Fault-tolerant cache coherence protocols for CMPs: evaluation and trade-offs

Ricardo Fernández-Pascual[1], José M. García[1], Manuel E. Acacio[1] and José Duato[2]

[1] Departamento de Ingeniería y Tecnología de Computadores
Universidad de Murcia, 30100 Murcia (Spain)
{rfernandez,jmgarcia,meacacio}@ditec.um.es
[2] Dpto. de Informática de Sistemas y Computadores
Universidad Politécnica de Valencia, 46022 Valencia (Spain)
jduato@disca.upv.es

Abstract. One way of dealing with transient faults that will affect the interconnection network of future large-scale Chip Multiprocessor (CMP) systems is by extending the cache coherence protocol. Fault tolerance at the level of the cache coherence protocol has been proven to achieve very low performance overhead in absence of faults while being able to support very high fault rates. In this work, we compare two already proposed fault-tolerant cache coherence protocols in a common framework and present a new one based in the cache coherence protocol used in AMD Opteron processors. Also, we thoroughly evaluate the performance of the three protocols, show how to adjust the fault tolerance parameters of the protocols to achieve a desired level of fault tolerance and measure the overhead achieved to be able to support very high transient fault rates.

1 Introduction

The number of transistors available due to current technology trends have enabled the design of progressively more powerful chips. However, these trends have several drawbacks which need to be overcome. Notably, the complexity of designing a system which takes advantage of so many components has forced architects to think of ways to simplify the design. This way, Chip Multiprocessors [2,7] have proved to be a viable way for building newer systems by exploiting thread-level parallelism. Further, tiled CMPs [14] which are built by replicating several *tiles* comprised by a core, private cache, part of a shared cache and an interconnection network interface further help in keeping complexity manageable, scale in a power-efficient way to larger number of cores and support a family of products with a varying number of tiles.

A main drawback of these trends is that, due to the miniaturization and the lower voltages, the susceptibility of future chips to transient failures will increase. Transient failures [11], also known as soft errors, occur when a component produces an erroneous output but continues working correctly after the event. Any event which upsets the stored or communicated charge can cause soft

errors. Typical causes include alpha-particles strikes, cosmic rays, radiation from radioactive atoms which exist in trace amounts in all materials, and electrical sources like power supply noise or radiation from lightning.

The increased importance of transient failures means that fault-tolerance measures have to be considered across all levels of chip design. Even for commodity systems, reliability needs to be above a certain level for the system to be useful for anything. In fact, since the number of components in a chip increases and the reliability of each component decreases, it is no longer economical to design and test assuming a worst case scenario for new chips. Instead, new designs will target the common case and assume a certain rate of transient failures.

One of the components which will be affected by transient failures in a CMP is the interconnection network (IN). The IN occupies a significant part of the chip real estate and is critical to the performance of the system. It handles the communication between the cores and caches, which is done by means of a cache coherence protocol. This requires very small and frequent messages. Hence, to achieve good performance the IN must provide very low latency and should avoid acknowledgment messages and other flow-control messages as much as possible.

Fault tolerance in the IN can be provided at the network level. There are several recent proposals [3, 12, 13] exploring this approach. Ensuring the reliable transmission of all messages imposes significant overheads in latency, power consumption and area. In contrast, we propose to deal with transient errors in the IN at the level of the cache coherence protocol. This allows for more flexibility to design a high-performance on-chip network which can be unreliable. At the same time, the higher level information available to the coherence protocol enables it to achieve fault tolerance but avoids using acknowledgment messages in most cases, protecting only those messages which are critical for correctness. These few acknowledgments are sent out of the critical path of coherence transactions to minimize the effect of fault tolerance on performance.

We have already proposed two fault-tolerant cache coherence protocols which are described in detail in previous works [5, 6]. The contributions of this paper are: an explanation of these protocols under a common framework, a description of another fault tolerant protocol which is based on a modern coherence protocol widely used in commercial systems [1] and an evaluation and comparison of this and our two previous fault tolerant protocols to show how the overhead introduced by fault-tolerance varies depending on the base protocol.

The rest of this paper is organized as follows. Section 2.1 describes the base architecture which is being extended. Section 2.2 summarizes our previously proposed fault tolerant protocols while section 3 describes a new fault tolerant protocol. The evaluation is presented in section 4 and section 5 concludes.

