading thread has read a value and the trailing thread has not yet.

When a processor wants to perform a write, issues a read-exclusive or upgrade request to the shared bus. Then, the message is read by both master and slave cores, which perform two actions. First, they observe if they have a match between the address of the message and its load queue and, if there is, a NACK is submitted to prevent the write which, in turn, will be retried later. When the master receives the age, it is compared with its own one and, if they are different, it means that one of the cores has committed an instruction to that address and the other has not. In other words, if the block leaves the cache, there will be a potential consistency error, so a NACK is submitted.

III. Making DCC scalable using a tiled-CMP architecture

To make the original DCC scalable we move from a shared-bus architecture towards a more scalable topology in a tiled-CMP architecture like a 2D-mesh. But changing from a shared-bus has several implications on how DCC works. For further details, see the extended version of this paper [8].

A. Changes to the coherence protocol

As in the original DCC paper, we have selected MOESI as our coherence protocol. Modifications needed to accommodate DCC in a direct network are described as follows.

A.1 Slave coherence

Since we no longer have a shared-bus, every core in the direct network must know the core-role mapping as well as the pairs (or trios) formed by the OS when the fault tolerance mode is on. The main difficulty when operating on this mode is that the protocol would have to track multiple owners for a block (master and slave), because in any cycle both could have that permission. Instead of that, we have modified DCC as follows: any upgrade or read-exclusive request from a node to a master core should be sent to its slave, too. In this way, an invalidation of the block in the master, will not cause a later coherence fail in the slave, since it has also seen the request. However, this simple solution results in an increment in the number of messages through the network, as we evaluate in Section V-D.

A.2 Consistency

As we saw in Section II-B.2, consistency in DCC is achieved by adding a structure called age table which is accessed when a read-exclusive or upgrade request arrives. In a shared-bus, one-single message is seen by all processors, but in a direct network it does not. The mode of operation in our scenario is different

and every read-exclusive or upgrade has to be sent to the master as well as to its slave. Then, each core accesses to its LSQ, looking for a match in the address of the request. Upon a positive match, the request cannot be served. At this point, we consider that, instead of sending a NACK to the requestor, it would be better to keep the request in the core until it can be satisfied, saving some bandwidth in the network. The request will be periodically retried in the core until it could complete.

In parallel to the access to the LSQ, the slave core sends its age from the age table to its master. The master compares the received age with its own and, if they differ, the request cannot be served, retrying later. Conversely, if the ages match, the request can be satisfied, the master is invalidated and it sends a mandatory invalidation to its slave. Again, this modification to the original DCC proposal results in an increase in the number of messages sent between the requestor and the master and slave to be invalidated.

A.3 Other considerations

In the original DCC proposal, the authors point out that the replacement of unverified blocks in L1 cache causes a buffering overflow solved with the creation of a new checkpoint. However, we have found that there is a potential consistency risk when replacing non-unverified blocks when using direct networks.

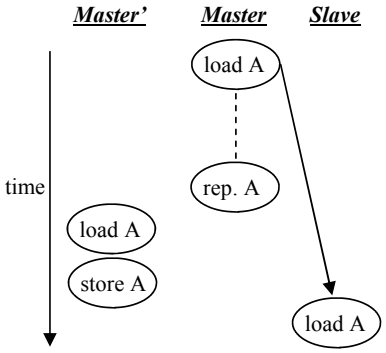

Fig. 1. Potential consistency error in DCC.

As we can see in Fig. 1, some actions could lead to a consistency error between *Master* and *Slave* in DCC. After the replacement of block A in *Master*, another core, *Master'*, acquires the block and eventually modifies it. When *Slave* executes the redundant load, it will perceive a different value. In the original DCC proposal with a shared-bus, the consistency window is capable of resolving this conflict. In spite of having the block replaced, *Master* can see through the bus that an external core has issued a read-exclusive or an upgrade request, therefore aborting the request. However, in a direct network like a mesh, *Master* does not notice that another core wants to acquire the block for writing purposes, since there is no information to guide the message from the requestor (*Master'* in Fig. 1) to the old holder of the block which replaced the block.

To imitate the shared-bus DCC behaviour, for every cache miss, the request for the block should be flooded all over the network, in order to avoid consistency errors. This solution, however, creates a large amount of network traffic with a big latency. A simpler solution would be, on every replacement of the leading core, checking that its pair has read the block. If the partner possesses the block, the replacement can be executed. If not, we will delay it until the pair reads the block some cycles later, causing a necessary extra overhead in L1 replacements. In this way, we will solve potential consistency errors between masters and slaves.

