A fault-tolerant directory-based cache coherence protocol for CMP architectures

Ricardo Fernández-Pascual, José M. García, Manuel E. Acacio and José Duato[†]
Universidad de Murcia, Spain. E-mail: {rfernandez,jmgarcia,meacacio}@ditec.um.es
[†]Universidad Politécnica de Valencia, Spain. E-mail: jduato@gap.upv.es

Abstract

Current technology trends of increased scale of integration are pushing CMOS technology into the deep-submicron domain, enabling the creation of chips with a significantly greater number of transistors but also more prone to transient failures. Hence, computer architects will have to consider reliability as a prime concern for future chip-multiprocessor designs (CMPs). Since the interconnection network of future CMPs will use a significant portion of the chip real state, it will be especially affected by transient failures. We propose to deal with this kind of failures at the level of the cache coherence protocol instead of ensuring the reliability of the network itself. Particularly, we have extended a directory-based cache coherence protocol to ensure correct program semantics even in presence of transient failures in the interconnection network. Additionally, we show that our proposal has virtually no impact on execution time with respect to a non fault-tolerant protocol, and just entails modest hardware and network traffic overhead.

1. Introduction

Recent technology improvements have made possible to put more than a billion transistors in a single chip. To date, the best way to use this sheer number of transistors seems to be implementing several processor cores and increasing amounts of cache memory in a single chip. Compared to other options, Chip Multiprocessors (CMPs) [2, 8] offer a way to utilize these resources to increase performance in an energy-efficient way while keeping complexity manageable by means of exploiting thread-level parallelism. There are already several commercial CMP systems, most of them based on the well-known shared-memory paradigm and with varying implementations of the cache coherence protocol.

Also, tiled architectures which are built by replicating several *tiles* comprised by a core, private cache, part of the shared cache and a network interface further help in keeping complexity manageable, scale well to a larger number of cores and support families of products with varying number of tiles. In this way, it seems likely that they will be the choice for future many-core CMP designs [12, 21, 23].

Tiled CMPs implement a point-to-point interconnection network which is best suited for directory-based cache coherence protocols. Furthermore, compared with snoopy based or token-based [10] protocols which require frequent broadcasts, directory-based ones are more scalable and energy-efficient.

On the other hand, the reliability of electronic components is never perfect. Components are subject to several types of failures, which can be either permanent, intermittent or transient. Transient failures [15], also known as soft errors or single event upsets, 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, electromagnetic interference (EMI) or radiation from lightning.

In many applications high availability and reliability are critical requirements. Even for commodity systems, reliability needs to be above a certain level for the system to be useful for anything. However, the same technology trends of increased scale of integration which make CMPs possible will make transient failures more common [3]. Also, the lower voltages used for power-efficiency reasons make transient failures even more frequent. Transient failures are already a problem for memories (and caches) which routinely use error detection and correction codes (ECC) to deal with them. Other parts of the system will need to use fault tolerance techniques to deal with transient failures as their frequency increases.

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. Hence, transient failures will have to be handled across all the levels of the system to avoid actual errors.

One of the components which will be affected by transient failures in a CMP is the interconnection network. The interconnection network 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. Communication is usually very fine-grained (at the level of cache lines) and requires very small and frequent messages. Hence, to achieve good performance the interconnection network must provide very low latency and should avoid acknowledgment messages and other flow-control messages as much as possible.

Differently from other authors, we propose to deal with transient failures in the interconnection network of CMPs at the level of the cache coherence protocol. In a previous work [7], we showed that a token-based coherence protocol can be extended to tolerate transient failures. In this work, we apply some of the lessons learned there to guarantee fault tolerance in widely used directory-based protocols.

In our failure model we assume that the interconnection network will either deliver a message correctly or not at all. This can be achieved by means of using an error detection code (CRC) in each message and discarding corrupted messages upon arrival. We also assume that caches and memories are protected by means of ECC.

Our cache coherence protocol extends a standard directory-based coherence protocol with fault tolerant measures: the integrity of data when cache lines travel through the network is ensured by means of explicit acknowledgments out of the critical path of cache misses, a number of timeouts to detect faults are added alongside with the ability to reissue some requests to avoid deadlocks due to transient faults, and request serial numbers are added to ensure correctness when requests are reissued.

Our proposal does not add any requirement to the interconnection network so it is applicable to current and future designs. Moreover, most fault tolerance proposals require some kind of checkpointing and rollback while ours does not. Our proposal could be used in conjunction to other techniques which provide fault tolerance to individual cores and caches in the CMP to achieve full coverage against transient failures inside the chip.

Although the cache coherence protocol is critical to the performance of parallel applications, we show that the fault tolerance measures introduced in our protocol add minimal overhead in terms of execution time. The main cost of our proposal is a slight increase in network traffic due to some extra acknowledgments

The rest of the paper is organized as follows. The base architecture and cache coherence protocol are described in section 2. Section 3 explains the fault tolerance measures added to the coherence protocol. A performance evaluation of the new protocol is done in section 4. In section 5 we review previous work relevant to this paper. Finally, section 6 concludes the paper.