2 Background

2.1 Base architecture

We assume single CMP systems built using a number of tiles [14]. Each tile contains a processor core, private L1 data and instruction caches, a bank of

the logically shared L2 cache and a network interface. The L2 cache is logically shared by all cores but it is physically distributed among the tiles. Each tile has its network interface which connects it to the on-chip IN. We assume in-order processors since that seems the most reasonable approach to build power-efficient CMPs with many cores. While we have assumed a tiled architecture and in-order processors, these choices are not constraints of the evaluated coherence protocols, whose functionality and correctness is not affected if out-of-order cores are used or a different arrangement is used instead of tiles.

We consider two base architectures: one using a token-based cache coherence protocol (TokenCMP) [10] and another one using a more traditional directory-based protocol (DirCMP). A third base architecture is described in section 3.

TokenCMP is a protocol based on token coherence which targets multiple CMPs and is well suited for single CMPs. Token coherence provides a framework for defining coherence protocols by separating the definition in a correctness substrate and a performance policy which define how the nodes exchange a fixed number of tokens among them. Most requests are *transient requests* which, in the case of *TokenCMP*, are broadcasted to all other nodes without ordering guarantees and without even a guarantee of being satisfied. *Token counting* rules ensure that coherency is maintained while *persistent requests* ensure forward progress by providing serialization when races between transient requests are detected. TokenCMP uses a performance policy similar to TokenB (*Token-using-broadcast*) with distributed arbitration for persistent requests.

DirCMP is a traditional MOESI-based directory cache coherence protocol [4] which uses an on-chip directory to maintain coherence between several private L1 caches and a shared non-inclusive L2 cache. It uses a directory cache in L2 and the L2 effectively acts as the directory for the L1 caches.

2.2 Our previous work

We have designed two fault-tolerant cache coherence protocols for CMPs based on two different approaches to cache coherence: token coherence and directory coherence. Both protocols have been shown to provide fault-tolerance with respect to transient faults in the IN with very little overhead. FtTokenCMP [5] is a token based coherence protocol which extends TokenCMP with fault tolerance, while FtDirCMP [6] is another fault-tolerant coherence protocol which is based in a more traditional directory protocol which we call DirCMP.

The fault tolerance measures of both protocols are similar in their intent and functionality and differ mostly in the implementation. In our experience, a fault tolerant cache coherence protocol needs to provide the following things: a fault detection mechanism, a fault recovery mechanism, and a mechanism to ensure that data is never lost or corrupted. Both protocols rely on error detection codes in messages to discard corrupted messages. It is assumed that the error detection code checks the whole message. Thus, from the point of view of the coherence protocols, a message can either arrive correctly or not arrive at all.

In both protocols, fault detection is achieved by means of a number of time-outs which detect deadlocks caused by discarded messages. This fault detection

mechanism is reliable and valid for every coherence protocol where a discarded message can be either harmless or lead to a deadlock in the same or a subsequent memory transaction. This is the case of TokenCMP, where discarded transient requests are harmless and the rest of message types lead to deadlock; and in the case of DirCMP where every discarded message leads to a deadlock. However, not all cache coherence protocols have this property: for example, some protocols do not require acknowledgments for invalidation messages, hence discarding an invalidation message would lead to an incoherence instead of a deadlock. Table 1 shows a summary of the timeouts used by each protocol.

Table 1. Timeouts summary for FtDirCMP and FtTokenCMP.

Timeout	When is it activated?	When is it deactivated?	Action when it triggers?
FtDirCMP			
Lost Request	When a request is issued.	When the request is satisfied.	The request is reissued with a new serial number.
Lost Unblock	When a request is answered.	When the unblock message is received.	An *UnblockPing* is sent.
FtTokenCMP			
Lost Token	When a persistent request becomes active.	When the persistent request is deactivated.	Request a token recreation.
Lost Persistent Deactivation	When an persistent request is activated.	When the persistent request is deactivated.	Send a persistent request ping.
Both protocols			
Lost Data	When a backup state is entered.	When the Ownership Acknowledgement arrives.	Issue an *OwnershipPing* / Request a token recreation.
Lost Backup Deletion Ack.	When a line enters the blocked state.	When the Backup Deletion Acknowledgement arrives.	Reissue the *AckO* / Request a token recreation.

