A Low Overhead Fault Tolerant Coherence Protocol for CMP Architectures

Ricardo Fernández-Pascual, José M. García, Manuel E. Acacio
Dpto. Ingeniería y Tecnología de Computadores
Universidad de Murcia
SPAIN
{r.fernandez,jmgarcia,meacacio}@ditec.um.es

José Duato
Dpto. Informática de Sistemas y Computadores
Universidad Politécnica de Valencia
SPAIN
jduato@disca.upv.es

Abstract

*It is widely accepted that transient failures will appear more frequently in chips designed in the near future due to several factors such as the increased integration scale. On the other hand, chip-multiprocessors (CMP) that integrate several processor cores in a single chip are nowadays the best alternative to more efficient use of the increasing number of transistors that can be placed in a single die. Hence, it is necessary to design new techniques to deal with these faults to be able to build sufficiently reliable Chip Multiprocessors (CMPs). In this work, we present a coherence protocol aimed at dealing with transient failures that affect the interconnection network of a CMP, thus assuming that the network is no longer reliable. In particular, our proposal extends a token-based cache coherence protocol so that no data can be lost and no deadlock can occur due to any dropped message. Using GEMS full system simulator, we compare our proposal against a similar protocol without fault tolerance (*TOKENCMP*). We show that in absence of failures our proposal does not introduce overhead in terms of increased execution time over* TOKENCMP*. Additionally, our protocol can tolerate message loss rates much higher than those likely to be found in the real world without increasing execution time more than 15%.*

1. Introduction

Chip Multiprocessors (CMPs) [4, 3] are currently accepted as the best way to take advantage of the increasing number of transistors available in a single chip, since they provide better performance without excessive power consumption exploiting thread-level parallelism.

In many applications, high availability and reliability are critical requirements. The use of CMPs in critical tasks can be hindered by the increased rate of transient faults due to the ever decreasing feature size and higher frequencies. To enable more useful chip multiprocessors to be designed, several fault tolerant techniques must be employed in their construction.

Moreover, the reliability of electronic components is never perfect. Electronic components are subject to several types of failures due to a number of sources. Failures can be either permanent, intermittent or transient. Permanent failures require the replacement of the component and are caused by electromigration among other causes. Intermittent failures are mainly due to voltage peaks or falls.

Transient failures [11], also known as soft errors or single event upsets, occur when a component produces an erroneous output and it continues working correctly after the event. The causes of transient errors are multiple and include alpha-particle strikes, cosmic rays, and 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. Any event which upsets the stored or communicated charge can cause soft errors in the circuit output.

Transient failures are much more common than permanent failures [13]. Currently, transient failures are already significant for some devices like caches, where error correction codes are used to deal with them. However, current trends of higher integration and lower power consumption will increase the importance of transient failures [6]. Since the number of components in a single chip increases so much, it is no longer economically feasible to assume a worst case scenario when designing and testing the chips. Instead, new designs will target the common case and assume a certain rate of transient failures. Hence, transient failures will affect more components and more frequently and will need to be handled across all the levels of the system to avoid actual errors.

Communication between processors in a CMP is very fine-grained (at the level of cache lines), hence small and frequent messages are used. In order to achieve the best possible performance it is necessary to use low-latency interconnections and avoid acknowledgement messages and other control-flow messages to ensure a reliable transmission.

In this work, we propose a way to deal with the transient failures that occur in the interconnection network of CMPs. We can assume that these failures cause the loss of some cache coherence messages, because either the interconnection network loses them, or the messages reach the destination node (or other node) corrupted. Messages corrupted by a soft error will be discarded upon reception using error detection codes.

We attack this problem at the cache coherence protocol level. We have designed a coherence protocol which assumes an unreliable interconnection network and guarantees correct execution in presence of dropped messages. Our proposal only modifies the coherence protocol and does not add any requirement to the interconnection network, so it is applicable to current and future designs. Since the coherence protocol is critical for good performance and correct execution of any workload in a CMP, it is important to have a fast and reliable protocol.

Up to the best of our knowledge, there has not been any proposal dealing explicitly with transient faults in the interconnection network of multiprocessors or CMPs from the point of view of the cache coherence protocol. Also, most fault tolerance proposals require some kind of checkpointing and rollback, while ours does not. Our proposal could be used in conjunction with other techniques which provide fault tolerance to individual cores and caches in the CMP to achieve full fault tolerance coverage inside the chip.

The main contributions of this paper are the following: we have identified the different problems posed to a token based CMP cache coherence protocol (TOKENCMP) by the loss of messages due to an unreliable interconnect, we have proposed modifications to the protocol and the architecture to cope with these problems without adding excessive overhead, and we have implemented such solutions in a full system simulator to measure their effectiveness and execution time overhead. We show that in absence of failures our proposal does not introduce overhead in terms of increased execution time over TOKENCMP. Additionally, our protocol can tolerate message loss rates much higher than those likely to be found in the real world without increasing execution time more than 15%.

The rest of the paper is organized as follows. In section 2 we present some related work. In sections 3 and 4 we describe the problems posed by an unreliable interconnection network and the solutions that we propose. Section 5 presents an evaluation of the overhead introduced by our proposal and its effectiveness in presence of faults. Finally, in section 6 we present some conclusions of our work.

2. Related work and background

There have been several proposals for fault tolerance targeting shared-memory multiprocessors. Most of them use variations of checkpointing and recovery: R.E. Ahmed *et al.* developed Cache-Aided Rollback Errors Recovery (CARER) [1], Wu *et al.* [17] developed error recovery techniques using private caches for recovering from processor transient faults in multiprocessor systems, Banâtre *et al.* proposed a *Recoverable Shared Memory* (RSM) which deals with processor failures on shared-memory multiprocessors using snoopy protocols [2], while Sunada *et al.* proposed *Distributed Recoverable Shared Memory with Logs* (DRSM-L) [15]. More recently, Pruvlovic *et al.* presented ReVive, which performs checkpointing, logging and memory based distributed parity protection with low overhead in error-free execution and is compatible with off-the-shelf processors, caches and memory modules [12]. At the same time, Sorin *et al.* presented SafetyNet [14] which aims at similar objectives but has less overhead, requires custom caches and can only recover from transient faults.