2. Base architecture and directory protocol

In this work, we assume a CMP system built using a number of tiles [21]. Each tile contains a processor, private L1 data and instruction caches, a bank of the L2 cache, and a network interface. The L2 cache is logically shared by all cores but it is physically distributed among all tiles. Each tile has its network interface to connect to the on-chip interconnection network. We assume in-order processors since that seems the most reasonable approach to build power-efficient CMPs with many cores, although the correctness of the protocol is not affected if out-of-order cores are used.

Our base architecture uses a traditional directory protocol adapted for CMP systems that we will refer to as DIRCMP. DIRCMP is a MOESI-based cache coherence protocol which uses an on-chip directory to mantain 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.

DIRCMP uses per line busy states to defer requests to lines with outstanding requests. Hence, the directory will attend only one request for each line at the same time. Also, it uses three-phase writebacks to coordinate writebacks with other requests. It includes a migratory sharing optimization to accelerate read-modify-write sharing behavior. Table 1 shows a simplified list of the main types of messages used by DIRCMP and a short explanation of their main function.

We assume a point-to-point ordered network for our base architecture (in particular, the 2D-mesh with dimension-ordered routing typically employed in tiled CMPs) and take advantage of this fact to simplify the design of our fault-tolerant protocol. Directory protocols used in most cc-NUMAs usually assume that the network is point-to-point unordered. That is, two messages sent from a node to another can arrive in a different order than they were sent. This assumption makes possible using any routing technique, including adaptive ones. In CMPs, adaptive routing may be useful in some situations. Fortunately, our protocol can be easily

Table 1. Message types used by DIRCMP.

Type	Description
GetX	Request data and permission to write.
GetS	Request data and permission to read.
Put	Sent by the L1 to initiate a write-back.
WbAck	Sent by the L2 to let the L1 actually perform the write-back.
Inv	Invalidation request sent to invalidate sharers before granting exclusive access.
Ack	Invalidation acknowledgment.
Data	Message carrying data and read permission.
DataEx	Message carrying data and write permission.
Unblock	Informs the L2 that the data has been received and the sender is now a sharer.
UnblockEx	Informs the L2 that the data has been received and the sender has now exclusive access to the line.
WbData	Write-back containing data.
WbNoData	Write-back containing no data.

extended to support unordered point-to-point networks (see [6] for details).

3. A fault tolerant directory coherence protocol

From now on, we consider a CMP system whose interconnection network is not reliable due to the potential presence of transient errors. We assume that these errors cause the loss of messages (either an isolated one or a burst of them) since they directly disappear from the interconnection network or arrive to their destination corrupted and are discarded.

Losing a message in DIRCMP will always lead to a deadlock situation, since either the sender will be waiting indefinitely for a response or the receiver was already waiting for the lost response. Additionally, losing a message carrying data can lead to loss of data if the corresponding memory line is not in any other cache and it has been modified since the last time that it was written to memory. Notice that losing any message cannot lead to an incoherence, since write access to a line is only granted after all the necessary invalidation acknowledgments have been actually received.

FTDIRCMP is an extension of DIRCMP which assumes an unreliable interconnection network. It will guarantee the correct execution of a program even if coherence messages are lost or discarded by the interconnection network due to transient errors.

FTDIRCMP uses extra messages to acknowledge the reception of a few critical data messages and to detect faults. When possible, those messages are kept out of the critical path of any cache miss and they are piggybacked in other messages in the most frequent cases. Table 2 shows the message types that are added by FTDIRCMP to those mentioned in table 1.

Thanks to the fact that every message lost in DIRCMP leads to a deadlock, FTDIRCMP can use

Table 2. New message types for FTDIRCMP.

Type	Description
AckO	Ownership acknowledgment.
AckBD	Backup deletion acknowledgment.
UnblockPing	Requests confirmation whether a cache miss is still in progress.
WbPing	Requests confirmation whether a writeback is still in progress.
WbCancel	Confirms that a previous writeback has already finished.
OwnershipPing	Requests confirmation of ownership.
NackO	Not ownership acknowledgment.

timeouts to detect potentially lost messages. FTDIRCMP uses a number of timeouts to detect faults and start corrective measures. Table 3 shows a summary of these timeouts.

Usually, when a fault occurs and a timeout triggers, FTDIRCMP reissues the request using a different serial number. The need for request serial numbers is explained in section 3.5. These reissued requests need to be identified as such by the node that answers to them and not be treated like an usual 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. Hence, the L2 directory needs to remember the blocker (last requester) of each line to be able to detect reissued requests. This information can be stored in the Miss Status Holding Register (MSHR) table or in a dedicated structure for the cases when it is not necessary to allocate a full MSHR entry.

3.1. Reliable data transmission

A fault tolerant cache coherence protocol needs to ensure that there is always at least one updated copy of the data of each line off the network and that such copy can be readily used for recovery in case of a fault that corrupts the data while it travels through the network.

There is always one owner node[1] for each line which is responsible of sending data to other nodes to satisfy read or write requests or to perform writeback when the data is modified.