Also, both protocols use essentially the same mechanism to avoid data loss, ensuring reliable transmission of owned data by means of exchanging a pair of acknowledgments. The mechanism works as follows: when a cache sends owned data to another cache, it keeps a backup copy of it. This backup copy may be used by the respective recovery mechanism if necessary, but it cannot be used by the cache for any other purpose. The backup will be kept until an *ownership acknowledgment* sent by the receiver arrives. On the other hand, the cache which receives the data can use it as soon as it arrives, but it cannot send it to another cache until it receives a *backup deletion acknowledgment* sent by the previous owner once its backup has been discarded. The last restriction is necessary to ensure that there is no more than one backup copy of each cache line, because otherwise fault recovery would be significantly more complex.

These acknowledgments are sent out of the critical path of cache misses so they do not directly affect the execution time of programs. Also, in many cases the acknowledgments are piggybacked in other messages of the same coherence transaction. However, the increased network traffic caused by this mechanism is the main overhead incurred by the fault tolerant measures of both protocols.

The fault recovery mechanism is different for each protocol. In FtTokenCMP, fault recovery is achieved by means of a centralized mechanism called the *token recreation process* arbitrated by the memory controller. This process works as long as there is a valid copy of data in some cache or one and only one backup

copy (which is guaranteed by the owned data transmission mechanism described above). The memory controller attends token recreation requests in FIFO order to avoid livelock and it works sending messages to every cache asking it to invalidate all tokens and send back to memory any data that it may have. Once the memory receives the data or invalidation acknowledgments from every cache, it sends it to the cache which requested the recovery with a new set of tokens.

To avoid creating an incoherence due to stale response messages still traveling through the IN after a *token recreation*, all coherence responses are tagged with a *token serial number* (TSN) which is increased during the token recreation process. Messages with a wrong TSN are discarded when received by any node. Token serial numbers are stored in every node in a dedicated structure (the *TSN table*), but only for those cache lines which have a serial number different than 0. We have found that having a very small number of entries of only a few bits each is enough for good results. When all entries are used, one of them is evicted setting its serial number to 0 by means of the *token recreation process*.

In contrast, FtDirCMP achieves fault recovery reissuing requests with a different *request serial number*. FtDirCMP does not need an specific serialization point for fault recovery since the directory (or the on-chip L2 directory cache) acts already as the serialization point for all requests. These reissued requests need to be identified as such by the node that answers to them and not be treated like an ordinary request. In particular, a reissued request should not wait in the incoming request buffer to be attended by the L2 or the memory controller until a previous request is satisfied, because that previous request may be precisely the older instance of the request that is being reissued in case of a false positive.

Since stale responses to a few reissued request messages may lead to an incoherence in FtDirCMP, we use *request serial numbers* to discard responses which arrive too late (when the request has already been reissued). Every message carries a serial number. Request serial numbers are chosen by the cache that issues the request while responses or forwarded requests will carry that of the request that they are answering to. When a request is reissued, it will be assigned a new serial number which will allow to distinguish between responses to the old request and to the new one. Nodes must remember the serial number of the requests that they are currently handling and discard any message which arrives with an unexpected serial number or from an unexpected sender. This information needs to be updated when a reissued request arrives.

In some cases, both protocols achieve deadlock recovery issuing ping messages when a timeout triggers to force the reissue of a message which is expected to finish a coherence transaction, like an *Unblock* message in case of FtDirCMP or a *Persistent Request Deactivation* message in case of FtTokenCMP.

The *token serial numbers* used in FtTokenCMP serve a similar purpose to *request serial numbers* used in FtDirCMP (e.g.: being able to discard stale messages after fault recovery which could cause an incoherence), but the latter are easier to implement and more scalable. *Token serial numbers* are associated with each cache line and need to be updated in a coordinated fashion during the *token recreation process*. Hence, they required an additional structure in each cache to

store them (only for those hopefully few lines that had a token serial number different than 0, but even for lines which were not currently in any cache). On the other hand, *request serial numbers* are associated with individual requests and so they are short-lived information which can be stored in the MSHR. However, *token serial numbers* do not need to be carried in request messages (only in responses) while *request serial numbers* are sent with every request and need to be propagated with every message which is sent as consequence of the request.

Notice that discarding any message in FtDirCMP or FtTokenCMP is always safe (even if it could be not strictly necessary in some cases) since the protocol already has provisions for lost messages of any type.