Another relevant aspect when using a direct network to fit the original DCC proposal, is synchronization when creating checkpoints, as we can see in Fig. 2. The synchronization request is issued at the end of an scheduled interval, or when events such as buffering overflows occur. In our direct network, the responsible for sending the synchronization request is called *Initiator*. The *Initiator* has to send a message to every master in the system. When the request is received, each master is synchronized with its slave-pair, creating and exchanging its state using a fingerprint. If fingerprints match with each other, an acknowledgment is sent back to the *Initiator*. Once all the acknowledgements have been received, the *Initiator* finally sends a message to each core in the system, giving the order to save the current state as the last checkpoint. After that, all cores resume execution. Besides, if one core finds a mismatch when comparing fingerprints, a NACK indicating a transient fault detection will be sent to the *Initiator*, which will expand the information, causing every core in the system to rollback to its previous saved checkpoint.

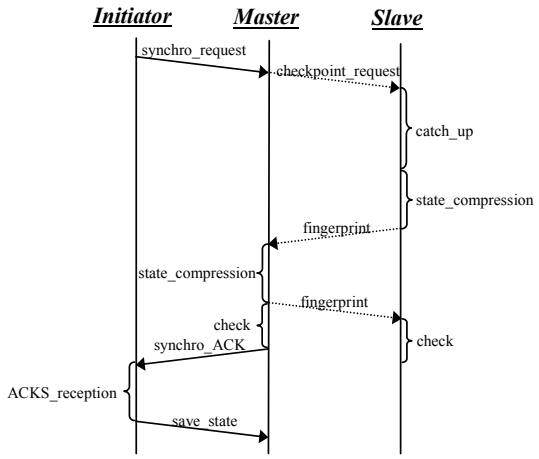

Fig. 2. Synchronization and checkpoint creation.

This mechanism for creating new checkpoints displays a variable latency directly dependent on the distance and the network congestion between the *Initiator* and the furthermost core. Thus, the *Initiator* will not send the save-state request until all ACKs confirming the synchronization have arrived. If any message could not arrive due to a permanent fault in one core, the *Initiator* would be waiting in an infinite loop. To avoid this situation, a timeout is set

up when waiting for ACKs. In our simulations, we have found that each checkpoint takes around 250-400 cycles, depending on the number of nodes and pairs allocation.

IV. Simulation Environment

We have ported the original DCC proposal with the changes explained in the previous section, to the functional simulator Virtutech Simics extended with Wisconsin GEMS v2.1. GEMS provides a detailed memory simulation through a module called Ruby and a pipeline simulation module called Opal. In order to accommodate all DCC constraints, we have modified the MOESI coherence protocol as well as the pipeline behaviour. The interconnection network has been simulated by the module Garnet.

TABLE I
System Parameters.

Processor Parameters	
Max. fetch/retire rate	4 inst./cycle
Processor Speed	2 GHz.
Cache Parameters	
Line Size	64 bytes
L1 Cache:	
Size	32 KB
Associativity	4 ways
Hit time	2 cycles
Shared L2 Cache:	
Size	512 KB/tile
Associativity	4 ways
Hit time	6+9 cycles (tag+data)
Memory Parameters	
Coherence Protocol	MOESI
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	
State Compression Latency	35 cycles
State Checkpoint Latency	8 cycles
Age Table Size	64 entries
Checkpoint Interval	10,000 cycles

The simulated system is a tiled-CMP consisting of a number of replicated cores (tiled) connected by a switched 2D-Mesh direct network. Each core has its own private L1 cache, a portion of the shared L2 cache and a connection to the on-chip network. Table I shows the main parameters of the simulated system.

All the simulations have been conducted using several scientific programs. Barnes, Cholesky, FFT, Ocean, Radix, Raytrace, Water-NSQ, and Water-SP are from the SPLASH-2 benchmark suite. Tomcatv is a parallel version of a SPEC benchmark and Unstructured is a computational fluid dynamics application. The results of the simulations have been extracted from the parallel phase of each benchmark executed for 2, 4, 8, 16 and 32 application threads and 4, 8, 16, 32 and 64 cores, respectively.