Data transmission needs to be reliable when ownership is transferred. Ownership can be transferred either with an exclusive data response or a writeback response. On the other hand, when ownership is not being transferred, data transmission does not need to be reliable because if the data carrying message is lost, the data can be sent again from the owner node when the request is

[1]From the point of view of the coherence protocol, a node can be either an L1 cache, an L2 cache bank or a memory bank.

Table 3. Timeouts summary.

Timeout	When is it activated?	Where is it activated?	When is it deactivated?	What happens when it triggers?
Lost Request	When a request is issued.	At the requesting L1 cache.	When the request is satisfied.	The request is reissued with a new serial number.
Lost Unblock	When a request is answered (even writeback requests).	At the responding L2 or memory.	When the unblock (or writeback) message is received.	An *UnblockPing*/*WbPing* is sent to the cache that should have sent the *Unblock* or writeback.
Lost backup deletion acknowledgment	When the *AckO* message is sent.	At the node that sends the *AckO*.	When the *AckBD* message is received.	The *AckO* is reissued with a new serial number.

reissued.

In order to ensure reliable data transmission of owned data, FtDirCMP adds some additional states to the usual set of MOESI states:

- **Backup (B)**: This state is similar to the Invalid (I) state, but the data is kept in the cache to be used for potential recovery (that is, when leaving the Modified, Owned or Exclusive states) and will abandon it once an *ownership acknowledgment* is received.
- **Blocked ownership (Mb, Eb and Ob)**: To prevent having more than one backup for a line at any given point in time, which is important to be able to recover in case of a fault, a cache that acquires ownership (entering the Modified, Owned or Exclusive states) will avoid transmitting the ownership to another cache until it receives a *backup deletion acknowledgment* message from the previous owner. For achieving this, we have added blocked versions of the Modified, Exclusive and Owned states. While a line is in one of these states, the cache will not attend external requests to that line which require ownership transference.

Using the states described above, the transmission of owned data between two nodes works as follows:

1. When a node sends owned data to another node, it does not transition to an *Invalid* state. Instead, it enters a *Backup* state in which the data is still kept for recovery, although no read or write permission on the line is retained. Depending on the particular case, the data may be kept in the same cache block, in a backup buffer [7] or in a writeback buffer. The cache will keep the data until it receives an *ownership acknowledgment*, which can be received as a message by itself or piggybacked along with an *UnblockEx* message.
2. When the data message is received by the new owner, it sends an *ownership acknowledgment* to the node that sent the data. Also, it does not transition to an M, O or E state. Instead it enters one of the blocked ownership states (Mb, Eb or Ob) until it receives the *backup deletion acknowledgment*. While in these states, the node will not transfer ownership to another node. This ensures that there is never more than one backup copy of the data. However, at this point the node has received the data (and possibly write permission to it) and the miss is already satisfied. The *ownership acknowledgment* will also carry a serial number, which can be the same than the data carrying message just received.
3. When the node that sent the data receives the *ownership acknowledgment*, it transitions to an *Invalid* state and sends a *backup deletion acknowledgment* to the other node with the same serial number as the received ownership acknowledgment.
4. Finally, once the *backup deletion acknowledgment* is received, the node that received the data transitions to an M, O or E state and can now transfer the ownership to another node if necessary.

Figure 1 shows an example of how a cache-to-cache miss which requires ownership change is handled in FtDirCMP and compares it with DirCMP.

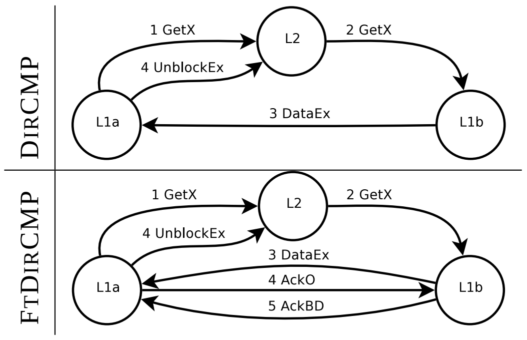
The ownership acknowledgment can be piggybacked in the *UnblockEx* message when the data is sent to the requesting L1 by the L2 (or to L2 by the memory). In that case, only an extra message (the backup deletion acknowledgment) needs to be sent.

These rules ensure that for every cache line there is always either an owner node that has the data, a backup node which has a backup copy of the data or both. They also ensure that there is never more than one owner or one backup node.

3.1.1 Optimizing ownership transference from memory to L1 caches

The rules explained above ensure the reliable transmission of owned data in all cases without adding any message to the critical path of cache misses in most cases. However there are potential performance problems created by the blocked ownership states, since a node cannot transfer the recently received owned data until the *backup deletion acknowledgment* message is received.