3 A new broadcast-based cache coherence protocol

No real system has been implemented yet using a coherence protocol based on the token framework. Also, many cache coherence protocols which are used in widely used systems cannot be precisely categorized as snoopy-based nor directory-based. AMD Hammer [1] is one of these protocols. It targets systems with a small number of processors using a tightly-coupled point-to-point unordered IN.

In this work, we have implemented HammerCMP which is an adaptation of AMD Hammer protocol to the tiled CMP environment and we have used it as a base for FtHammerCMP, a new fault tolerant protocol for small scale CMPs.

Like DirCMP, HammerCMP sends requests to a home L2 bank which acts as the serialization point for requests to its cache lines. There is no directory information, and all requests are forwarded using broadcast to all other caches. All of them answer to the forwarded requests sending either an acknowledgment or a data message to the requestor. When the requestor receives all the acknowledgments informs to the home L2 controller that the miss has been satisfied.

HammerCMP avoids the overhead of directory information and the latency of accessing the directory structure at the cost of much more IN traffic. Also, all processors need to intervene in all misses, like in a snoopy protocol.

Using the principles described in section 2.2, FtHammerCMP adds fault tolerance measures to HammerCMP as the ones described for FtDirCMP. It uses the same set of timeouts for detecting faults and reissues requests using different request serial numbers in a way very similar to FtDirCMP for recovering from faults. Reliable owned data transference is done using the same pair of acknowledgments as the other two protocols.

4 Evaluation

4.1 Methodology

We have performed full system simulations of a mix of scientific applications with fault injection with the aims of determining adequate values for some protocol parameters, assess the fault tolerance capability of each protocol and measure the overhead introduced by the fault tolerance measures. For this, we have used

a custom version of Multifacet GEMS [9] detailed memory model and Virtutech Simics [8]. Every simulation has been performed several times using different random seeds to account for the variability of multithreaded execution, this is represented by the error bars in the figures which enclose the resulting 95% confidence interval of the results. We have simulated tiled CMP systems as described in section 2.1. Table 2(a) shows the most relevant parameters of the systems.

Table 2. Characteristics of simulated architectures and input sizes used for benchmarks in the simulations.

(a) System characteristics

16-Way Tiled CMP System	
Processor speed	2 GHz
Cache parameters	
Cache line size	64 bytes
L1 cache:	
Size, associativity	32 KB, 4 ways
Hit time	2 cycles
Shared L2 cache:	
Size, associativity	1024 KB, 4 ways
Hit time	15 cycles
Memory parameters	
Memory access time	300 cycles
Memory interleaving	4-way
Network parameters	
Topology	2D Mesh
Non-data message size	8 bytes
Data message size	72 bytes
Channel bandwidth	64 GB/s

(b) Input sizes

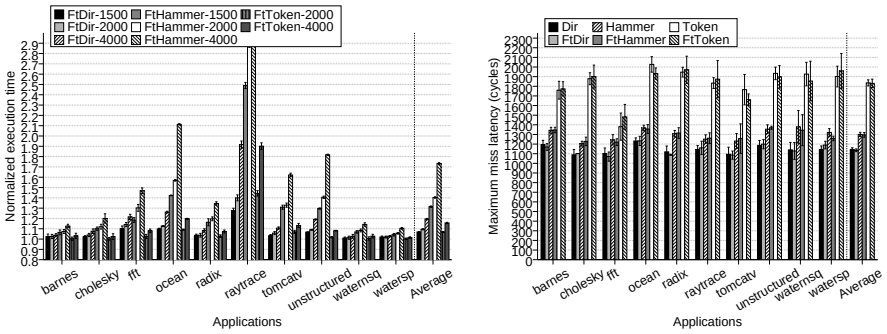
Benchmark	**Input Size**
Barnes	8192 bodies, 4 time steps
Cholesky	tk16.O
FFT	256K complex doubles
Ocean	258×258 ocean
Radix	1M keys, 1024 radix
Raytrace	10Mb, teapot.env scene
Tomcatv	256 points, 5 iterations
Unstructured	Mesh.2K, 5 time steps
Water-NSQ	512 molecules, 4 time steps
Water-SP	512 molecules,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

Finally, we have used a selection of scientific applications for the evaluation: 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 experimental results reported here correspond to the parallel phase of each program only. Problem sizes are shown in table 2(b).

