# Extending the TOKENCMP Cache Coherence Protocol for Low Overhead Fault Tolerance in CMP Architectures

Ricardo Fernández-Pascual, José M. García, *Member*, *IEEE*,
Manuel E. Acacio, and José Duato, *Member*, *IEEE*

**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 (CMPs) 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 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 the GEMS full-system simulator, we compare our proposal against a similar protocol without fault tolerance (TOKENCMP). We show that in the 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 the execution time by more than 15 percent.

**Index Terms**—Fault tolerance, cache coherence, CMP, transient failures, TOKENCMP.

## 1 INTRODUCTION

CHIP multiprocessors (CMPs) [3], [6] 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 be able to design more useful CMPs, 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 [14], 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. There are multiple causes of transient errors, which 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), and radiation from lightning. Any event that upsets the stored or communicated charge can cause soft errors in the circuit output.

Transient failures are much more common than permanent failures [19]. 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 [8]. Since the number of components in a single chip greatly increases, 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 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 acknowledgment messages and other control-flow messages as much as possible.

In this work, we propose a way to deal with the transient failures that occur in the interconnection network of CMPs. We only consider traffic due to accesses to coherent memory and ignore for now accesses to noncoherent memory like memory-mapped I/O. We can assume that

- *R. Fernández-Pascual, J.M. García, and M.E. Acacio are with the Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Campus de Espinardo, Facultad de Informatica, Murcia (30080, Spain). E-mail: {rfernandez, jmgarcia, meacacio}@ditec.um.es.*
- *J. Duato is with the Departamento de Informática de Sistemas y Computadores, Universidad Politécnica de Valencia, Camino de Vera, s/n, Valencia (46022, Spain). E-mail: jduato@disca.upv.es.*

*Manuscript received 30 May 2007; revised 4 Oct. 2007; accepted 30 Oct. 2007; published online 12 Nov. 2007.*
*Recommended for acceptance by R. Iyer.*
*For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2007-05-0168.*
*Digital Object Identifier no. 10.1109/TPDS.2007.70803.*

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. Our proposal adds only those acknowledgments that are absolutely needed and does so without affecting the critical path of most operations.

We attack this problem at the cache coherence protocol level. In particular, we assume that the interconnection network is no longer reliable and extend the TOKENCMP [12] cache coherence protocol to guarantee correct execution in the presence of dropped messages. Our proposal only modifies the coherence protocol and does not add any requirement to the interconnection network, making it applicable to current and future designs. We protect dirty data with acknowledgment messages out of the critical path of cache misses and provide a mechanism for recovering from lost nondata messages. 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. Our protocol does not add a significant execution time overhead but adds a small network traffic overhead (around 10 percent).

There have been several proposals for fault tolerance targeting shared-memory multiprocessors. Most of them use variations of checkpointing and recovery: Ahmed et al. developed the Cache-Aided Rollback Errors Recovery (CARER) [1], Wu et al. [22] developed error recovery techniques using private caches for recovering from processor transient faults in multiprocessor systems, Banâtre et al. proposed the *Recoverable Shared Memory* (RSM), which deals with processor failures on shared-memory multiprocessors using snoopy protocols [2], whereas Sunada et al. proposed the *Distributed Recoverable Shared Memory with Logs* (DRSM-L) [20]. More recently, Prvulovic 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 [16]. At the same time, Sorin et al. presented SafetyNet [18], which has similar objectives but has less overhead, requires custom caches, and can only recover from transient faults. Several commercial systems have been built using fault tolerance techniques and targeting high-availability needs, like Tandem (now HP) NonStop systems [4], and IBM zSeries [17], or those offered by Stratus.

Recently, Meixner and Sorin have proposed an error detection technique for multiprocessors [13] based on token coherence, which can detect any coherence error but provides no recovery mechanism. In addition, Aggarwal et al. presented a mechanism to provide dynamic reconfiguration of CMPs, which enables fault containment in dealing with transient errors and reconfiguration in dealing with hard errors but does not directly address the problems caused by a faulty interconnection network in the coherence protocol.

To the best of our knowledge, there has not been any proposal explicitly dealing with transient faults in the interconnection network of multiprocessors or CMPs from the point of view of the cache coherence protocol. In addition, most fault tolerance proposals require some kind of checkpointing and rollback, whereas our proposal does not require one. Our proposal could be used in conjunction with other techniques that 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 that the use of an unreliable interconnect poses 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 the 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 the execution time by more than 15 percent.

A preliminary and partial version of this paper was presented in [5]. Here, we extend that work with a more extensive evaluation process, including a commercial application (the Apache benchmark) in addition to the suite of scientific benchmarks already considered and better adjustment of the time-outs used for detecting faults. We also consider the out-of-order execution model. Additionally, we have rewritten the description of the cache coherence protocol to make comprehension easier.

The rest of this paper is organized as follows: In Section 2, we present some background about token coherence that is necessary to better understand the rest of this paper. In Sections 3 and 4, we describe the problems posed by an unreliable interconnection network to TOKENCMP and the solutions that we propose. Section 5 presents the evaluation of the overhead introduced by our proposal and its effectiveness in the presence of faults. Finally, in Section 6, we summarize the main conclusions of our work.

## 2 Token Coherence Background

Regarding the cache coherence protocol background, token coherence [9], [10] 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 easily adapt the protocol for different purposes [9] since the performance policy can be modified without worrying about infrequent corner cases, whose correctness is guaranteed by the correctness substrate. Token coherence protocols can avoid both the need for a totally ordered network and the introduction of additional indirection caused by the directory in the common case of cache-to-cache transfers.