This is not a problem when the data is received by an L1 cache since the node can already use the data while it waits for said acknowledgment. However, in the case of L2 misses, the L2 cannot answer the L1 request imme-



Initially, for both protocols, L1b has the data in modifiable (M), exclusive (E) or owned[2] (O) state and L1a requests write access to L2 (1) which forwards the request to L1b (2). In DIRCMP, L1b sends the data to L1a (3) and transitions to invalid state. Subsequently, when L1a receives the data, it transitions to a modifiable (M) state and sends an *UnblockEx* message to L2. In FTDIRCMP, when L1b receives the forwarded *GetX*, it sends the data to L1a and transitions to the backup state (3). When L1a receives the data, it transitions to the blocked ownership and modifiable (Mb) state and sends the *UnblockEx* message to L2 and an *AckO* message to L1b (4). When L1b receives the *AckO*, it discards the backup data, transitions to invalid (I) state and sends a *AckBD* message to L1a (5), which transitions to the usual modifiable (M) state when receives it.

Figure 1. Cache-to-cache write miss.

diately after receiving the data from memory because, according to the rules described above, it first needs to send an *ownership acknowledgment* to memory and wait for the *backup deletion acknowledgment*. Hence, in the case of L2 misses, these rules would add two messages in the critical path of misses.

To avoid increasing the latency of L2 misses, we relax the rules in these cases. We allow the L2 to send the data directly to the requesting L1 just after receiving it, keeping a backup until it receives the ownership acknowledgment from the L1. In fact, the L2 does not send the ownership acknowledgment to memory until it receives it from the L1 (most times piggybacked on an unblock message) since this way we can piggyback it with an *UnblockEx* message.

To implement this behavior, we modify the set of states for the L2 cache so that a line can be either internally blocked or externally blocked, or both (which would correspond to the blocked states already described). The ownership of internally blocked lines cannot be transferred from one L1 cache to another, and the ownership of an externally blocked line cannot be transferred from L2 to memory. This ensures that there is at most one backup of the data out of the chip, although there may be another in the chip, which is enough to guarantee correctness in case of faults.

[2]In owned state, additional invalidation messages and their corresponding acknowledgments would be needed.

3.2. Faults detected by the *lost request timeout*

The *lost request timeout* starts when a request (*GetX* or *GetS* message) is issued and stops once it is satisfied (that is, when the L1 cache acquires the data and the requested access rights for it). Hence, it will trigger whenever a request takes too much time to be satisfied or cannot be satisfied because any of the involved messages has been dropped, causing a deadlock. It is maintained by the L1 for each pending miss. Hence, the extra hardware required to implement it is one extra counter for each MSHR entry.

When the *lost request timeout* triggers, FTDIRCMP assumes that some message which was necessary to finish the transaction has been lost due to a transient fault and retries the request. The particular message that may have been lost is not very important: it can be the request itself (*GetX* or *GetS*), an invalidation request sent by the L2 or the memory controller (*Inv*), a response to the request (*Data* or *DataEx*) or an invalidation acknowledgment (*Ack*). The timeout is restarted after the request is reissued to be able to detect additional faults.

To retry the request, the L1 chooses a new request serial number and will ignore any response which arrives with the old serial number after the *lost request timeout* triggers. See section 3.5 for more details.

As mentioned before, the L2 needs to be able to detect reissued requests and merge them in the MSHR with the original request (assuming it was not lost). The L2 will identify an incoming request as reissued if it has the same requestor and address than another request currently in the MSHR but a different request serial number.

A node which holds a line in *backup* state should also detect reissued requests to be able to resend the data (using the new serial number). Hence, every cache that transmits owned data needs to remember the destination node of that data at least until the ownership acknowledgment is received. This way, if a *DataEx* response is lost, it will be detected using the *lost request timeout* and corrected by resending the request.

This timeout is also used for writeback requests (*Put* messages). The timeout starts when the *Put* message is sent and stops once the writeback acknowledgment (*WbAck* message) is received. When it triggers, the *Put* message will be reissued with a different serial number. This way, this timeout can detect the loss of *Put* and *WbAck* messages but not the loss of *WbData* or *WbNoData* messages which is handled by the *lost unblock timeout*.

3.3. Faults detected by the *lost unblock timeout*

Unblock messages (*Unblock* or *UnblockEx*) are sent by the L1 once it receives the data and all required invalidation acknowledgments to notify the L2 that the miss has been satisfied. When the L2 receives one of these messages, it proceeds to attend the next miss for that line, if any.

When an unblock message is lost, the L2 will be blocked indefinitely and will not be able to attend further requests for the same line. Lost unblock messages cannot be detected by the *lost requests timeout* because that timeout is deactivated once the request is satisfied, just before sending the unblock message.

To avoid a deadlock due to a lost unblock message, the L2 starts the *lost unblock timeout* when it answers to a request and waits for an unblock message to finalize the transaction. When this timeout triggers, it will send an *UnblockPing* message to the L1.

When an L1 cache receives an *UnblockPing* message and it has already satisfied that miss (hence it has already sent a corresponding unblock message which may have been lost or not), it will answer with a reissued *Unblock* or *UnblockEx* message, depending on whether it has exclusive or shared access to the line. If the miss has not been resolved yet (hence no unblock message could have been lost because it was not sent in the first place), the *UnblockPing* message will be ignored. The L1 cache can check whether the miss has been already resolved or not by looking at its MSHR for a pending miss for the same address.