4.2 Adjusting the fault detection timeouts

All fault tolerant protocols achieve fault detection by means of a number of timeouts. Each protocol requires up to four timeouts which are active at different places and times during a memory transaction or cache replacement. The value of these timeouts determine the latency of fault detection, hence shorter values help to achieve lesser performance degradation in presence of faults since fault recovery will start earlier. For example, for the three fault tolerant protocols considered in this work figure 1(a) shows how the execution time increases with the value of these timeouts under a fixed fault rate.

Since false positives occur when a timeout triggers before a miss has had enough time to be satisfied, to avoid false positives the timeout values should



(a) Execution time varying timeouts

(b) Maximum miss latency

Fig. 1. Relative execution time with respect to DirCMP without faults for each fault tolerant protocol with 250 corrupted messages per million using different values for the fault detection timeouts and maximum miss latency (in cycles) of each coherence protocol without faults..

be large enough to allow every memory transaction to finish, assuming that no fault occurs. Figure 1(b) shows the measured maximum latency in CPU cycles of each protocol when no faults occur and disabling all the timeouts.

Looking at figure 1(b), we can see that the maximum latency of the fault-tolerant protocols is almost exactly the same than that of their corresponding non fault-tolerant counterpart. This is expected, since the behavior of the fault-tolerant protocols when no timeout triggers is almost the same than that of the non fault-tolerant ones, except for the ownership acknowledgments which are sent out of the critical path of cache misses.

This latency is around 1250 cycles for the FtDirCMP protocol, 1350 for the FtHammerCMP protocol and 2000 for the FtTokenCMP protocol. Hence, we can choose any value greater than those for the timeouts to avoid having any false positive for these workloads. Using shorter values is still possible but would increase the number of false positives and could degrade performance and increase network traffic due to the retried requests or token recreation requests. However, if the chosen values are too low (lower than the time required to finish a transaction), the recovery mechanism would be invoked too frequently preventing forward progress.

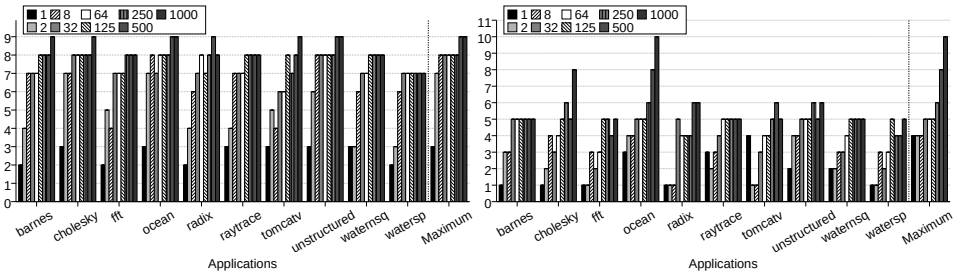
Finally, we have considered using different values for each of the four timeouts of each protocol, but our experiments do not show any significant advantage in doing so.

We have chosen a value of 2000 cycles for all timeouts in the FtTokenCMP protocol and 1500 cycles in the FtDirCMP and FtHammerCMP protocols. These values are large enough to avoid false positives in every case and, as shown below, achieve very low performance degradation when faults actually occur. Making this value smaller achieves very little benefit while significantly increasing the risk of false positives.

4.3 Effect of the request serial number size in fault-tolerance

In the case of FtDirCMP and FtHammerCMP, the ability to correctly recover from faults depends on the number of bits used for encoding the request serial number which is used to discard stale responses to reissued requests (for example, to discard old acknowledgments to reissued invalidation messages which could lead to incoherence). This number should be as low as possible to reduce overhead in terms of increased message size and hardware resources to store it while being sufficient to ensure that when a request is reissued (even several times in a row) every response to the old request is discarded. Since the number of reissued messages increases as the fault rate increases, the number of bits used to encode request serial numbers determines the maximum fault rate supported.

To measure this, we have performed simulations of FtDirCMP using a wide variety of fault rates. We have used 32-bit request serial numbers for those simulations but we have recorded how many lower order bits were required to distinguish all the request serial numbers that needed to be compared. For doing this, every time that two request serial numbers are compared, we record the position of the least significant bit which is different in both numbers. Then, we assume that the maximum of all these measures is an upper bound of the number of bits required to ensure correctness for each fault rate. These results are shown in figure 2.