Regarding the cache coherence protocol background, Token coherence [7, 8] is a framework for designing coherence protocols whose main asset is that it decouples the correctness substrate from several different performance policies. This allows great flexibility, making it possible to adapt the protocol for different purposes easily [7] since the performance policy can be modified without worrying about infrequent corner cases, whose correctness is guaranteed by the correctness substrate.

The main observation of the token framework is that simple token counting rules can ensure that the memory system behaves in a coherent manner. *Token counting* specifies that each line of the shared memory has a fixed number of tokens and that the system is not allowed to create or destroy tokens. A processor is allowed to read a line only when it holds at least one of the line's tokens and has valid data, and a processor is allowed to write a line only when it holds all of its tokens and valid data. One of the tokens is distinguished as the *owner token*. The processor or memory module which has this token is responsible for providing the data when another processor needs it or write it back to memory when necessary. The owner token can be either clean or dirty, depending on whether the contents of the cache line are the same as in main memory or not, respectively. In order to allow processors to receive tokens without data, a *valid-data bit* is added to each cache line (independently of the usual valid-tag bit). These simple rules prevent a processor from reading the line while another processor is writing it, ensuring coherent behavior at all times.

Token coherence avoids starvation by issuing a persistent request when a processor detects potential starvation. Persistent requests, unlike transient requests, are guaranteed to eventually succeed. To ensure this, each token protocol must define how it deals with several pending persistent requests.

Token coherence provides the framework for designing several particular coherence protocols. TOKENCMP [10] is a performance policy which targets hierarchical multiple CMP systems. It uses a distributed arbitration scheme for persistent requests, which are issued after a single retry to optimize the access to contended lines.

3. Problems arising in CMPs with an unreliable interconnection network

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 message or a burst of them) since they directly disappear from the interconnection network or arrive to their destination corrupted and are discarded.

Instead of detecting faults and return to a consistent state previous to the occurrence of the fault, our aim is to design a coherence protocol that can guarantee the correct semantics of program execution over an unreliable interconnection network without ever having to perform a checkpointing or rollback. We do not try to address the full range of errors that can occur in a CMP system. We only concentrate on those errors that affect directly the interconnection network and which can be tolerated modifying the coherence protocol. Hence, other mechanisms should be used to complement our proposal to achieve full fault tolerance for the whole CMP. Next, we present the problems caused by the loss of messages in the TOKENCMP protocol and later we show how these problems can be solved.

From the point of view of the coherence protocol, we assume that a coherence message either arrives correctly to its destination or it does not arrive at all. In other words, we assume that no incorrect or corrupted messages can be processed by a node. To guarantee this, error detection codes are used. Upon arrival, the CRC is checked using specialized hardware and the message is discarded if it is wrong. To avoid any negative impact on performance, the message is speculatively assumed to be correct because this is by far the most common case.

There are several types of coherence messages that can be lost which translate into a different impact in the coherence protocol. Firstly, losing transient requests is harmless. Note that even when we state that losing the message is harmless we mean that no data loss, deadlock, or incorrect execution would be caused, although some performance degradation may happen.

Since invalidations (which can be persistent or transient requests) in the base protocol require acknowledgement (the caches holding tokens must respond to the requester), losing a message cannot lead to an incoherence.

Losing any other type of message, however, may lead to deadlock or data loss. Particularly, losing coherence messages containing one or more tokens would lead to a deadlock, because the total number of tokens in the whole system must remain constant to ensure correctness. More precisely, if the number of tokens decreases because a message carrying one or more tokens does not reach its destination, no processor will be able to write to that line of memory anymore.

The same thing happens when a message carrying data and tokens is lost, as long as it does not carry the owner token. No data loss can happen because there is always a valid copy of the data at the cache which has the owner token.

Another different case occurs if the lost coherence message contains a dirty owner token, since it must also carry the memory line. Hence, if the owner token is lost, no processor (or memory module) would send the data and a deadlock and possibly data loss would occur. In the TOKENCMP protocol, like in most cache coherence protocols, the data in memory is not updated on each write, but only when it is evicted from the owner cache. Also, the rules governing the owner token ensure that there is always at least a valid copy of the memory line which travels along with it every time that the owner token is transmitted. So, losing a message carrying the owner token means that it is possible to totally lose data.

Finally, while a persistent request is in process, we have to deal also with errors in the persistent request messages. Losing a persistent request or persistent request deactivation would create inconsistencies among the persistent request tables at each cache in a distributed arbitration scheme which would lead to deadlock situations too.

4. A fault tolerant token coherence protocol

The main principle that guided the protocol development was to prevent adding significant overhead to the fault-free case and to keep the flexibility of choosing any particular performance policy. Therefore, we should try to avoid modifying the usual behavior of transient requests. For example, we should avoid placing point-to-point acknowledgements in the critical path as much as possible.

Once a problematic situation has been detected, the main recovery mechanism used by our protocol is the *token recreation process* described later. That process resolves a deadlock ensuring both that there is the correct number of tokens

Table 1. Timeouts summary.

Timeout	When is it activated?	Where is it activated?	When is it deactivated?	What happens when it triggers?
Lost Token	When a persistent request becomes active.	At the starver cache.	When the persistent request is satisfied or deactivated.	Request a token recreation.
Lost Data	When a backup state is entered. (When the owner token is sent.)	At the cache that holds the backup.	When the backup state is abandoned. (When the Ownership Acknowledgement Arrives.)	Request a token recreation.
Lost Backup Deletion Acknowledgement	When a line in a blocked status needs to be replaced.	At the cache that needs to do the replacement.	When the blocked state is abandoned. (When the Backup Deletion Acknowledgement arrives.)	Request a token recreation.
Lost Persistent Deactivation	When a persistent request from another cache is activated.	At every cache (by the persistent request table).	When the persistent request is deactivated.	Send a persistent request ping.

and one and only one valid copy of the data.