Unblock messages are also exchanged between the L2 and the memory controller in an analogous way. Hence, FtDirCMP uses an unblock timeout and *UnblockPing* in the memory controller too.

Also, this timeout is used to detect lost writeback messages (*WbData* and *WbNoData*) in a similar manner. When a *Put* is received by the L2 (or the memory), the timeout is started and a *WbAck* is sent to L1 (or L2) to indicate that it can perform the eviction and whether data must be sent or not. Upon receiving this message, the L1 stops its *lost request timeout*, sends the appropriate writeback message and assumes that the writeback is already done. Once the writeback message arrives to L2, the *lost unblock timeout* is deactivated. If the writeback message is lost (or it just takes too long to arrive), the timeout will trigger and the L2 will send a *WbPing* message to L1. The L1 will answer with a new writeback message (in case it still has the data) or a *WbCancel* message which tells the L2 that the writeback has already been performed. Note that modified data cannot be lost thanks to the rules described in section 3.1.

3.4. Faults detected by the *lost backup deletion acknowledgment timeout*

As explained in section 3.1, when ownership has to be transferred from a node to another, FtDirCMP uses a pair of acknowledgments to ensure the reliable transmission of the data. Losing any of these acknowledgments would lead to a deadlock which will not be detected by the *lost request* or *lost unblock* timeout (unless the ownership acknowledgment was lost along with an unblock message) because these timeouts are deactivated once the miss has been satisfied.

For these reasons, we intoduce the *lost backup deletion acknowledgment* timeout which is started when an ownership acknowledgment is sent and is stopped when the backup deletion acknowledgment arrives. This way, it will trigger if any of these acknowledgments is lost or arrives too late. When it triggers, a new *AckO* message will be sent with a newly assigned serial number.

If the ownership acknowledgment was actually lost, the new message will hopefully arrive to the node that is holding a backup of the line and that backup will be discarded and an *AckBD* message will be returned.

If the first ownership acknowledgment did arrive to its destination (false positive), the new message will arrive to a node which no longer has a backup and which already responded with an *AckBD* message. Anyway, a new *AckBD* message will be sent using the serial number of the new message. The old *AckBD* message will be discarded (if it was not actually lost) because it carries an old serial number.

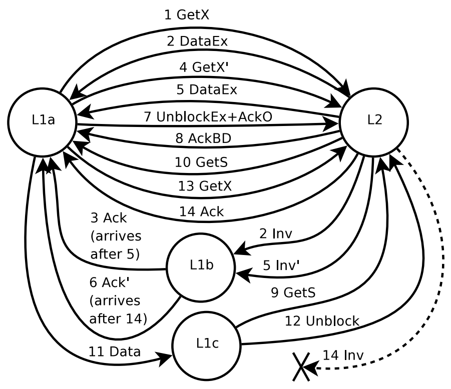
3.5. Request serial numbers

As described above, when a *lost request timeout* triggers FtDirCMP assumes that the request message or some response message has been lost due to a transient fault and then reissues the request hoping that no fault will occur this time. However, sometimes the timeout may trigger before the response has been received due to unusual network congestion or any other reason that causes an extraordinarily long latency for solving a miss. That is, there may be false positives.

In case of a false positive, two or more duplicate response messages would arrive to the requestor and, in some cases, the extra messages could lead to an incoherence (see figure 2 for an example). For this reason, FtDirCMP uses *request serial numbers* to discard responses which arrive too late, when the request has already been reissued.

Every request and every response message carries a serial number. Request serial numbers are chosen by the L1 cache that issues the request (or by the L2 in case of writebacks from L2 to memory). Responses or forwarded requests will carry the serial number 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 responses to the new one.

The L1 cache, L2 cache and memory controller must remember the serial number of the requests that they are currently handling and discard any message which ar-



Initially, the data is present in L2 in O state and in L1b in S state. L1a makes a write request (1) to L2 which sends an invalidation to L1b and a *DataEx* message to L1a (2). The *DataEx* also tells L1a that it needs to wait for one invalidation acknowledgment before entering the M state. When L1b receives the invalidation, it sends an acknowledgment to L1a (3). However, due to network congestion, this message takes a long time to arrive to L1a, hence the *lost request timeout* triggers and L1a reissues the write request (4) to L2. The reissued request arrives to L2 which resends the data message and the invalidation request (5). When the invalidation request arrives to L1b, it resends an acknowledgment (6) to L1a. Notice that the first acknowledgment (3) will arrive to L1a before the second one (6) because the network is point-to-point ordered. It will be accepted if request serial numbers are not used (otherwise it would be discarded and L1a would wait for the second one) and an *UnblockEx* message (6) carrying also the ownership acknowledgment will be sent to L2 which will answer with an *AckBD* (8). Now, due to the stale acknowledgment (6) that is traveling through the network, the system can arrive to an incoherent state: another cache L1c issues a read request (9) which is forwarded (10) by L2 to L1a. L1a answers it (11) and transitions to O state. L1c receives it (11), sends an unblock (12) and transitions to S state. Next, L1a issues a new write request (13) to L2 which sends an invalidation message to L1c and an acknowledgment message to L1a which tells it that it needs to wait for one invalidation acknowledgment (14). The invalidation request gets lost due to corruption. If the stale acknowledgment (6) arrives now to L1a, it will assume that it can transition to M state despite the fact that L1c is still in S state, thus violating coherency.