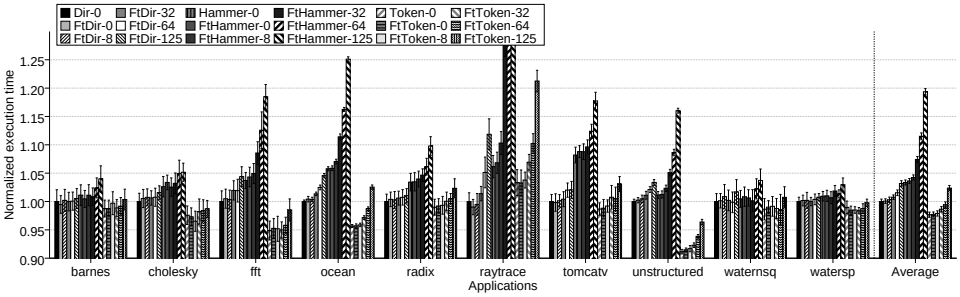
(a) FtDirCMP (b) FtHammerCMP

Fig. 2. Required RSN bits to discard every old response to a reissued message.

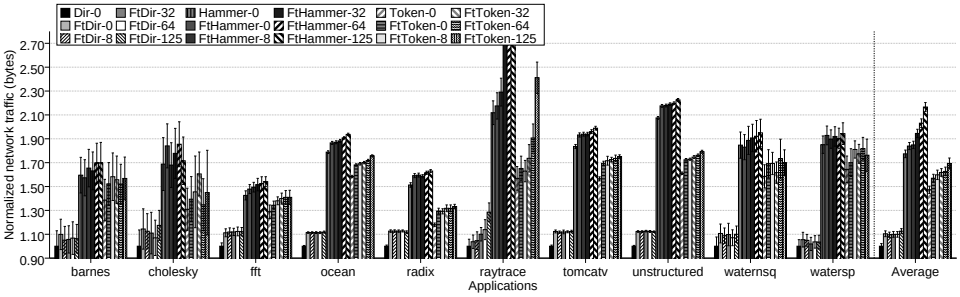
As it can be seen, when using the FtDirCMP protocol 9 bits are enough for all the tested fault rates and 8 bits suffice for fault rates up to 250 corrupted messages per million. In case of using the FtHammerCMP protocol, 8 bits provide fault tolerance up to 500 corrupted messages per million, while 10 bits are required for the maximum tested fault rate, 1000 corrupted messages per million. Hence, we have chosen to use 8 bits to encode the request serial numbers in the rest of our experiments for both protocols which is enough to achieve fault tolerance up to 250 corrupted messages per million, which is already an unrealistic and unreasonably high failure rate.

4.4 Execution time overhead

We have measured the execution time of each one of the fault-tolerant protocols using the fault tolerance parameters determined above with several message loss rates and compared it to the execution time of the non fault tolerant protocols in a fault-free scenario. The results are shown in figure 3(a). Fault rates are expressed in number of messages discarded per million of messages that travel trough the network and all results are normalized with respect to the execution time of the DirCMP protocol.



(a) Execution time overhead



(b) Network overhead (bytes)

Fig. 3. Execution time and network overhead of each protocol for several fault rates.

We can see that the run-time overhead of each fault-tolerant protocol when compared to its non fault-tolerant counterpart in a fault-free scenario is not measurable. This is consistent with the fact that, when no faults occur, the only difference in the behavior of the fault-tolerant protocols with respect to the non fault-tolerant ones is just the extra acknowledgments used to ensure reliable owned data transmission, which are sent out of the critical path of misses.

For these workloads, both FtTokenCMP and FtDirCMP achieve very similar execution times when no faults occur (less than 3% difference on average). FtHammerCMP execution time difference is less than 4% higher than FtDirCMP also, and 6% higher than FtTokenCMP.

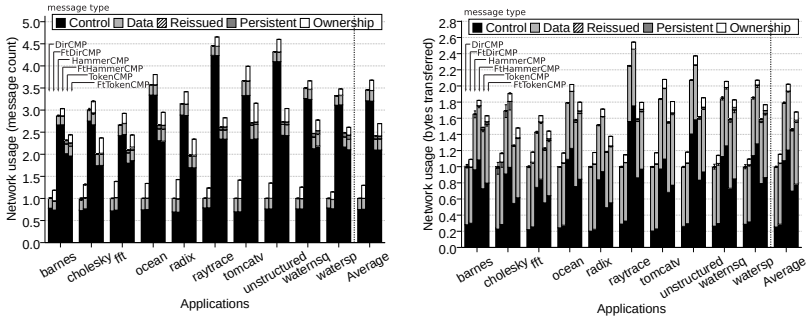
As the fault rate increases, so does the execution time of each protocol. In the case of FtDirCMP and FtTokenCMP, the average performance degradation is almost unmeasurable until the message loss rate reaches 32 corrupted messages per million. However, even when the fault rate reaches 64 corrupted messages per million, the execution time of FtTokenCMP is lower than the execution time of the directory protocol in a fault free scenario.