As shown in the previous section, only the messages carrying transient read/write requests can be lost without negative consequences. For the rest of the cases, losing a message results in a problematic situation. However, all of these cases have in common that they lead to deadlock. Hence, a possible way to detect faults is using timeouts for transactions. We use four timeouts for detecting message losses: the *"lost token timeout"* (see section 4.1), the *"lost data timeout"*, the *"lost backup deletion acknowledgement timeout"* (see section 4.2) and the *"lost persistent deactivation timeout"* (see section 4.3.2). Notice that all these timeouts along with the usual retry timeout of the token protocol (except the *lost persistent deactivation timeout*) can be implemented using just one hardware counter, since they do not need to be activated simultaneously. For the *lost persistent deactivation timeout*, an additional counter per processor at each cache or memory module is required. A summary of these timeouts can be found in table 1.

Since the time to complete a transaction cannot be bounded reliably with a reasonable timeout due to the interaction with other requests and the possibility of network congestion, our fault detection mechanism may produce false positives, although this should be very infrequent. Hence, we must ensure that our corrective measures are safe even if no fault really occurred.

We present a summary of all the problems that can arise due to loss of messages and their proposed solutions in table 2. In the rest of this section, we explain how our proposal prevents or solves each one of these situations in detail.

Table 2. Summary of the problems caused by loss of messages.

Fault / Lost message	Effect	Detection and Recovery
Transient read/write request	Harmless	
Response with tokens	Deadlock	Lost token timeout, token recreation
Response with tokens and data	Deadlock	Lost token timeout, token recreation
Response with a dirty owner token and data	Deadlock and data loss	Reliable transfer of owned data using acknowledgements, lost data timeout
Persistent read/write requests	Deadlock	Lost token timeout, token recreation
Persistent request deactivations	Deadlock	Lost persistent deactivation timeout, persistent request ping
Ownership acknowledgement	Deadlock and cannot evict line from cache	Lost data timeout
Backup deletion acknowledgement	Deadlock	Lost backup deletion acknowledgement timeout

4.1. Dealing with token loss

When a processor tries to write to a memory line which has lost a token, it will eventually timeout and issue a persistent request. In the end, after the persistent request gets activated, all the available tokens in the whole system for the memory line will be received by the starving cache. Also, if the owner token was not lost and is not blocked (see section 4.2), the cache will receive it too together with data. However, since the cache will not receive all the tokens, it will not be able to complete the write miss, and finally the processor will be deadlocked.

We use the *"lost token timeout"* to detect this deadlock situation. It will start when a persistent request is activated and will stop once the miss is satisfied or the persistent request is deactivated. The value of the timeout should be long enough so that, in normal circumstances, every transaction will be finished before triggering this timeout[1].

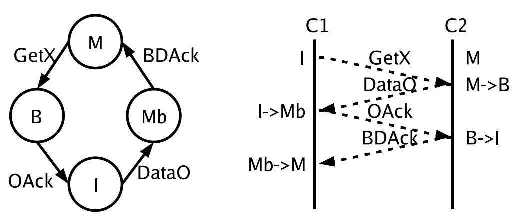
Hence, if the starving cache fails to acquire the necessary tokens within certain time after the persistent request has been activated, the *lost token timeout* will trigger. In

[1]Using a value too short for any of the timeouts used to detect faults would lead to many false positives which would hurt performance.

that case, we will assume that some token carrying message has been lost and we will request a token recreation process for recovery to the memory module (see section 4.4). Notice that the *lost token timeout* may be triggered for the same coherence transaction that loses the message or for a subsequent transaction for the same line. Once the token recreation has been done, the miss can be satisfied immediately.

4.2. Avoiding data loss

To avoid losing data in our fault tolerant coherence protocol, a cache (or memory controller) that has to send the owner token will keep the data line in a *backup* state. A line in backup state will not be evicted from the cache until an *ownership acknowledgement* is received, even if every token is sent to other caches. This acknowledgement is sent by every cache in response to a message carrying the owner token. While a line is in *backup* state its data is considered invalid and will be used only if required for recovery. Hence, the cache will not be able to read from that line[2]. Also, when a line enters in a backup state the *lost data timeout* will start and will stop once the backup state is abandoned.



Cache C1 broadcasts a transient exclusive request (GetX). C2, which has all the tokens and hence it is in *modified* state (M), answers to C1 with a message (DataO) carrying the data and all the tokens, including the owner token. Since C2 needs to send the owner token, it goes to a *backup* state (B) and starts the *lost data timeout*. When C1 receives the DataO message, it satisfies the miss and enters a *modified and blocked* state (Mb), sending an ownership acknowledgement to C2. When C2 receives it, it discards the backup, goes to *invalid* state (I), stops the *lost data timeout* and sends a *backup deletion acknowledgement* to C1. Once C1 receives it, it transitions to a normal *modified* state.

Figure 1. Transition diagram for the states and events involved in data loss avoidance and message interchange example.

A cache line in a backup state will be used for recovery if no valid copy is available when a message carrying the owner token is lost. To be able to do this in an effective way, it is necessary to ensure that there is a valid copy of the data or one and only one backup copy at all times, or both[3]. Hence, a cache which has received the owner token recently cannot transmit it again until it is sure that the backup copy for that line has been deleted. We say that the line will be in a *blocked ownership* state. A line will leave this state when the cache receives a *backup deletion acknowledgement* which is sent by any cache when it deletes a backup copy after receiving an *ownership acknowledgement*. Figure 1 shows an example of how the owner token is transmitted with our protocol.