Figure 2. Transaction where request serial numbers are needed to avoid incoherency.

rives with an unexpected serial number or from an unexpected sender. This information needs to be updated when a reissued request arrives. Discarding any message in FtDirCMP 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.

Analogously, serial numbers are also used to be able to discard duplicated unblock messages, duplicated writeback messages or duplicated backup deletion acknowledgments. These duplicated messages can appear due to unnecessary *UnblockPing*, *WbPing* or duplicated ownership acknowledgment messages sent in the case of false positives of the *lost unblock timeout* or the *lost backup deletion acknowledgment timeout*.

3.5.1 Choosing serial numbers and their size

Serial numbers are used to discard duplicated responses to reissued requests[3]. This means that they need to be different for requests to the same address and by the same node (specially for reissued requests) but it does not matter if the same serial number is used for requests to different addresses or by different requestors.

On the other hand, the number of available serial numbers is finite. In our implementation, we use a small number of bits to encode the serial number in messages and MSHRs to minimize the overhead.

Since the initial serial number of a request is not important, we can choose it “randomly”. For example, in our implementation, each node has a wrapping counter which is used to choose serial numbers for new requests.

On the other hand, when reissuing a request, it is desirable to minimize the chances of using the serial number of any response to the former request currently traveling through the network. For this, serial numbers for reissued requests are chosen sequentially increasing the serial number of the previous attempt (wrapping to 0 if necessary). This way, when using n bits to encode the serial number, we would have to reissue the same request 2^n times before having any possibility of receiving a response to an old request and accepting it, which could cause problems in some situations as shown in figure 2.

3.6. Overhead estimation

The main overhead introduced by our protocol is the extra network traffic due to the acknowledgment used to ensure reliable ownership transference. Message sizes may also have to be increased to make room for the request serial numbers and the CRC.

There is also a small hardware overhead due to the counters that need to be added to MSHRs for the timeouts and additional space in the MSHR (or a separate structure) for storing the request serial number of the transaction, the identity of the requester currently being serviced (in the L2 cache and memory controller), and the identity of the receiver of owned data when transferring ownership (in the L1 cache, to make possible to detect reissued forwarded requests). Finally, FtDirCMP requires two virtual channels more than DirCMP.

4. Evaluation

4.1. Methodology

We have experimentally measured the overhead of FtDirCMP in comparison with DirCMP both in terms of execution time overhead and network traffic overhead. For this, we have performed full system simulations using Multifacet GEMS [11] detailed memory model and Virtutech Simics [9].

We have simulated a tiled CMP as described in section 2. Table 4 shows the most relevant configuration parameters of the modeled system. The values chosen for the fault-detection timeouts have been chosen exper-

[3]In this context, we can consider *UnblockPing*, *WbPing*, *AckO* and *WbAck* messages as requests too.

Table 4. Characteristics of simulated architectures.

16-Way Tiled CMP System	
Processor parameters	
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
Fault tolerance parameters	
Lost request timeout	2000 cycles
Lost unblock timeout	4000 cycles
Lost backup deletion acknowledgment	4000 cycles
Request serial number size	8 bits

imentally to minimize the number of false positives, thus ensuring minimal performance degradation in the fault-free scenario.

Finally, we have used a selection of scientific applications for the evaluation: 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), Water-NSQ (512 molecules, 4 time steps), and Water-SP (512 molecules, 4 time steps) are from the SPLASH-2 [22] benchmark suite. Tomcatv (256 points, 5 iterations) is a parallel version of a SPEC benchmark and Unstructured (Mesh.2K, 5 time steps) is a computational fluid dynamics application. The experimental results reported here correspond to the parallel phase of each program only. Every simulation has been performed several times using different random seeds to account for the variability of multithreaded execution. Such variability is represented by the error bars in the figures which enclose the resulting 95% confidence interval of the results.

4.2. Results

We have measured the execution time of DIRCMP in a fault-free scenario and compared it to FTDIRCMP with several message loss rates. The results are shown in figure 3. Fault rates are expressed in number of messages discarded per million of messages that travel through the network.

First, we see that there is no measurable overhead in terms of execution time for FTDIRCMP with respect to DIRCMP when there are no faults (DirCMP-0 and FtDirCMP-0 bars respectively). This is consistent with the fact that, when no faults occur, the main difference between our proposal and a standard directory-based protocol is just the extra acknowledgments used to ensure reliable data transmission, which are sent out of the critical path of misses.