On the other hand, HammerCMP and FtHammerCMP without faults have the same performance than FtDirCMP under a fault rate of 125 messages lost per million. Also FtHammerCMP performance degradation starts being significant under a fault rate of 8 messages corrupted per million and the rate at which it increases is noticeably worse than in the case of the other protocols. This is due to the much higher network traffic of HammerCMP and FtHammerCMP in comparison to all the other protocols even in absence of faults as can be seen in figure 3(b).

4.5 Network overhead

In absence of faults, the most important difference in the behavior of our protocols with respect to their non fault-tolerant counterparts is the exchange of acknowledgments to ensure that owned data is transferred safely and avoid data loss. Although they are sent out of the critical path of cache misses so that they do not have effect in the miss latency, these acknowledgments introduce additional network traffic which is the main cost of the fault tolerance measures.

We have measured the network overhead of our proposal in terms of the relative increase in the number of messages and the number of bytes transmitted through the network. We have increased one byte the message sizes of the fault tolerant protocols with respect to the non fault-tolerant ones to accommodate the *request serial numbers* and *token serial numbers*. This means 1.14% increase in size for data messages and 12.5% increase for control messages. The results of these measurements are shown in figure 4.



(a) Number of messages (b) Bytes transmitted

Fig. 4. Network overhead of the fault tolerant protocols.

We can see that, in terms of message traffic, the overhead of the fault-tolerant protocols comes entirely from the acknowledgments used to ensure reliable data transmission ("*Ownership*" part of each bar). This overhead is less than 30% for all our fault-tolerant protocols. Moreover, the overhead drops considerably when it is measured in terms of bytes, even considering that every message is one byte longer in the fault-tolerant protocols.

Figure 3(b) shows the network overhead under several fault rates. The network traffic increases slowly with the fault rate due to the reissued messages or the token recreation messages. In the case of FtDirCMP, the increase is almost unmeasurable for the fault rates shown. However, as can be seen for FtHammerCMP, once the network traffic reaches certain point (around 1.9 in our plot), the slope becomes steeper. This is due to the fact that the capacity of the network is exceeded and this increases the average latency which in turn causes a number of false positives which lead to more reissues which further increase the network traffic and consequently the execution time. Hence, network capacity can become a limiting factor for the fault tolerance of our protocols.

4.6 Hardware requirements

The *token serial number table* is implemented with a small associative table at each cache and at the memory controller to store those serial numbers whose value is not zero. Using two bits to encode the serial number and 16 entries at each node is enough for supporting the fault rates used in this paper. If the tokens of any line need to be recreated more than 4 times the counter wraps to zero (effectively freeing an entry in the table) and if more than 16 different lines need to be stored in the table, the least recently modified line is evicted by means of using the token recreation process to set its serial number to zero.

On the other hand, *request serial numbers* do not need to be kept once the memory transaction has been completed. They can be stored in the MSHR or (optionally) in a small associative structure in cases where a full MSHR is not needed. As shown in section 4.3, using 8 bits to encode request serial numbers is enough to achieve tolerance to very high fault rates, and even less bits are required to support more realistic but still very high fault rates.

Also, to be able to detect reissued requests in FtDirCMP and FtHammerCMP, the identity of the requester currently being serviced by the L2 or the memory controller needs to be recorded, as well as the identity of the receiver of owned data when transferring ownership from one L1 cache to another to be able to detect reissued forwarded requests.

The timeouts used for fault detection require the addition of counters to the MSHRs or a separate pool of timeout counters. Although there are up to four different timeouts involved in any coherence transaction, no more than one counter is required at any time in the same node for a single coherence transaction. In the case of FtTokenCMP, all but one timeout can be implemented using the same hardware already used to implement the starvation timeout required by token protocols. Also, our fault-tolerant protocols require one extra virtual channel than their non fault-tolerant counterparts.