The main observation of the token framework is that simple token counting rules can ensure that the memory system behaves in a coherent manner. The following *Token counting* rules are introduced in [9]:

- **Conservation of Tokens.** Each line of shared memory has a fixed number of $T$ tokens associated with it. Once the system is initialized, tokens may not be created or destroyed. One token for each block is the owner token. The owner token may be either clean or dirty.

- **Write Rule.** A component can write a block only if it holds all $T$ tokens for that block and has valid data. After writing the block, the owner token is set to dirty.
- **Read Rule.** A component can read a block only if it holds at least one token for that block and has valid data.
- **Data Transfer Rule.** If a coherence message carries a dirty owner token, it must contain data.
- **Valid-Data Bit Rule.** A component sets its valid-data bit for a block when a message arrives with data and at least one token. A component clears the valid-data bit when it no longer holds any tokens. The home memory sets the valid-data bit whenever it receives a clean owner token, even if the message does not contain data.
- **Clean Rule.** Whenever the memory receives the owner token, the memory sets the owner token to clean.

Considering these rules, we can relate token protocols with traditional MOESI protocols and define each of the states, depending on the number of tokens that a processor has:

| | |
|---|---|
| 0 tokens : | **I**nvalid. |
| 1 to $T-1$ tokens, but not the *owner token* : | **S**hared. |
| 1 to $T-1$ tokens, including the *owner token* : | **O**wned. |
| $T$ tokens, dirty bit inactive : | **E**xclusive. |
| $T$ tokens, dirty bit active : | **M**odified. |

The rules above ensure that cache coherence is maintained but do not ensure forward progress. Token coherence avoids starvation by issuing a persistent request whenever a processor detects potential starvation. Persistent requests, unlike transient requests, which are issued most of the time, are guaranteed to eventually succeed. To ensure this, each token protocol must define how it deals with several pending persistent requests.

In this work, we will consider a distributed persistent request scheme using a persistent request table at each cache as described in [9]. Each processor will be able to activate at most one persistent request at a time by broadcasting a persistent read request activation or a persistent write request activation. Once the request has been satisfied, the processor will broadcast a persistent request deactivation. To avoid livelock, a processor will not be able to issue a persistent request again until all the persistent requests issued by other processors before its last persistent request was deactivated have also been deactivated.

Token coherence provides the framework for designing several particular coherence protocols. The performance policy of a token-based protocol is used to instruct the correctness substrate to move tokens and data through the system. To date, only a few performance policies have been designed; among them is *Token-using-broadcast* (TOKENB), which is a performance policy to achieve low-latency cache-to-cache transfer misses, although it requires more bandwidth than traditional protocols [10]. TOKENCMP [12] is a performance policy similar to TOKENB, 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

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 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 returning 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 checkpointing or a 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 directly affect the interconnection network. 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 correctly arrives to its destination or 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 assumed to be correct because this is by far the most common case, and the CRC check is done in parallel with the initial processing of the message (like accessing the cache tags and MSHR to check the line state).

There are several types of coherence messages that can be lost, which translate into a different impact on the coherence protocol. First, 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 either be persistent or transient requests) in the base protocol require acknowledgment (the caches holding the tokens must respond to the requester), losing a message cannot lead to incoherence.

Losing any other type of message, however, may lead to a 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 that 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 the memory is not updated on each write, but only when it is evicted from the owner cache. In addition, the rules governing the owner token ensure that there is always at least a valid copy of the memory line that travels along with it every time the owner token is transmitted. Thus, losing a message carrying the owner token means that it is possible to totally lose the data.

Finally, while a persistent request is in process, we also have to deal 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 also lead to deadlock situations.

The most obvious solution to the problems depicted above is to ensure that no message is lost while traveling through the interconnection network by means of reliable end-to-end message delivery using acknowledgment messages and sequence numbers in a way similar to TCP [15]. However, this solution has several drawbacks:

- Adding acknowledgments to every message would increase the latency of cache misses, since a cache would not be able to send a message to another cache until it has received the acknowledgment for the previous message.
- That solution would significantly increase network traffic. The number of messages would be at least doubled (one acknowledgment for each message).
- Extra message buffers would be needed to store the messages until an acknowledgment is received in case they need to be resent.

## 4 A Fault-Tolerant Token Coherence Protocol

Instead of ensuring reliable end-to-end message delivery, we have extended the TOKENCMP protocol with fault tolerance measures. To do this, we have added the following states to the traditional MOESI states[1] used by the non-fault-tolerant protocol:

- **Backup (B).** This state is similar to the Invalid state, but the data line is kept in the cache to be used for recovery by the *token recreation process* if necessary. A line will enter a Backup state when the ownership needs to be transferred to a different cache (that is, when leaving the Modified, Owned, or Exclusive state) and will abandon it and become invalid once an *ownership acknowledgment* message is received.
- **Blocked ownership (Mb, Eb, and Ob).** To prevent having more than one backup for a line at any given point in time, a cache that receives the owner token (entering the Modified, Exclusive, or Owned state) will avoid transmitting the owner to another cache until it receives a *backup deletion acknowledgment* message. To achieve 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 ignore external requests to write to that line. Persistent requests will be attended just after receiving the *backup deletion acknowledgment* message.
- **Recreating tokens (R).** A line will enter this state when a fault is detected, and a *token recreation process* is requested.

1. There are also many intermediate states that were not considered in this paper for simplicity, both in the fault-tolerant and non-fault-tolerant protocols.