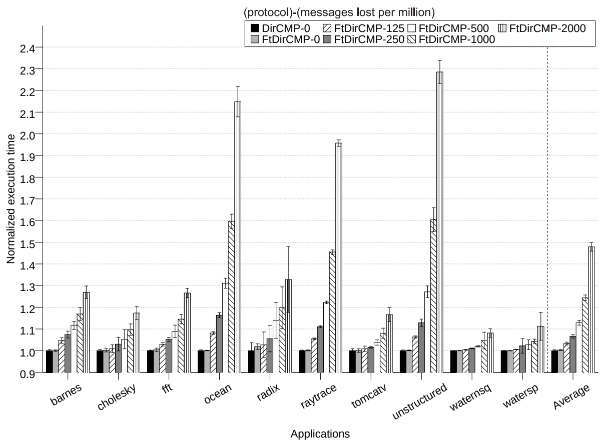


Figure 3. Execution time overhead of FTDIRCMP compared to DIRCMP for several fault rates.

As the fault rate increases, so does the execution time. The performance degradation depends mainly on the latency of the error detection mechanism. Hence, shortening the fault detection timeouts can reduce performance degradation when faults happen but at the risk of increasing the number of false positives which could lead to performance degradation in the fault-free case. With the timeouts used in this work, the performance degradation is not severe for most benchmarks even with fault rates which are unrealistically high. The execution time of three benchmarks doubles when the fault rate reaches 2000 messages lost per million (FtDirCMP-2000 bar), but on average execution time increases less than 50% even for the highest fault rate tested. Obviously, DIRCMP would not be able to execute correctly for any fault rate greater than zero.

We have also measured the network overhead of our proposal in terms of the relative increase of both number of messages and bytes transmitted through the network. These results are shown in figure 4 for the fault-free scenario and categorized by type of message. We can see that, on average, the overhead in terms of number of messages that FTDIRCMP introduces is less than 30%. Moreover, the overhead drops to 10% when it is measured in terms of bytes. These overheads represent the main cost of the fault tolerance features of our protocol. As can be seen, the overhead comes entirely from the acknowledgments used to ensure reliable ownership transference as explained in section 3.1 (portion *ownership* of each bar).

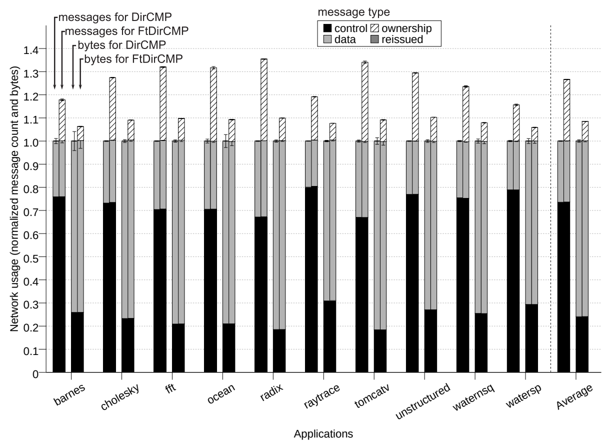


Figure 4. Network overhead of FtDirCMP **compared to** DirCMP **without faults.**

5. Related work

Fault tolerance for multiprocessors has been thoroughly studied in the past. Most proposals deal with transient errors by means of checkpointing and recovery. For example, Pruvlovic *et al.* presented ReVive [19], which performs checkpointing, logging and memory based distributed parity protection with low overhead in error-free execution and is compatible with off-the-self processors, caches and memory modules. At the same time, Sorin *et al.* presented SafetyNet [20] which aims at the same objectives but has less overhead and uses custom caches.

Recently, Meixner *et al.* proposed error detection techniques [13, 14] for multiprocessors which can detect errors that lead to memory consistency or coherence violations, but do not provide any recovery mechanism. Also, Aggarwal *et al.* [1] proposed a mechanism to provide dynamic reconfiguration of CMPs which enables fault containment and reconfiguration, but does not directly address the problems caused by a faulty interconnection network in the coherence protocol.

An alternative way to solve the problem of transient failures in the on-chip interconnection network is making the network itself fault tolerant. There are several proposals [4, 5, 16–18] exploring this approach. Ensuring the reliable transmission of all messages through the network imposes significant overheads in latency, power consumption and area. In contrast, our protocol allows for more flexibility to design a high-performance on-chip network which can be unreliable. The protocol itself ensures the reliable retransmission of those few messages that carry owned data and could cause data loss.

In a previous work [7], we presented a low overhead fault tolerant protocol based on the token coherence framework [10]. This work applies similar ideas for directory-based cache coherence protocols. Directory-based protocols are better known than token-based ones and are actually used in commercial systems. Also, we expect that directory-based protocols will be used predominantly for future CMPs due to their good scalability characteristics in terms of interconnection network usage and power consumption. Hence, we think that our current work is more relevant than the previous one.