The two acknowledgements necessary to finalize this transaction are out of the critical path of the miss. However, there is a period after receiving the owner token until the *backup deletion acknowledgement* arrives during which a cache cannot answer to write requests because it would have to transmit the owner token, which is blocked. This blocking also affects persistent requests, which are serviced immediately after receiving the *backup deletion acknowledgement*. This blocked period could increase the latency of some cache-to-cache transfer misses, however we have found that it does not have impact on performance, as most writes are sufficiently separated in time.

This mechanism also affects replacements (from L1 to L2 and from L2 to memory), since the replacement cannot be performed until an *ownership acknowledgement* is received. We have found that the effect on replacements is much more harmful for performance than the effect of cache-to-cache transfer misses mentioned above.

To alleviate the effect of the blocked period in the latency of replacements, we propose using a small *backup buffer* to store the backup copies. In particular, we add a backup buffer to each L1 cache. A line is moved to the backup buffer when it is in a backup state, it needs to be replaced and there is enough room in the backup buffer[4]. The backup buffer acts as a small victim cache, except that only lines in backup states are moved to it. We have found that a small backup buffer with just 1 or 2 entries is enough to practically remove the negative effect of blocked ownership states (see section 5.2).

4.2.1 Handling the loss of an owned data carrying message or an ownership acknowledgement

Losing a message which carries the owner token means that possibly the only valid copy of the data is lost. However, there is still an up to date backup copy at the cache which

[2] It is possible for a cache to receive valid data and a token before abandoning a backup state, only if the data message was not lost. In that case, it will be able to read from that line and the line will be transitioned to an intermediate backup state until the *ownership acknowledgement* is received.

[3] Having more than one backup copy would make recovery impossible, since it could not be known which backup copy is the most recent one.

[4] We do not move the line to the backup buffer immediately after it enters a backup state to avoid wasting energy in many cases and avoid wasting backup buffer space unnecessarily.

sent the data carrying message. Since the data carrying message does not arrive to its destination, no corresponding *ownership acknowledgement* will be received by the cache and the *lost data timeout* will trigger.

If an *ownership acknowledgement* is lost, the backup copy will not be discarded and no *backup deletion acknowledgement* will be sent. Hence, the backup copy will remain in one of the caches and the data will remain blocked in the other. Eventually, the *lost backup deletion acknowledgement timeout* will trigger too.

When the *lost backup deletion acknowledgement timeout* triggers, the cache requests a token recreation process to recover the fault (see section 4.4). The process can solve both situations: if the *ownership acknowledgement* was lost, the memory controller will send the data which had arrived to the other cache; if the data carrying message was lost, the cache will use the backup copy as valid data after the recreation process ensures that all other copies have been invalidated.

4.2.2 Handling the loss of a backup deletion acknowledgement

When a *backup deletion acknowledgement* is lost, a line will stay in a blocked ownership state. This will prevent it from being replaced or to answer any write request. Both things would lead to a deadlock if they are not resolved.

If a miss cannot be resolved because the line is blocked in some other cache waiting for a *backup deletion acknowledgement* which has been lost, eventually a persistent request will be activated for it and after some time the *lost token timeout* will trigger. Hence, the *token recreation process* will be used to solve this case.

To be able to replace a line in a blocked state when the *backup deletion acknowledgement* is lost, we use the *lost backup deletion acknowledgement* timeout. It is activated when the replacement is necessary, and deactivated when the *backup deletion acknowledgement* arrives. If it triggers, a *token recreation process* will be requested.

The token recreation process will solve the fault in both cases, since even lines in blocked states are invalidated and must transfer their data to the memory controller.

4.3. Dealing with errors in persistent requests

Assuming a distributed arbitration policy, persistent request messages (both requests and deactivations) are always broadcasted to keep the persistent request tables at each cache synchronized. Losing one of these messages will lead to an inconsistency among the different tables.

If the persistent request tables are inconsistent, some persistent requests may not be activated by some caches or some persistent requests may be kept activated indefinitely. These situations could lead to starvation.

4.3.1 Dealing with the loss of a persistent request

Firstly, it is important to note that the cache which issues the persistent request will always eventually activate it, since no message is involved to update its own persistent request table.

If a cache holding at least one token for the requested line which is necessary to satisfy the miss does not receive the persistent request, it will not activate it in its local table and will not send the tokens and data to the starver. Hence, the miss will not be resolved and the starver will deadlock.

Since the persistent request has been activated at the starver cache, the *lost token timeout* will trigger eventually and the token recreation process will solve this case too.

On the other hand, if the cache that does not receive the persistent request did not have tokens necessary to satisfy the miss, it will eventually receive an unexpected deactivation message which it should ignore.

4.3.2 Dealing with the loss of a deactivation message

If a persistent request deactivation message is lost, the request will be permanently activated at some caches. To avoid this, caches will start the *lost persistent deactivation timeout* when a persistent request is activated and will stop it when it is deactivated. When this timeout triggers, the cache will send a *persistent request ping* to the starver. A cache receiving a *persistent request ping* will answer with a persistent request or persistent request deactivation message whether it has a pending persistent request for that line or not, respectively. The *lost persistent deactivation timeout* is restarted after sending the *persistent request ping* to cope with the potential loss of this message.