The main principle that has guided the protocol development has been 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 acknowledgments 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 a correct number of tokens and one and only one valid copy of the data.

As shown in Section 3, 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 something in common that leads to a deadlock. Hence, a possible way to detect faults is by using time-outs for transactions. We use four time-outs for detecting message losses: the "*lost token time-out*" (see Section 4.1), the "*lost data time-out*," the "*lost backup deletion acknowledgment time-out*" (see Section 4.2), and the "*lost persistent deactivation time-out*" (see Section 4.3.2). Notice that all these time-outs, along with the usual retry time-out of the token protocol (except the *lost persistent deactivation time-out*), can be implemented using just one hardware counter, since they do not need to be simultaneously activated. For the *lost persistent deactivation time-out*, an additional counter per processor at each cache or memory module is required. A summary of the time-outs used by our proposal can be found in Table 1.

Since the time to complete a transaction cannot be reliably bounded with a reasonable time-out 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.

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

### 4.1 Dealing with Token Loss

When a processor tries to write to a memory line that has lost a token, it will eventually lead to a time-out and issue a persistent request. In the end, after the persistent request is activated, all the available tokens in the whole system for the memory line will be received by the starving cache. In addition, if the owner token was not

TABLE 1
Time-Outs 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 enters the blocked state. | At the cache that holds the owner token. | 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. |

lost and is not blocked (see Section 4.2), the cache will also receive it, together with the 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 time-out*" 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 time-out should be long enough so that, in normal circumstances, every transaction will be finished before triggering this time-out.[2]

Hence, if the starving cache fails to acquire the necessary tokens within a certain time after the persistent request has been activated, the *lost token time-out* will be triggered. In 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. This process will also take care of false positives of the *lost token time-out*, which could lead to an increase in the total number of tokens and to coherence violations by means of the *token serial number* (see Section 4.4). Notice that the *lost token time-out* 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 immediately satisfied.

### 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 a backup state will not be evicted from the cache until an *ownership acknowledgment* is received, even if every token is sent to other caches. This acknowledgment is sent by every cache in response to a message carrying the owner token. While a line is in a *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.[3] Likewise, when a line enters a backup state, the *lost data time-out* will start and will stop once the backup state is abandoned.

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.[4] Hence, a cache that has received the owner token recently cannot transmit it again until it is sure that the backup copy for that line has been deleted. In this situation, the line enters the *blocked ownership* state. A line will leave this state when the cache receives a *backup deletion acknowledgment*, which is sent by any cache when it deletes a backup copy after receiving an *ownership acknowledgment*. Fig. 1 shows an example of how the owner token is transmitted with our protocol.

The two acknowledgments 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 acknowledgment* 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 immediately serviced after receiving the *backup deletion acknowledgment*. This blocked period could increase the latency of some cache-to-cache transfer misses; however, we have found that it does not have any 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 acknowledgment* is received. We have found that the effect on replacements is much more harmful for performance than the effect on 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, when it needs to be replaced, and when there is enough room in the backup buffer.[5] 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

2. Using a value too short for any of the time-outs used to detect faults would lead to many false positives, which would hurt performance.

3. 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, since it will be transitioned to an intermediate backup and valid state until the *ownership acknowledgment* is received.

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

5. 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.

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 | Lost data timeout, token recreation using backup state |
| 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, token recreation |
| Backup deletion acknowledgement | Deadlock | Lost backup deletion acknowledgement timeout, token recreation |

backup buffer with just one or two entries is enough to practically remove the negative effect of backup states (see Section 5.2). Alternatively, a write-back buffer could achieve the same effect.

#### 4.2.1 Handling the Loss of an Owned Data-Carrying Message or an Ownership Acknowledgment

Losing a message that carries the owner token means that there is a possibility that the only valid copy of the data is lost. However, there is still an up-to-date backup copy at the cache that sent the data-carrying message. Since the data-carrying message does not arrive at its destination, no corresponding *ownership acknowledgment* will be received by the cache, and the *lost data time-out* will be triggered.

If an *ownership acknowledgment* is lost, the backup copy will not be discarded, and no *backup deletion acknowledgment* will be sent. Hence, the backup copy will remain in one of the caches, and the data will remain blocked in the other. Eventually, either the *lost data time-out* or the *lost backup deletion acknowledgment time-out* will also be triggered.

When either time-out is triggered, the cache requests a token recreation process to recover the fault (see Section 4.4). The process can solve both situations: If the *ownership acknowledgment* was lost, the memory controller will send the data that had arrived at 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 the other copies have been invalidated.

#### 4.2.2 Handling the Loss of a Backup Deletion Acknowledgment

When a *backup deletion acknowledgment* is lost, a line will stay in a blocked ownership state. This will prevent it from being replaced or from answering any write request. Both conditions 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 acknowledgment* that has been lost, eventually, a persistent request will be activated for it, and after some time, the *lost token time-out* will be triggered. Hence, the *token recreation process* will be used to solve this case.

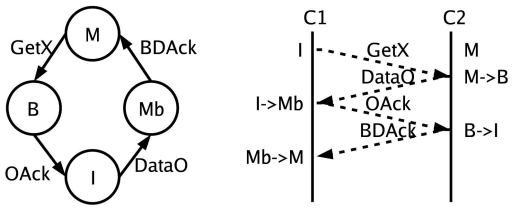


Fig. 1. Transition diagram for the states and events involved in data loss avoidance and message interchange example. Cache C1 broadcasts a transient exclusive request (GetX). C2, which has all the tokens and, hence, is in a *modified* state (M), answers to C1 with a message (DataO) carrying the data and all the tokens, including the owner token. Since C2 needs to send the owner token, it goes to the *backup* state (B) and starts the *lost data time-out*. When C1 receives the DataO message, it satisfies the miss and enters the *modified and blocked* state (Mb), sending an ownership acknowledgment to C2. When C2 receives it, it discards the backup, goes to the *invalid* state (I), stops the *lost data time-out*, and sends a *backup deletion acknowledgment* to C1. Once C1 receives it, it transitions to the normal *modified* state.