V. Evaluation Results

A. DCC overhead in a direct network

We have compared the performance results of the extended DCC proposal in a direct network scenario, with the results in a non-fault tolerance base scenario. Fig. 3 reports the execution time overhead which DCC obtains with respect to a non-fault tolerance system for 2, 4, 8, 16 and 32 nodes. As we can see, DCC incurs in a noticeable time overhead, more severe in applications like Ocean, Raytrace and Unstructured. The time overhead is splitted into *Checkpoint_time*, the time employed in the creation of new checkpoints, and *Window_time*, the overhead obtained as a result of the actions of the consistency window that were explained in sections II-B.2 and III-A.2.

Our results coincide with the original DCC proposal where leading and trailing cores are separated 100 cycles on average. Consequently, any read-exclusive or upgrade request to modified blocks are delayed until the window is closed by the trailing core. It can be observed that, on average, the execution time increases with the number of nodes as a result of the growth of the network traffic. This way, consistency window constraints affect more deeply system configurations with many cores. In conclusion, the time overhead grows to 6.4%, 10.2%, 19.2%, 42.5% and 47.1% when considering 2, 4, 8, 16 and 32 nodes respectively. In addition, although the time needed by a checkpoint is not negligible, its impact on performance degradation is hidden by the huge overhead caused by the consistency window. On the other hand, checkpoint time creation is responsible for an increase in the time overhead between 2.6% and 4.6% with no significant differences when varying the number of nodes.

The results obtained contrast with those obtained in the original DCC proposal when using a shared-bus as the interconnection network. As reported in [1], the execution time overhead is just 3.9%, 4% and 4.9% for 2, 4 and 8 nodes respectively.

B. Delay in L1 replacements

As observed in Section III-A.3, there is a potential consistency risk when replacing cache blocks. To prevent it, we propose the replacement of blocks after an additional check with the trailing core. Results in the previous section consider that additional check and the corresponding replacement delay. However, in order to measure the effect of this extra delay on the execution time overhead, we run some experiments assuming non-delayed replacements.

In Fig. 4 we can observe that, without replacement delays, the execution time is lower, as expected. However, there is still a significant overhead compared to a non-fault tolerance system. Summarizing, the delay introduced because of the L1 replacements causes an extra overhead of 2%, 3.2%, 3.6% and 7.7% for 4, 8, 16 and 32 nodes, respectively, which is not negligible, specially when considering a higher number of cores.

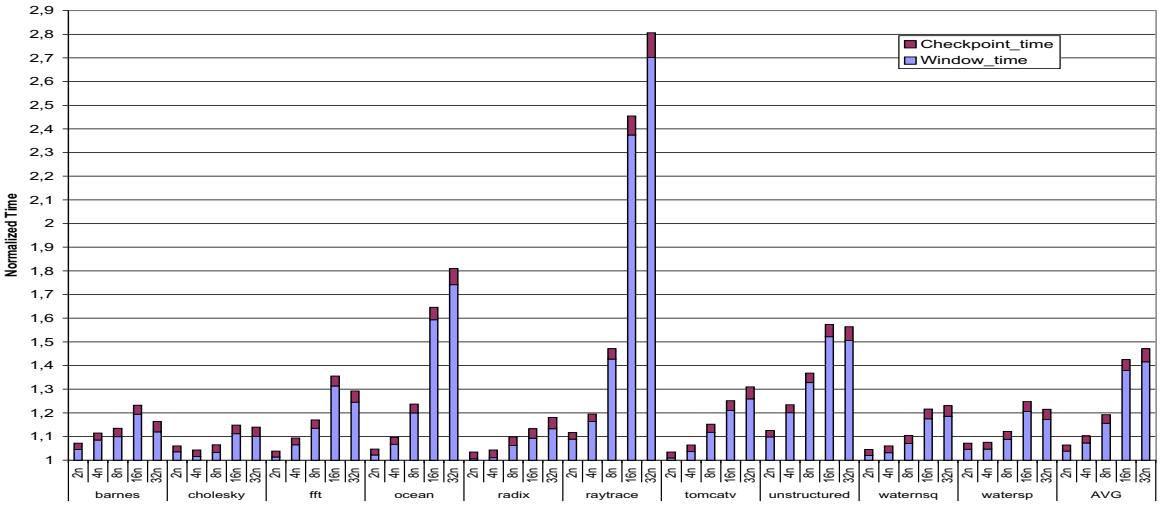

Fig. 3. Execution time overhead in DCC with respect to a non-fault tolerance system.