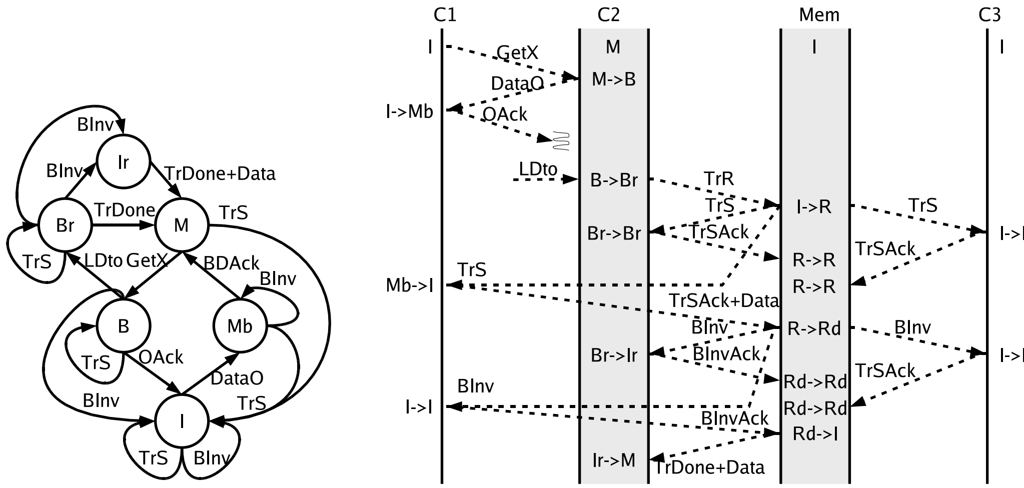
If the cache receives a persistent request from the same starver before the *lost persistent deactivation timeout* triggers, it should assume that the deactivation message has been lost and deactivate the old request, because caches can have only one pending persistent request.

4.4. Token recreation process

The *token recreation* is the main fault recovery mechanism provided by our proposal. This process needs to be effective, but since it should happen very infrequently, it does not need to be particularly efficient. In order to avoid any race and keep the process simple, the memory controller will serialize the token recreation process, attending token recreation requests for the same line in FIFO order.

The process works as long as there is at least a valid copy of the data in some cache or one and only one backup copy of the data or both things (the valid data or backup can be at the memory too). The protocol guarantees that these conditions are true at every moment, despite any message loss[5].

[5] In particular, these conditions are true if no message has been lost,



In a transaction like the one of figure 1 the *ownership acknowledgement* gets lost. Hence, C2 keeps the line in backup state (B). After some time, the *lost data timeout* triggers (LDto) and C2 sends a *token recreation request* message (TrR) to the memory controller and enters the *backup and recreating* state. The memory controller sends a *set token serial number* message (TrS) to each cache. C2 and C3 receive this message and answer with an acknowledgement (TrSAck) without changing their states, since they are either in invalid or backup state. On the other hand, C1 is in *modified and blocked* state, hence it returns an acknowledgement with data (TrSAck+Data) and changes its state to *invalid* (I). When the memory receives the acknowledgement with data, it sends a *backup invalidate* message to each cache. C1 and C3 answer with an acknowledgement (BInvAck) without changing their states, while C2 discards its backup data (which could be invalid since C1 may have written already to the cache line), sets its state to *invalid and recreating* (Ir) and answers with an acknowledgement too. When the memory receives all the acknowledgements, it sends a *destruction done* message to C2 including the new data (TrDone+Data). Finally, C2 receives the new data and sets its state to *modified* (M).

Figure 2. Transition diagram for the states and events involved in the token recreation process (used in this case to recover from the loss of an ownership acknowledgement).

If there is at least a valid copy of the data, it will be used for the recovery. Otherwise, the backup copy can be used for recovery.

At the end of the process, there will be one and only one copy of the data with all the tokens (recreating any token which may have been lost) at the cache which requested the token recreation process.

There is one exception to this when the data was actually lost (hence no valid copy of it exists, only a backup copy) and the *token recreation process* was requested by a cache other than the one which holds the backup copy. In this case, the *token recreation process* will fail to recreate the tokens, but the cache that holds the backup copy will eventually request another token recreation process (because its *lost data acknowledgement timeout* will trigger), and this new process will succeed using its backup copy to recover the data.

When recreating tokens, we must ensure the *Conservation of Tokens* invariant [7]. In particular, if the number of tokens increases, a processor would be able to write to the memory line while other caches hold readable copies of the line, violating the memory coherence model. So, to avoid increasing the total number of tokens for a memory line even in the case of a false positive, we need to ensure that all the old tokens are discarded after the recreation process. To achieve this we define a *token serial number* conceptually associated with each token and each memory line.

All the valid tokens of the same memory line should have the same serial number. The serial number will be transmitted within every coherence response. Every cache in the system must know the current serial number associated with each memory line and should discard every message received containing an incorrect serial number. The *token recreation process* modifies the current *token serial number* associated with a line to ensure that all the old tokens are discarded. Hence, if there was no real failure but a token carrying message was delayed on the network due to congestion (a false positive), it will be discarded when received by any cache because the *token serial number* will not match.

hence the *token recreation process* is safe for false positives and can be requested at any moment.

To store the token serial number of each line we propose a small associative table present at each cache. Only lines with an associated serial number different than zero must keep an entry in that table. The overhead of the token serial number is small. In the first place, we will need to increase it very infrequently, so a counter with a small number of bits should be enough (we use a two bit wrapping counter). Secondly, most memory lines will keep the initial serial number unchanged, so we only need to store those ones which have changed it and assume the initial value for the rest. Thirdly, the comparisons required to check the validity of received messages can be done out of the critical path of cache misses.

Since the *token serial number* table is finite, serial numbers are reset using the own token recreation mechanism when the table is full and a new entry is needed, since resetting a *token serial number* actually frees up its entry in the table.

The information of the tables must be identical in all the caches except while it is being updated by the token recreation process. The process works as follows:

When a cache decides that it is necessary to start a *token recreation* process, it sends a *recreate tokens* request to the memory controller responsible for that line. The memory can also decide to start a *token recreation process*, in which case no message needs to be sent. The memory will queue *token recreation* requests for the same line and service them in order of arrival.

When servicing a *token recreation* request, the memory will increase the *token serial number* associated to the line and send a *set token serial number* message to every cache.