In both protocols, fault detection is achieved by means of a number of timeouts which detect deadlocks caused by lost messages. Also we provide essentially the same mechanism to achieve the reliable transmission of owned data. The main differences between our previous fault tolerant protocol and this one are:

- In our previous protocol, fault recovery was done by means of a centralized mechanism called the *token recreation process* arbitrated by the memory controller. In this work, fault recovery is achieved simply reissuing requests with a different serial number.
- The *token serial numbers* used in our previous protocol serve a similar purpose to *request serial numbers*, but the latter are easier to implement and more scalable. *Token serial numbers* were associated with each cache line and needed to be updated in a coordinated fashion during the *token recreation process*. Hence, they required an additional structure in each cache to store them (even for lines which were not currently in the cache, but only for those hopefully few lines that had a token serial number different than 0). 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.

6. Conclusion

In this work, we have shown how to build a fault-tolerant directory-based coherence protocol which can ensure the correct execution of programs even if the interconnection network is subject to transient failures and does not correctly deliver all the coherence messages. Our protocol uses error detection codes (CRC) to detect corrupted messages and discard them upon reception, uses a number of timeouts to detect faults, adds acknowledgments only for a small number of messages, and uses request retries to resolve deadlock situations caused by transient failures.

We have evaluated the overhead of our protocol with respect to a base directory-based non fault-tolerant coherence protocol both in terms of execution time overhead and network usage overhead. We have found that, in absence of failures, the overhead of our protocol is minimal: the execution time does not increase, there is a very small hardware overhead and the network traffic increases moderately.

We have also performed fault injection to check the correctness of the protocol and to measure the performance degradation caused by several fault rates. We have found only a moderate performance degradation for fault rates which are much higher than what can be expected in a real scenario. Hence, we expect that the transient faults occurring in the interconnection network of a system using our protocol would have a negligible effect in performance.

In this way, our protocol provides a solution to transient failures in the interconnection network of CMPs using directory-based cache coherence protocols with very low overhead which can be combined with other fault tolerance measures to build reliable 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] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. In *34th Int’l Symp. on Computer Architecture (ISCA 2007)*, June 2007.

[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] R. Baumann. Soft errors in advanced computer systems. *IEEE Design and Test of Computers*, 22(3):258–266, 2005.

[4] K. Constantinides, S. Plaza, J. Blome, B. Zhang, V. Bertacco, S. Mahlke, T. 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.

[5] T. Dumitras, S. Kerner, and R. Mărculescu. Towards on-chip fault-tolerant communication. In *ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation*, pages 225–232, January 2003.

[6] R. Fernández-Pascual, J. M. García, M. E. Acacio, and J. Duato. A fault-tolerant directory-based cache coherence protocol for shared-memory architectures. Technical report, Universidad de Murcia, December 2007. TR-DITEC-UM-0001-2007.

[7] R. Fernández-Pascual, J. M. García, M. E. Acacio, and J. Duato. A low overhead fault tolerant coherence protocol for CMP architectures. In *13th Int’l Symp. on High-Performance Computer Architecture (HPCA’07)*, pages 157–168, February 2007.

[8] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE MICRO Magazine*, 20(2):71–84, March-April 2000.

[9] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[10] M. M. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *The 30th Annual International Symp. on Computer Architecture*, pages 182–193, June 2003.

[11] M. M. 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. *Computer Architecture News*, 33(4):92–99, September 2005.

[12] M. R. Marty and M. D. Hill. Virtual hierarchies to support server consolidation. In *ISCA ’07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 46–56, June 2007.

[13] A. Meixner and D. J. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 73–82, June 2006.

[14] A. Meixner and D. J. Sorin. Error detection via online checking of cache coherence with token coherence signatures. In *13th Int’l Symp. on High-Performance Computer Architecture (HPCA-13)*, pages 145–156, February 2007.

[15] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The soft error problem: An architectural perspective. In *11th Int’l Symp. on High-Performance Computer Architecture (HPCA’05)*, February 2005.

[16] S. Murali, T. Theocharides, N. Vijaykrishnan, M. J. Irwin, L. Benini, and G. D. Micheli. Analysis of error recovery schemes for networks on chips. *IEEE Design and Test of Computers*, 22(5):434–442, 2005.

[17] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C. R. Das. Exploring fault-tolerant network-on-chip architectures. In *Proceedings of the 2006 Int’l Conference on Dependable Systems and Networks (DSN’06)*, pages 93–104, 2006.

[18] M. Pirretti, G. Link, R. Brooks, N. Vijaykrishnan, M. Kandemir, and M. Irwin. Fault tolerant algorithms for network-on-chip interconnect. In *Proceedings of the IEEE Computer society Annual Symp. on VLSI*, pages 46–51, February 2004.

[19] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback. In *29th Annual Int’l Symp. on Computer Architecture (ISCA’02)*, pages 111–122, May 2002.

[20] D. J. Sorin, M. M. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *29th Annual Int’l Symp. on Computer Architecture (ISCA’02)*, pages 123–134, May 2002.

[21] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, J.-W. 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.

[22] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *22nd Int’l Symp. on Computer Architecture (ISCA’95)*, pages 24–36, June 1995.

[23] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *32th Int’l Symp. on Computer Architecture (ISCA’05)*, pages 336–345, June 2005.