To be able to replace a line in a blocked state when the *backup deletion acknowledgment* is lost, we use the *lost backup deletion acknowledgment time-out*. It is activated when the replacement is necessary and deactivated when the *backup deletion acknowledgment* arrives. If it is triggered, 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 indefinitely be kept activated. These situations could lead to starvation.

#### 4.3.1 Dealing with the Loss of a Persistent Request

First, it is important to note that the cache that 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 that 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 be deadlocked.

Since the persistent request has been activated at the starver cache, the *lost token time-out* will eventually be triggered, and the token recreation process will also solve this case.

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 that should be ignored.

#### 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 time-out* when a persistent request is activated and will stop it when it is deactivated. When this time-out is triggered, 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 time-out* 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 time-out* is triggered, it should assume that the deactivation message has been lost and has deactivated 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 (the valid data or backup can also be at the memory). The protocol guarantees that these conditions are true at every moment, despite any message loss.[6] 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 that may have been lost) at the cache that 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 that 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 time-out* will be triggered), 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 presented in Section 2. 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. Thus, 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* that is 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.

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; thus, a counter with a small number of bits should be enough (we use a two-bit wrapping counter). Second, most memory lines will keep the initial serial number unchanged; thus, we only need to store those that have changed it and assume the initial value for the rest. Third, 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 owner token recreation mechanism whenever the table is full and a new entry is needed, because resetting a *token serial number* actually frees up its entry in the table.

Additionally, when a token serial number needs to be reset (either to replace it from the token serial number table or because it has reached the maximum value and needs to be incremented), the interconnect should be drained and the line flushed from all caches to ensure that no old token remains in the network.

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 acknowledgment to the memory. The acknowledgment will also include the 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

6. In particular, these conditions are true if no message has been lost; hence, the *token recreation process* is safe for false positives and can be requested at any moment.

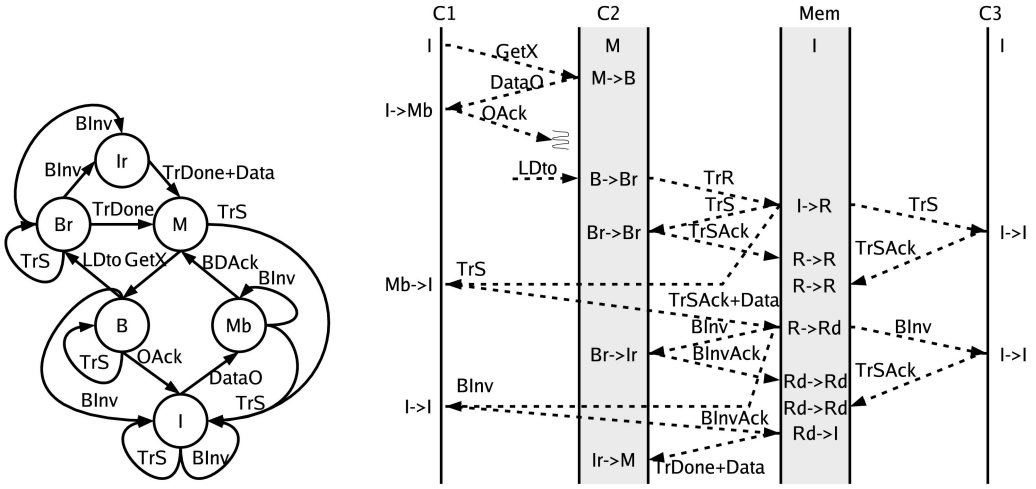


Fig. 2. Transition diagram for the states and events involved in the token recreation process (used in this case to recover from the loss of an ownership acknowledgment). In a transaction like that in Fig. 1, the *ownership acknowledgment* is lost. Hence, C2 keeps the line in a backup state (B). After some time, the *lost data time-out* is triggered (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 acknowledgment (TrSAck) without changing their states, since they are either in an invalid or a backup state. On the other hand, C1 is in a *modified and blocked* state; hence, it returns an acknowledgment with data (TrSAck + Data) and changes its state to *invalid* (I). When the memory receives the acknowledgment with data, it sends a *backup invalidate* message to each cache. C1 and C3 answer with an acknowledgment (BInvAck) without changing their states, while C2 discards its backup data (which could be invalid since C1 may have already written to the cache line), sets its state to *invalid and recreating* (Ir), and answers with an acknowledgment as well. When the memory receives all the acknowledgments, 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).

blocked owner state. However, if the line was held in a backup state, it will remain in that way.

If the memory controller receives an acknowledgment with data, it will send a *backup invalidate* message to all the caches. When receiving that request, the caches will send an acknowledgment 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 acknowledgments (including the acknowledgments for the backup invalidation if it has been requested), it will send a *destruction done* message to the cache that 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 a 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 time-outs 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 time-out* of that cache will eventually be triggered and recover from this fault. Fig. 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 (1,000 in our current implementation) until an acknowledgment is received. Serial numbers are used to detect and ignore duplicates unnecessarily sent.