When receiving that message, each cache updates the *token serial number*, destroys any token that it could have and sends an acknowledgement to the memory. The acknowledgement will also include data if the cache had valid data (even if it was in a blocked owner state).

Since all the tokens held by a cache are destroyed, the state of the line will become invalid, even if the line was in a blocked owner state. However, if the line was held in a backup state, it will remain that way.

If the memory controller receives an acknowledgement with data, it will send a *backup invalidate* message to all the caches. When receiving that request, the caches will send an acknowledgement and discard its backup copy. This avoids having two backup copies when several faults occur and two or more backup recreation processes are requested in quick succession.

Once the memory receives all the acknowledgements (including the acknowledgements for the backup invalidation if it has been requested), it will send a *destruction done* message to the cache which initiated the recreation process (unless it is the memory itself). The *destruction done* message will include the data if it was received by the memory or the memory had a valid copy itself, otherwise it means that there was no valid copy of the data and there must be a backup copy in some cache (most likely in the same cache that requested the token recreation).

When a cache receives a *destruction done* message with data, it will recreate all the tokens (with the new *token serial number*) and hence set its state to *modified*. If the *destruction done* message came without data and the cache was in backup state, it will use the backup data and recreate the tokens anyway. If the *destruction done* message came without data and the cache did not have a backup copy, it will not be able to recreate the tokens, instead it will restart the usual timeouts for the cache miss. As mentioned above, when this last case happens there must be a backup copy in another cache and the *lost data timeout* of that cache will eventually trigger and recover from this fault. Figure 2 shows an example of the *token recreation* process at work.

4.4.1 Handling faults in the token recreation process

Since the efficiency of the token recreation process is not a great concern, we can use unsophisticated (brute force) methods to avoid problems due to losing the messages involved. Hence, all of these messages are repeatedly sent every certain number of cycles (1000 in our current implementation) until an acknowledgement is received. Serial numbers are used to detect and ignore duplicates unnecessarily sent.

4.5. Hardware overhead of our proposal

Firstly, to implement the token serial number table we have added a small associative table at each cache and at the memory controller to store those serial numbers whose value is not zero. In this work, we have assumed that each serial number requires two bits (if the tokens of any line need to be recreated more than 4 times the counter will wrap) and that 16 entries are sufficient (if more than 16 different lines need to be stored in the table, the least recently modified entry will be chosen for eviction using the token recreation process to reset the serial number).

Most of the timeouts employed to detect faults can be implemented using the same hardware that is already used to implement the starvation timeout required by token coherence protocols, although the counters may need more bits since the new timeouts are longer. For the *lost persistent deactivation* timeout it is necessary to add a new counter per processor at each cache and at the memory controller.

Also, some hardware is needed to calculate and check the error detection code used to detect and discard corrupt messages.

Finally, to avoid performance penalty in replacements due to the blocked ownership period, we have proposed to add a small backup buffer at each L1 cache. The backup buffer can be effective having just one entry, as will be shown in section 5.2.

5. Evaluation

5.1. Methodology

We have evaluated the performance of our proposal using full system simulation. We have used Virtutech Simics [5] functional simulator with Multifacet GEMS [9] timing infrastructure. Although GEMS can model out-of-order processors, we have used the in-order model provided by Simics to keep simulation times tractable and because most probably future cores in CMPs will use in-order execution to reduce power consumption. Using out-of-order execution would not affect the correctness of the protocol at all and would not have measurable effect in the overhead introduced by the fault tolerance measures compared to the non fault tolerant protocol.

We have implemented the proposed fault tolerant coherence protocol using the detailed memory model provided by GEMS simulator (Ruby) to evaluate its overhead compared to the TOKENCMP [10] protocol and to check its effectiveness dealing with message losses. TOKENCMP is a token based coherence protocol without fault tolerance provision but that has been optimized for performance in CMPs.

We model a CMP whose more relevant configuration parameters are shown in table 3. Although we use an in-order processor model for simulation efficiency, the simulated processor frequency is four times as fast as the memory model frequency to approximate a 4-way superscalar model. We have evaluated CMP configurations consisting on 4, 8 and 16 processor cores.

Finally, 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 [16] 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.

5.2. Measuring the overhead for the fault-free case

First, we evaluate both execution time overhead and network overhead of our protocol when no messages are lost. As previously explained, the execution time overhead depends on the size of the backup buffer (see section 4.2). Figure 3 plots it using different sizes for the backup buffer, including the case of not having a backup buffer at all.

As derived from figure 3, without a backup buffer the overhead in terms of execution time is more than 5% on average for the 4-core CMP and more than 10% for some benchmarks, which we think is not acceptable. The results for 8-core and 16-core CMPs are similar too. We have found that this slowdown is due to the increased latency of the misses which need a replacement of an owned line first, since the replacement is no longer immediate but has to wait until an *ownership acknowledgement* is received from the L2 cache.

Fortunately, the use of a very small backup buffer is enough to avoid nearly all this penalty. In the 4-core CMP, a backup buffer of just one entry cuts down the penalty to less than 0.5% on average. For 8 cores the penalty is reduced to less that 0.5% using a single entry too. And for the 16-core architecture, the slowdown using one entry in the backup buffer is less than 1%.

Additionally, a backup buffer big enough could even improve the execution time when compared to the non fault tolerant protocol in some cases, as seen in figure 3 for some benchmarks when the backup buffer size is 2 or 4. This is because the backup buffer can act temporarily as a victim cache when a miss happens while a line was in a backup state. However, there is no significant performance improvement with respect to TOKENCMP on average.

The other potential source of miss latency overhead in our protocol is due to the fact that a cache holding a line in an blocked owner state cannot respond to write requests (not even persistent write requests). The blocked time lasts while the *ownership acknowledgement* travels to the previous owner and until the *backup deletion acknowledgement*

Table 3. Characteristics of architectures simulated.