Finally, a less important source of overhead is the increased pressure in caches and writeback buffers because of the blocked ownership and backup states and the effect of the reliable ownership transference mechanism in replacements. When a backup buffer or a writeback buffer is used, we have not been able to detect any effect in the execution time due to these reasons. The size of the writeback buffer may need to be increased, but our previous work [5] shows that one extra entry would be enough to avoid any slowdown.

5 Conclusions

We propose implementing fault tolerance measures at the cache coherence protocol level to deal with transient faults in the interconnection network of CMPs and provide several cache coherence protocols which can ensure the correct execution of parallel programs using a non reliable on-chip IN.

In this work, we have presented a new fault tolerant protocol based on AMD Hammer protocol which could be useful for small scale CMPs. We have thoroughly compared and evaluated the performance of two previously presented fault tolerant cache coherence protocols and the new one. We have shown that the overhead imposed in the execution time due to the fault tolerant measures is negligible. Further, we have shown that the performance impact of moderate fault rates in the IN is insignificant when using our protocols.

We have explained how to tune the fault tolerance parameters of the protocols to achieve the desired level of fault tolerance, performance degradation in presence of faults and overhead in absence of faults. We have shown that, even for fault rates which are unrealistically high, the hardware overhead of our proposals is low. The main cost of our fault tolerance measures is a moderate increase in network traffic, but this increase is much lower that the difference in network usage between protocols, specially considering currently used protocols like AMD Hammer.

Our evaluation shows that a token coherence based protocol can provide sightly better performance than a directory based one even when the token based protocol is subjected to higher fault rates, but at the cost of much higher network usage. We have found that the network usage of our protocols increases with the fault rate and hence network capacity can be a limiting factor for fault tolerance. Due to the efficient network usage of directory-based protocols and the small difference in performance with respect to the other two fault tolerant protocols shown in our evaluation, we think that FtDirCMP is a good cache coherence protocol for large scale tiled CMPs.

Acknowledgements

This work has been jointly supported by the Spanish MEC and European Comission 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. A. Ahmed, P. Conway, B. Hughes, and F. Weber. AMD Opteron^{TM} Shared-Memory MP Systems. In *14th HotChips Symp.*, August 2002.
2. L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proc. of 27th Int'l Symp. on Computer Architecture (ISCA'00)*, pages 282–293, June 2000.
3. K. Constantinides, S. Plaza, J. Blome, B. Zhang, V. Bertacco, S. Mahlke, Todd Austin, and M. Orshansky. BulletProof: a defect-tolerant CMP switch architecture. In *12th Int'l Symp. on High-Performance Computer Architecture (HPCA'06)*, pages 3–14, February 2006.
4. D. J. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.
5. Ricardo Fernández-Pascual, José M. García, Manuel E. Acacio, and José Duato. A low overhead fault tolerant coherence protocol for CMP architectures. In *13th Int'l Symposium on High-Performance Computer Architecture (HPCA'07)*, pages 157–168, February 2007.
6. Ricardo Fernández-Pascual, José M. García, Manuel E. Acacio, and José Duato. A fault-tolerant directory-based cache coherence protocol for shared-memory architectures. In *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2008)*, June 2008.
7. Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE MICRO Magazine*, 20(2):71–84, March-April 2000.
8. Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.
9. Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33(4):92–99, September 2005.
10. Michael R. Marty, Jesse D. Bingham, Mark D. Hill, Alan J. Hu, Milo M. K. Martin, and David A. Wood. Improving multiple-CMP systems using token coherence. In *11th Int'l Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 328–339. IEEE Computer Society, February 2005.
11. S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The soft error problem: An architectural perspective. In *11th Int'l Symposium on High-Performance Computer Architecture (HPCA'05)*, February 2005.
12. Srinivasan Murali, Theocharis Theocharides, N. Vijaykrishnan, Mary Jane Irwin, Luca Benini, and Giovanni De Micheli. Analysis of error recovery schemes for networks on chips. *IEEE Design and Test of Computers*, 22(5):434–442, 2005.
13. D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C.R. Das. Exploring fault-tolerant network-on-chip architectures. In *Procs. of the 2006 Int'l Conf. on Dependable Systems and Networks (DSN'06)*, pages 93–104, 2006.
14. M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, Jae-Wook Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, May 2002.