### 4.5 Hardware Overhead of Our Proposal

First, 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 four times, the counter will wrap) and that 16 entries per processor 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 time-outs employed to detect faults can be implemented using the same hardware already employed to implement the starvation time-out required by token coherence protocols, although the counters may need more bits since the new time-outs are longer. For the *lost persistent deactivation time-out*, it is necessary to add a new counter per processor at each cache and at the memory controller.

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

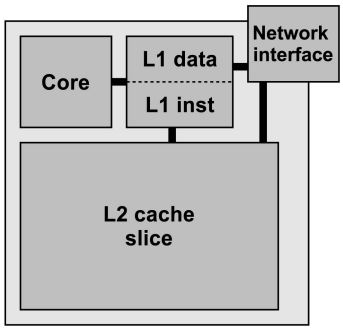


Fig. 3. Diagram of an individual tile.

Our protocol also uses two additional virtual channels with respect to TOKENCMP. One of the channels is used for sending ownership acknowledgment and the other for backup deletion acknowledgment messages. These virtual channels are also used for sending the messages involved in the token recreation process.

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. A backup buffer with just one entry can be effective, 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 the Virtutech Simics [7] functional simulator with the Multifacet GEMS [11] timing infrastructure. GEMS can model both in-order and out-of-order processors using Opal.

We have simulated two likely design points for future CMP systems: a 4-way CMP system with out-of-order cores and a 16-way CMP system with in-order cores. Both configurations are designed as an array of replicated tiles connected over a point-to-point switched network. As shown in Fig. 3, each tile contains a processor, private L1 data and instruction caches, and part of the shared L2 cache. We estimate that the two configurations would require a comparable number of transistors.

Using out-of-order execution does not affect the correctness of the protocol at all and does not have an important 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 [12] protocol and to check its effectiveness when dealing with message losses. TOKENCMP is a token-based coherence protocol without fault tolerance provision, but has been optimized for performance in CMPs.

The most relevant configuration parameters of the modeled systems are shown in Table 3. In particular, the values chosen for the fault-detection time-outs have been experimentally fixed to minimize the performance degradation in the presence of faults while avoiding false positives that would reduce performance in the fault-free case. Using even shorter time-out values would only moderately reduce the performance degradation in the presence of faults but would significantly increase the risk of false positives.

TABLE 3
Characteristics of Simulated Architectures

| 4 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 per core, 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 | 8 bytes |
| Data message size | 72 bytes |
| Channel bandwidth | 64 GB/s |
| **Fault tolerance parameters** | |
| Lost token timeout | 2000 cycles |
| Lost data timeout | 1000 cycles |
| Lost backup deletion acknowledgement | 1000 cycles |
| Lost persistent deactivation timeout | 1000 cycles |
| Token serial number size | 2 bits |
| Token serial number table size | 16 entries |
| Backup buffer size | 0, 1, 2 or 4 entries |

Finally, all the simulations have been conducted using several scientific programs and the Apache HTTP server. Barnes, Cholesky, FFT, Ocean, Radix, Raytrace, Water-NSQ, and Water-SP are from the SPLASH-2 [21] 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. In the case of Apache, we use version 2.2.4 serving static web pages of different sizes. Table 4 shows the input sizes used in the simulations. We have performed several simulations with different random seeds for each benchmark to account for the variability of multithreaded execution; this variability is represented by the error bars in the figures, which enclose the resulting 95 percent confidence interval of the results.

### 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

TABLE 4
Benchmarks and Input Sizes Used in the Simulations

| Benchmark | Input Size |
|---|---|
| Apache | 300 http transactions |
| Barnes | 8192 bodies, 4 time steps |
| Cholesky | tk16.O |
| FFT | 256K complex doubles |
| Ocean | $258 \times 258$ ocean |
| Radix | 1M keys, 1024 radix |
| Raytrace | 10Mb, teapot.env scene |
| Tomcatv | 256 points, 5 iterations |
| Unstructured | Mesh.2K, 5 time steps |
| Water-NSQ | 512 molecules, 4 time steps |
| Water-SP | 512 molecules, 4 time steps |

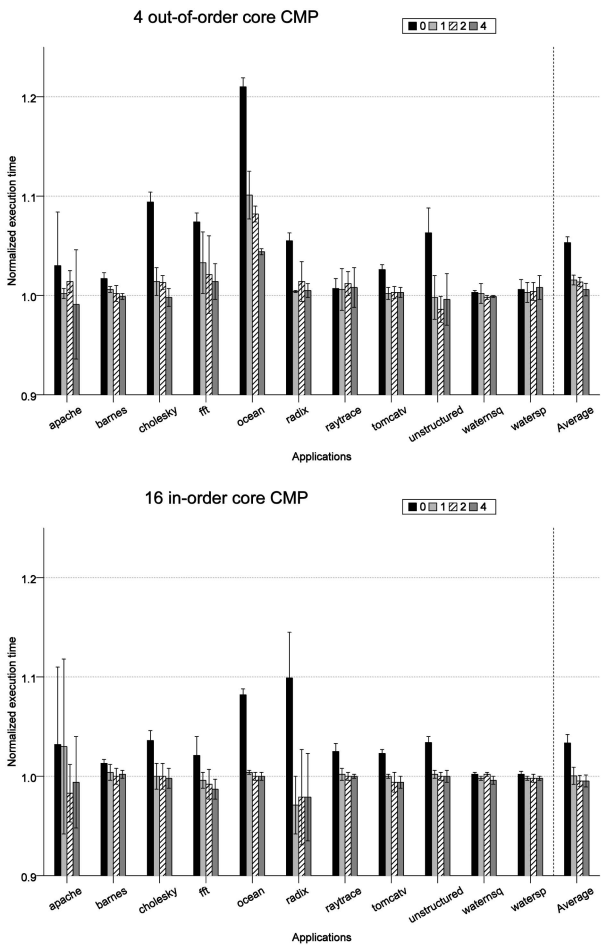


Fig. 4. Execution time overhead of our proposal compared to TOKENCMP for several backup buffer sizes.