4, 8 or 16-Way CMP System	
Processor Parameters	
Processor speed	2 GHz
Max. fetch/retire rate	4
Cache Parameters	
Cache line size	64 bytes
L1 cache:	
Size, associativity	32 KB, 2 ways
Hit time	2 cycles
Shared L2 cache:	
Size, associativity	512 KB, 4 ways
Hit time	15 cycles
Memory Parameters	
Memory access time	300 cycles
Memory interleaving	4-way
Network Parameters	
Topology	2D Torus
Non-data message size	2 flits
Channel bandwidth	64 GB/s
Fault tolerance parameters	
Lost token timeout	20000 cycles
Lost data timeout	6667 cycles
Lost backup deletion acknowledgement	10000 cycles
Lost persistent deactivation timeout	10000 cycles
Token serial number size	2 bits
Token serial number table size	16 entries
Backup buffer size	0, 1, 2 or 4 entries

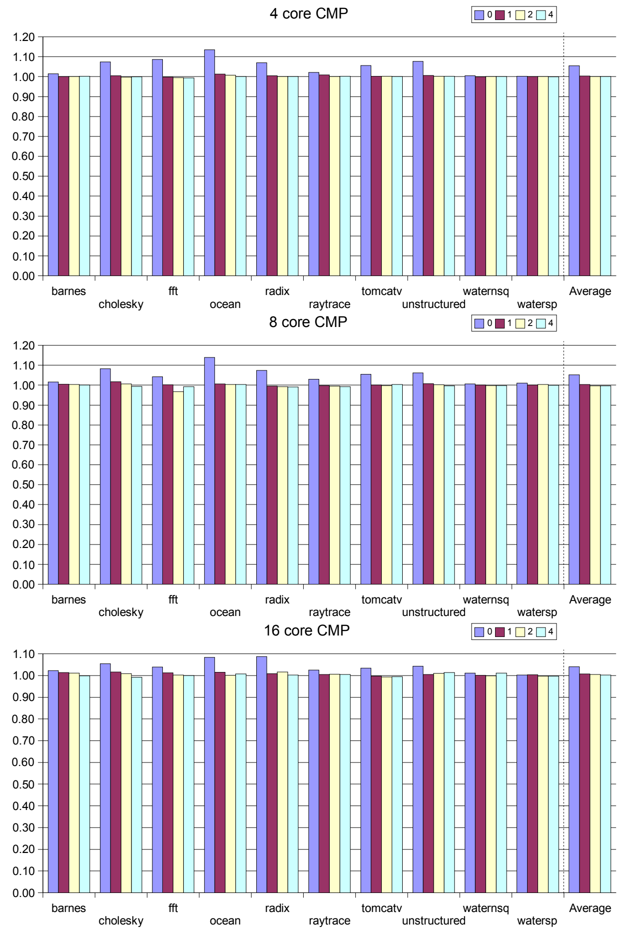


Figure 3. Execution time overhead of our proposal compared to TOKENCMP for several backup buffer sizes.

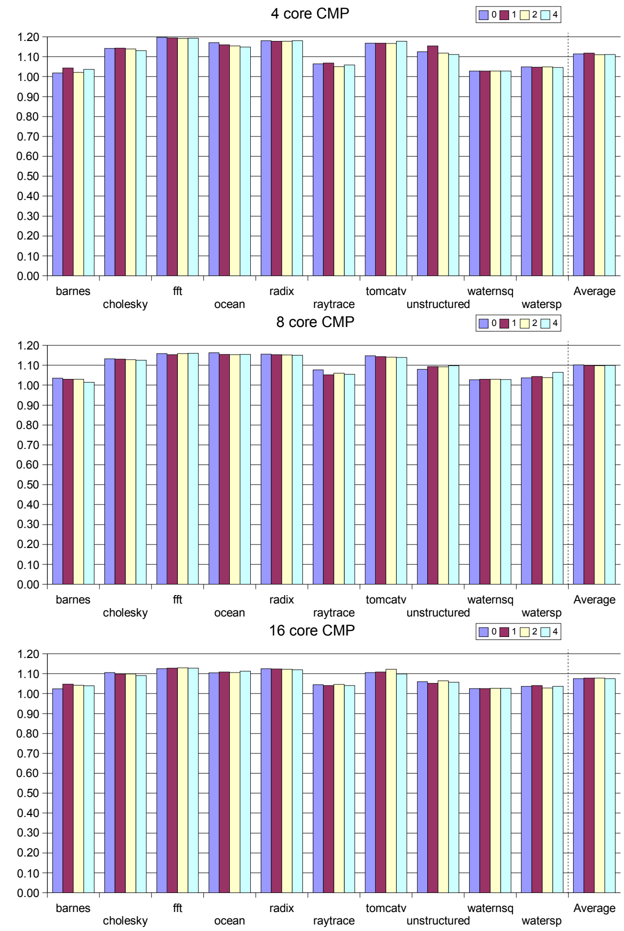


Figure 4. Network traffic overhead of our protocol compared to TOKENCMP.

arrives to the new owner. The results shown in figure 3 suggest that the effect of this overhead in the total execution time is negligible, since the writes that different cores perform on the same line are usually sufficiently separated in time and the new owner can progress its execution as soon as the data is received.

On the other hand, figure 4 shows the network overhead measured as relative increase of bytes transmitted trough the network for the same benchmarks and configurations employed above. As we can see, the network overhead is mostly independent of the size of the backup buffer. As we increase the number of processors, the relative network overhead decreases slightly (12% for 4 processors, 10% for 8 processors and 8% for 16 processors on average). The network overhead is due to the acknowledgements used to guarantee the correct transmission of the owner token and its associated data.