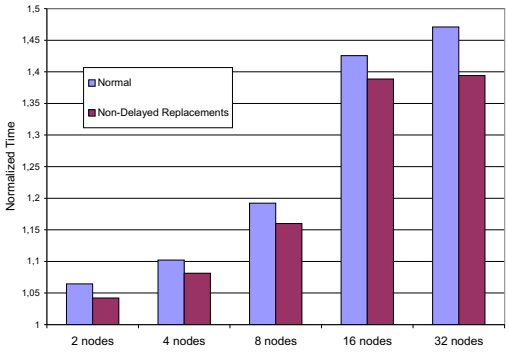

Fig. 4. Execution time overhead because of delayed L1 replacements.

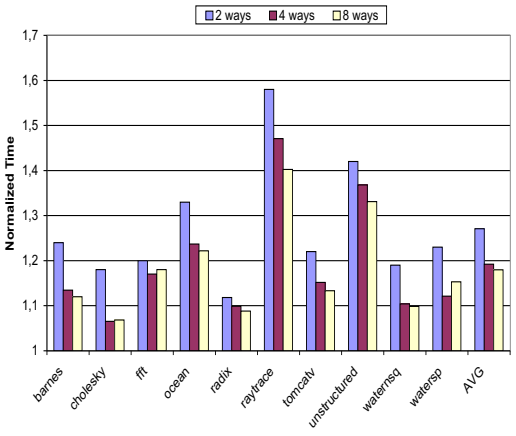

Fig. 5. Execution time overhead in DCC with different L1 cache associativities.

C. Cache associativity analysis for DCC

Cache associativity is of paramount importance for the original DCC proposal. We must bear in mind that, if an unverified block is replaced, a new checkpoint must be performed. So, the replacement policy has been modified to avoid to pick unverified blocks to be replaced. If the number of ways is too small, there exists a performance degradation because sometimes, when allocating new blocks, there are only available a reduced number of cache lines.

Although 4-way L1-caches are common in real processors nowadays, some companies still advocate for 2-way caches. As we can see in Fig. 5, the time overhead of DCC with 8 nodes for a 2-way L1 cache, is around 27% and, when the number of ways is increased, the overhead goes down. It can be also observed that an 8-way cache does not perform much better than a 4-way cache. So, due to the hardware cost that an 8-way cache represents, we conclude that, for DCC, the best option is a 4-way cache.

Fig. 6 shows how many cycles there are between the creation of two checkpoints. As suggested in [1], this number is initially fixed to 10,000 cycles by the system. However, as a result of the cache buffering overflow phenomenon associated to DCC approach, we need to do some unscheduled checkpoints. With 2-way associative caches, with much more overflows, the time between checkpoints ranges from 4,500 cycles in the best case to 3,700 cycles in the worst. In conclusion, with 2-way caches it will be needed the creation of approximately twice the number of checkpoints than in an 8-way cache configuration, leading to a considerable performance degradation (as showed in Fig. 5).

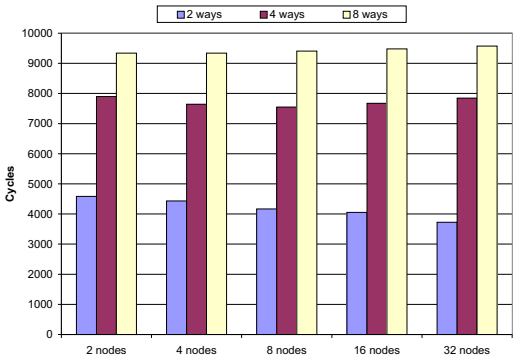

Fig. 6. Checkpoint time interval in DCC depending on L1 cache associativity.

D. Traffic Network Increase for DCC

It is obvious that replacing a shared-bus with a direct network like a 2D-mesh leads to a natural overhead in the number of messages within the network. In a direct network, we lose the broadcast capabilities incurring, then, in a larger number of messages in the network. However, the DCC approach generates additional extra traffic, as a consequence of changing a shared-bus with a 2D-mesh. This can be summed up in that we need to assure that every message seen by the leading core is also seen by the trailing one. The simpler solution is sending the message to both of them. Also, there is more extra traffic in the checkpoint creation phase, because the synchronization mechanism is more complex in a mesh than in a shared-bus.