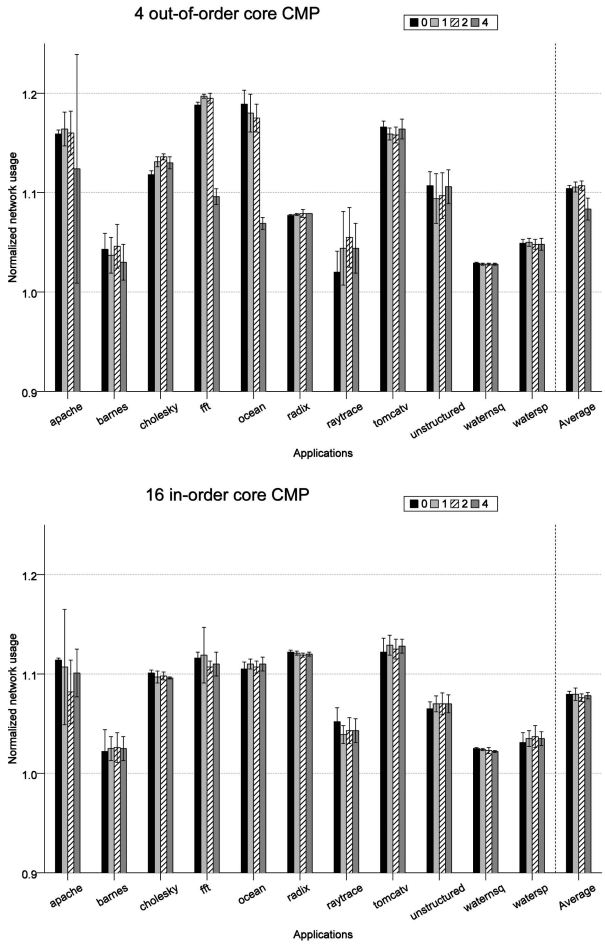


Fig. 5. Network traffic overhead of our protocol compared to TOKENCMP.

lost. As previously explained, the execution time overhead depends on the size of the backup buffer (see Section 4.2). Fig. 4 plots the execution time overhead using different sizes for the backup buffer, including the case of not having a backup buffer at all.

As derived in Fig. 4, without a backup buffer, the overhead in terms of execution time is more than 5 percent, on the average, for the four-core CMP and more than 20 percent for some benchmarks, which we think is not acceptable. The results for 16-core CMPs are also similar. We have found that this slowdown is due to the increased latency of the misses that need a replacement of an owned line first, since the replacement is no longer immediate but has to wait until an *ownership acknowledgment* 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 four-core CMP, a backup buffer of just one entry cuts down the penalty to less than 2 percent, on the average. Likewise, for the 16-core architecture, the slowdown using one entry in the backup buffer is less than 1 percent.

The other potential source of miss latency overhead in our protocol is due to the fact that a cache holding a line in a blocked owner state cannot respond to write requests (not even persistent write requests). The blocked time lasts while the *ownership acknowledgment* travels to the previous owner and until the *backup deletion acknowledgment* reaches the new owner. The results shown in Fig. 4 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, Fig. 5 shows the network overhead measured as a relative increase in bytes transmitted through the network for the same benchmarks and configurations employed above. As shown in our previous work [5], where we simulated a four-way in-order CMP, the relative network overhead slightly decreases as we increase the number of processors (11 percent for 4 processors and 8 percent for 16 processors, on the average). The network overhead is due to the acknowledgments used to guarantee the correct transmission of the owner token and its associated data. On the average, we have found a 10 percent network overhead that represents the cost of extending the TOKENCMP protocol with fault tolerance properties.

### 5.3 Measuring the Supported Fault Tolerance Ratio

We have shown that our protocol introduces negligible overhead in the average execution time and slight network overhead. On the other hand, our proposal is capable of guaranteeing the correct execution of a multithreaded workload on a CMP, even in the presence of transient faults. However, the failures and the necessary recovery

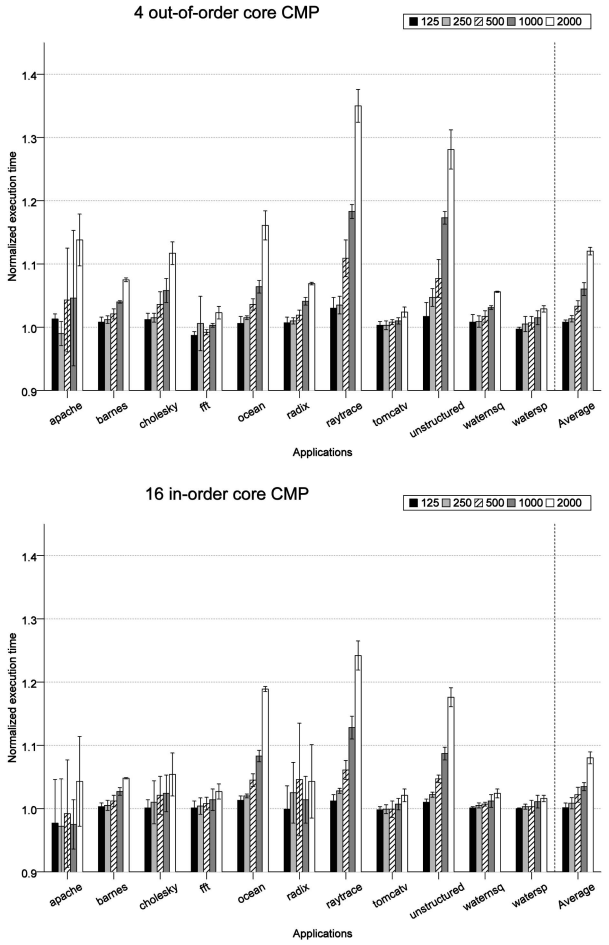


Fig. 6. Execution time overhead under several message loss rates.