5.3. Measuring the supported fault-tolerance ratio

We have shown that, when a backup buffer with just one entry at each L1 cache is used, our protocol introduces negligible overhead in the execution time and slight network overhead (around 10% more traffic). On the other hand, our proposal is capable of guaranteeing the correct execution of a multi threaded workload on a CMP even in the presence of transient faults. However, the failures and the necessary recovery introduce certain overhead which we would like to keep as small as possible.

Figure 5 shows the execution time overhead of the protocol using a backup buffer with one entry under several message loss rates. Failures rates are expressed in number of messages lost per million of messages that travel through each switch in the network. These failure rates are much higher than realistic failure rates, so these tests overstress the fault tolerance provisions of the protocol. Obviously, the base TOKENCMP protocol (or any previously proposed cache coherence protocol) would not be able to execute correctly any of these tests.

Figure 5. Execution time overhead under several message loss rates.

As we can see, our proposal can support failure rates of up to 250 messages lost per million with an average degradation of 8% in the execution time in a 4-core CMP. In an 8-core system, the same loss rate yields 11% average slowdown, and in a 16-core CMP the degradation is 15%. Hence, our protocol can support a message loss rate of up to 250 messages per million without increasing the execution time more than 15%. As expected, higher failure rates create a higher slowdown in the execution but the fault tolerance measures of the protocol still allow the program to complete correctly, confirming the robustness of such measures. The slowdown depends almost linearly on the failure rate.

Additionally, figure 5 shows that the slowdown observed for a given fault rate increases with the number of processors. This is expected, since greater number of processors means greater number of messages traveling through the network in less time, and hence higher number of faults will occur in less time because the fault rate is independent of time in our experiments. The execution time overhead per fault is approximately the same and depends mainly on the values of the timeouts used to detect faults.

The results shown in this work use long timeouts for detecting faults which have been chosen experimentally to avoid false positives. Using shorter timeout values would reduce the performance degradation in presence of faults at the expense of some false positives which would degrade performance in the fault-free scenario.

6. Conclusions

The rate of transient failures in near future chips will increase due to a number of factors like the increased scale of integration, the lower voltages used and changes in the design process. This will create problems for CMPs and new techniques will be required to avoid errors. One important source of problems will be faults in the interconnection network used to communicate between the cores, the caches and the memory. In this work, we have shown which problems appear in a CMP system with a token based cache coherence protocol when the interconnection network is subject to transient failures and we have proposed a new cache coherence protocol aimed at dealing with those faults that ensures the correct execution of programs while introducing very small overhead. The main recovery mechanism introduced by our protocol is the *token recreation process*, which takes a cache line to a valid state and ensures forward progress after a fault is detected.

We have implemented our protocol using a full system simulator and we have presented results comparing it to a similar cache coherence protocol previously proposed [10] which does not support any fault tolerance but is tuned for performance in CMPs. We have shown that in the fault free scenario the overhead introduced by our proposal is between 5% and 11% when no backup buffer is used, and that using a backup buffer able to store just one cache line in each L1 cache is enough to reduce it to insignificant levels for 4, 8 and 16 way CMPs.

We have checked that our proposal is capable of supporting message loss rates of up to 250 messages lost per million without increasing the execution time more than 15%. The message loss rates used for our tests are several orders of magnitude higher than the rates expected in the real world, hence under real world circumstances no important slowdown should be observed even in the presence of transient failures in the interconnection network.

The hardware overhead required to provide the fault-tolerance is minimal: just a small associative table at each cache to store the *token serial number*, some extra counters at each cache, and a very small backup buffer at each L1 cache.

In this way, our protocol provides a solution to transient failures in the interconnection network with very low overhead which can be easily combined with other fault tolerance measures to achieve full system fault tolerance in future CMPs.

Acknowledgements

This work has been supported by the Spanish Ministry of Ciencia y Tecnología and the European Union (Feder Funds) under grant TIC2003-08154-C06-03. Ricardo Fernández-Pascual has been supported by the fellowship 01090/FPI/04 from the Comunidad Autónoma de la Región de Murcia (Fundación Séneca, Agencia Regional de Ciencia y Tecnología).

References

[1] R. Ahmed, R. Frazier, and P. Marinos. Cache-aided rollback error recovery (CARER) algorithm for shared-memory multiprocessor systems. In *Fault-Tolerant Computing. FTCS-20.*, pages 82–88, June 1990.

[2] M. Banâtre, A. Gefflaut, P. Joubert, C. Morin, and P. A. Lee. An architecture for tolerating processor failures in shared-memory multiprocessors. *IEEE Transactions on Computers*, 45(10):1101–1115, October 1996.

[3] 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.

[4] 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.

[5] 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.

[6] A. Maheshwari, W. Burleson, and R. Tessier. Trading off transient fault tolerance and power consumption in deep submicron (DSM) VLSI circuits. *IEEE transactions on very large scale integration (VLSI) systems*, 12(3):299–311, March 2004.

[7] M. M. Martin. *Token Coherence*. PhD thesis, University of Wisconsin-Madison, December 2003.

[8] M. M. Martin, M. D. Hill, and D. A. Wood. Token coherence: A new framework for shared-memory multiprocessors. *IEEE Micro*, 23(6):108–116, November/December 2003.

[9] 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.

[10] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. K. Martin, and D. 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] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback. In *29th Annual Int'l Symposium on Computer Architecture (ISCA'02)*, pages 111–122, May 2002.

[13] T. Sato. Exploiting instruction redundancy for transient fault tolerance. In *18th International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 547–554, November 2003.

[14] 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 Symposium on Computer Architecture (ISCA'02)*, pages 123–134, May 2002.

[15] D. Sunada, M. Flynn, and D. Glasco. Multiprocessor architecture using an audit trail for fault tolerance. In *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, pages 40–47, June 1999.

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

[17] K. Wu, W. Fuchs, and J. Patel. Error recovery in shared memory multiprocessors using private caches. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):231–240, April 1990.