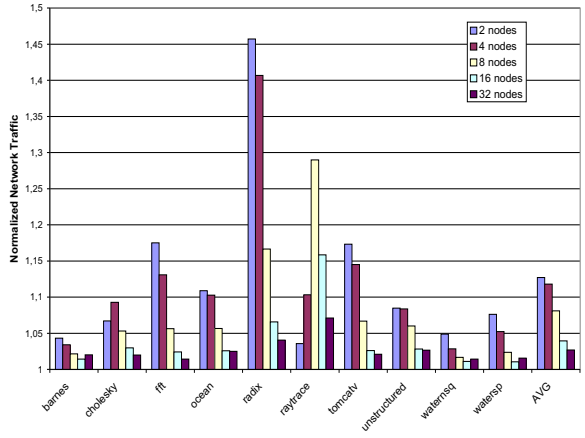

Fig. 7. Network traffic increase in DCC in a mesh.

The total traffic is increased in 12.6%, 11.7%, 8.1%, 3.9% and 2.6%, on average, for 2, 4, 8, 16 and 32 nodes, respectively, for a 4-way associative cache, as we can see in Fig. 7.

VI. Conclusions

In this paper we have shown how DCC could fit in a more realistic and scalable architecture as a tiled-CMP. Although there are some complications with the coherence protocol and the consistency window, DCC could be adapted to use a direct network instead of a shared-bus network. However, all these changes result in noticeable performance degradation. Simulations with SPLASH-2 benchmarks and other scientific parallel programs show that the execution time overhead is 10%, 19.5%, 39% and 42% for 4, 8, 16, and 32 core pairs, respectively, in contrast to the 5% performance degradation reported for the original DCC proposal in a shared-bus with 8 core pairs.

In addition, we have also shown that the performance degradation is mostly due to the consistency window needed to permit, for both master and slave, the access to the shared memory. The traffic network is also increased with a direct network because the DCC mechanism is not properly adapted to this environment. We have also pointed out how L1 cache associativity is an important fact to bear in mind regarding DCC behaviour, in terms of number of cache buffering overflows and, consequently, in performance degradation.

To conclude, in this paper we have seen that, although DCC is a promising approach, when moving towards a scalable tiled-CMP architecture with an increasing number of core pairs, there is still room for improvement in the design of a low overhead and resilient chip multiprocessor.

VII. Future work

Due to the overhead obtained by DCC because of its consistency window, we think that a better approach to fault tolerant architectures should avoid redundant accesses to shared memory. This also will reduce the coherence mechanism complexity, leading to less network traffic overhead. Proposals like SRTR [3] or CRTR [5] provide full tolerance but with a noticeable performance degradation because of resource contention in both the core and the communication between master and slave threads. As part of our future work, we will study the potential of redundant mechanisms based on SMT processors. The communication between cores will be based on structures in a SRTR fashion in order to avoid the overheads incurred by DCC for this reason. Finally, we will explore solutions as value prediction which would lighten the pressure over the ROB and other critical structures leading to less performance degradation. This way we would be able to keep a moderated simple microarchitecture without changes in coherence and consistency mechanisms.

Acknowledgements

This work has been jointly supported by the Spanish MEC and European Commission FEDER funds under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03", and also by the EU FP6 NoE HiPEAC IST-004408.

References

[1] Christopher LaFrieda, Engin Ipek, Jose F. Martinez, and Rajit Manohar, "Utilizing dynamically coupled cores to form a resilient chip multiprocessor," in *DSN'07*, Edinburgh, UK, 2007.

[2] Steven K. Reinhardt and Shubhendu S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *ISCA'00*, Vancouver, British Columbia, Canada, 2000.

[3] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient fault recovery using simultaneous multithreading," in *ISCA'02*, Anchorage, Alaska, 2002.

[4] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *ISCA'02*, Anchorage, Alaska, 2002.

[5] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz, "Transient-fault recovery for chip multiprocessors," in *ISCA'03*, San Diego, California, 2003.

[6] Eric Rotenberg, "Ar-smt: A microarchitectural approach to fault tolerance in microprocessors," in *FTCS'99*, Madison, Wisconsin, 1999.

[7] Jared C. Smolens, Brian T. Gold, Babak Falsafi, and James C. Hoe, "Reunion: Complexity-effective multicore redundancy," in *MICRO 39*, Orlando, Florida, 2006.

[8] Daniel Sánchez, Juan L. Aragón, and José M. García, "Evaluating dynamic core coupling in a scalable tiled-cmp architecture," in *WDDD'08*, Beijing, China, 2008.