introduce a certain overhead that we would want to keep as small as possible.

Fig. 6 shows the execution time overhead of the protocol using a backup buffer with one entry under several message loss rates. Failure 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; hence, 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 correctly execute any of these tests.

As we can see, our proposal can support failure rates of up to 2,000 messages lost per million with an average degradation of 12 percent in the execution time in a four-core CMP. In a 16-core system, the same loss rate yields an 8 percent average slowdown. Hence, our protocol can support a message loss rate of up to 2,000 messages per million without increasing the execution time by more than 12 percent. 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, the extent of this slowdown is very sensitive to the values of the time-outs used to detect message losses. In particular, in our previous work [5], we used very different and much higher time-out values (6,777-20,000 cycles instead of 1,000-2,000) in order to avoid false positives as much as possible. Using those time-outs, the performance degradation in the presence of faults was much higher due to the increased latency to detect a fault and start a recovery process. The new shorter values used for this paper have been experimentally determined so that the false positive rate remains almost zero (hence, the overhead in the absence of faults is almost the same), but the performance degradation in the presence of faults is much lower.

## 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 the 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 (which is an extension of the already proposed TOKENCMP [12]) aimed at dealing with those faults, ensuring 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 the results, comparing this protocol with the original version of TOKENCMP, 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 percent and 20 percent 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 almost insignificant levels for 4- and 16-way CMPs.

We have checked that our proposal is capable of supporting message loss rates of up to 2,000 messages lost per million without increasing the execution time by more than 15 percent. 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 main cost of our proposal is a 10 percent increase in network traffic due to some extra acknowledgment messages. 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.

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

Although this work extends a token coherence-based protocol, the same ideas could be applied to other types of protocols. In fact, we are designing a directory-based fault-tolerant protocol for CMPs with similar characteristics to the one presented in this paper.

## ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their insightful comments and suggestions that have significantly helped improve the final version of this paper. This work has been jointly supported by the Spanish MEC and European Commission FEDER funds under Grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03." Ricardo Fernández-Pascual has been supported by Fellowship 01090/FPI/04 from the Comunidad Autónoma de la Región de Murcia (Fundación Séneca, Agencia Regional de Ciencia y Tecnología).

## REFERENCES

[1] R.E. Ahmed, R.C. Frazier, and P.N. Marinos, "Cache-Aided Rollback Error Recovery (CARER) Algorithm for Shared-Memory Multiprocessor Systems," *Proc. 20th Ann. Int'l Symp. Fault-Tolerant Computing (FTCS '90),* pp. 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 Trans. Computers,* vol. 45, no. 10, pp. 1101-1115, Oct. 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," *Proc. 27th Int'l Symp. Computer Architecture (ISCA '00),* pp. 282-293, June 2000.

[4] D. Bernick, B. Bruckert, P. Del Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "Nonstop Advanced Architecture," *Proc. 2005 Int'l Conf. Dependable Systems and Networks (DSN '05),* pp. 12-21, 2005.

[5] R. Fernández-Pascual, J.M. García, M.E. Acacio, and J. Duato, "A Low Overhead Fault Tolerant Coherence Protocol for CMP Architectures," *Proc. 13th Int'l Symp. High-Performance Computer Architecture (HPCA '07),* pp. 157-168, Feb. 2007.

[6] L. Hammond, B.A. Hubbert, M. Siu, M.K. Prabhu, M. Chen, and K. Olukotun, "The Stanford Hydra CMP," *IEEE MICRO Magazine,* vol. 20, no. 2, pp. 71-84, Mar.-Apr. 2000.

[7] 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,* vol. 35, no. 2, pp. 50-58, 2002.

[8] A. Maheshwari, W. Burleson, and R. Tessier, "Trading Off Transient Fault Tolerance and Power Consumption in Deep Submicron (DSM) VLSI Circuits," *IEEE Trans. Very Large Scale Integration (VLSI) Systems,* vol. 12, no. 3, pp. 299-311, Mar. 2004.

[9] M.M.K. Martin, "Token Coherence," PhD thesis, Univ. of Wisconsin-Madison, Dec. 2003.

[10] M.M.K. Martin, M.D. Hill, and D.A. Wood, "Token Coherence: A New Framework for Shared-Memory Multiprocessors," *IEEE Micro,* vol. 23, no. 6, pp. 108-116, Nov./Dec. 2003.

[11] M.M.K. Martin, D.J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, and D.A. Wood, "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News,* vol. 33, no. 4, pp. 92-99, Sept. 2005.

[12] 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," *Proc. 11th Int'l Symp. High-Performance Computer Architecture (HPCA '05),* pp. 328-339, Feb. 2005.

[13] A. Meixner and D.J. Sorin, "Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures," *Proc. 13th Int'l Symp. High-Performance Computer Architecture (HPCA '07),* pp. 145-156, Feb. 2007.

[14] S.S. Mukherjee, J. Emer, and S.K. Reinhardt, "The Soft Error Problem: An Architectural Perspective," *Proc. 11th Int'l Symp. High-Performance Computer Architecture (HPCA '05),* Feb. 2005.

[15] J.B. Postel, "Transmission Control Protocol," RFC 793, Sept. 1981.

[16] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-Effective Architectural Support for Rollback," *Proc. 29th Ann. Int'l Symp. Computer Architecture (ISCA '02),* pp. 111-122, May 2002.

[17] T.J. Slegel, R.M. Averill III, M.A. Check, B.C. Giamei, B.W. Krumm, C.A. Krygowski, W.H. Li, J.S. Liptay, J.D. MacDougall, T.J. McPherson, J.A. Navarro, E.M. Schwarz, K. Shum, and C.F. Webb, "IBM's S/390 G5 Microprocessor Design," *IEEE Micro,* vol. 19, no. 2, pp. 12-23, 1999.

[18] D.J. Sorin, M.M.K. Martin, M.D. Hill, and D.A. Wood, "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery," *Proc. 29th Ann. Int'l Symp. Computer Architecture (ISCA '02),* pp. 123-134, May 2002.

[19] L. Spainhower and T.A. Gregg, "IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective," *IBM J. Research and Development,* vol. 43, nos. 5/6, pp. 863-873, Sept. 1999.

[20] D. Sunada, M. Flynn, and D. Glasco, "Multiprocessor Architecture Using an Audit Trail for Fault Tolerance," *Proc. 29th Ann. Int'l Symp. Fault-Tolerant Computing (FTCS '99),* pp. 40-47, June 1999.

[21] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA '95),* pp. 24-36, June 1995.

[22] K.L. Wu, W.K. Fuchs, and J.H. Patel, "Error Recovery in Shared Memory Multiprocessors Using Private Caches," *IEEE Trans. Parallel and Distributed Systems,* vol. 1, no. 2, pp. 231-240, Apr. 1990.

**Ricardo Fernández-Pascual** received the MS degree in computer science from the Universidad de Murcia, Murcia, Spain, in 2004. During that year, he joined the Computer Engineering Department as a PhD student with a fellowship from the regional government. Since 2006, he has been an assistant professor in the Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia. His research interests include general computer architecture, fault tolerance, chip multiprocessors, and performance simulation.

**José M. García** received the MS degree in electrical engineering and the PhD degree in computer engineering from the Technical University of Valencia, Valencia, Spain, in 1987 and 1991, respectively. He is currently a professor in the Departamento de Ingeniería y Tecnología de Computadores, the Head of the Research Group on Parallel Computer Architecture, and the Dean of the School of Computer Science, Universidad de Murcia, Murcia, Spain. He has developed several courses on Computer Structure, Peripheral Devices, Computer Architecture, Parallel Computer Architecture, and Multicomputer Design. He specializes in computer architecture, parallel processing, and interconnection networks. His current research interests include high-performance and fault-tolerant coherence protocols for chip multiprocessors (CMPs) and shared-memory multiprocessor systems and high-speed interconnection networks. He has published more than 90 refereed papers in different journals and conference proceedings in these fields. He is a member of the European Network of Excellence on High-Performance and Embedded Architecture and Compilation (HiPEAC) and a member of several international associations like the IEEE and the ACM. He is also a member of some European associations like Euromicro and ATI.

**Manuel E. Acacio** received the MS and PhD degrees in computer science from the Universidad de Murcia, Murcia, Spain, in 1998 and 2003, respectively. In 1998, he joined the Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, where he is currently an associate professor of computer architecture and technology. His research interests include prediction and speculation in multiprocessor memory systems, multiprocessor-on-a-chip architectures, and power-aware cache-coherence protocol design.

**José Duato** received the MS and PhD degrees in electrical engineering from the Universidad Politécnica de Valencia, Valencia, Spain, in 1981 and 1985, respectively. He is currently a professor in the Departamento de Informática de Sistemas y Computadores, Universidad Politécnica de Valencia. He was an adjunct professor in the Department of Computer and Information Science, The Ohio State University, Columbus. His current research interests include interconnection networks, multiprocessor architectures, networks of workstations, and switch fabrics for IP routers. He has published more than 250 refereed papers. He proposed the first theory of deadlock-free adaptive routing for wormhole networks, whose versions have been used in the design of the routing algorithms for the MIT Reliable Router, the Cray T3E supercomputer, the internal router of the Alpha 21364 microprocessor, and the BlueGene/L supercomputer. He is the first author of *Interconnection Networks: An Engineering Approach*, which was coauthored by Prof. Sudhakar Yalamanchili from the Georgia Institute of Technology, Atlanta, and Prof. Lionel Ni from Michigan State University, East Lansing. He served as a member of the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems* and the *IEEE Transactions on Computers*. He has been the general cochair for the 2001 International Conference on Parallel Processing and the program committee chair for the 10th International Symposium on High Performance Computer Architecture (HPCA-10). He has also served as a cochair, a member of the steering committee, the vice chair, or a member of the program committee in more than 40 conferences, including the most prestigious conferences in his area like HPCA, the International Symposium on Computer Architecture, the International Parallel Processing Symposium/Symposium on Parallel and Distributed Processing, the International Conference on Parallel Processing, the International Conference on Distributed Computing Systems, Europar, and the International Conference on High Performance Computing. He is a member of the IEEE.