Dealing with transient faults in the interconnection network of CMPs at the cache coherence level

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

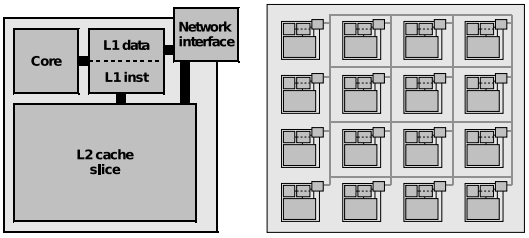
***Abstract*— The importance of transient faults is predicted to grow due to current technology trends of increased scale of integration. One of the components that will be significantly affected by transient faults is the interconnection network of CMPs. To deal efficiently with these faults and differently from other authors, we propose to use fault-tolerant cache coherence protocols that ensure the correct execution of programs when not all messages are correctly delivered. We describe the extensions made to a directory-based cache coherence protocol to provide fault tolerance and provide a modified set of token counting rules which are useful to design fault-tolerant token-based cache coherence protocols. We compare the directory-based fault-tolerant protocol with a token-based fault-tolerant one. We also show how to adjust the fault tolerance parameters to achieve the desired level of fault tolerance and measure the overhead achieved to be able to support very high fault rates. Simulation results using a set of scientific, multimedia and commercial applications show that the fault tolerance measures have virtually no impact on execution time with respect to a non fault-tolerant protocol. Additionally, our protocols can support very high rates of transient faults at the cost of slightly increased network traffic.**

I. Introduction

CHIP Multiprocessors (CMPs) have become the preferred way to effectively take advantage of the increased availability of transistors while keeping design complexity manageable. Further, tiled architectures which are built by replicating several *tiles* comprised by a core, private cache, part of the shared cache and a network interface help in keeping complexity more manageable, scale well to a larger number of cores and support families of products with varying number of tiles. In this way, it seems likely that they will be the choice for future many-core CMP designs [23], [24]. Figure 1(b) shows a 16-core CMP organized by replicating the tile structure shown in figure 1(a).

A main drawback of current technology trends is that, due to the miniaturization and the lower voltages used for power efficiency reasons, the susceptibility of future chips to transient faults will increase. Transient faults [3], [17], also known as soft errors or single event upsets, occur when a component produces an erroneous output but continues working correctly after the event. Any event which upsets the stored or communicated charge can cause soft errors. Typical causes include alpha-particle strikes, cosmic rays, radiation from radioactive atoms which exist in trace amounts in all materials, and electrical sources like power supply noise, electromagnetic interference (EMI) or radiation from lightning.

R. Fernández-Pascual, J.M. García and M.E. Acacio are with the Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia (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 (Spain). E-mail: jduato@disca.upv.es.



(a) Diagram of an individual tile (b) 16-way tiled CMP with 2D mesh interconnection network

Fig. 1. Tiled Chip Multiprocessors.

Reliability is not only required for some critical applications: even for commodity systems reliability needs to be above a certain level for the system to be useful for anything.

In fact, since the number of components in a chip increases and the reliability of each component decreases, it is no longer economical to design new chips and test assuming a worst case reliability scenario. Instead, new designs will target the common case and assume a certain rate of transient faults. Hence, transient faults will have to be handled across all the levels of the system to avoid actual errors. Transient faults are already a problem for memories and caches which routinely use error detection and correction codes (ECC) to deal with them. Other parts of the system will need to use fault tolerance techniques to deal with transient faults as their frequency increases.

One of the components which will be affected by transient faults in a CMP is the interconnection network. It occupies a significant part of the chip real estate and is critical to the performance of the system. It handles the communication between the cores and caches, which is done by means of a cache coherence protocol. Communication is usually very fine-grained (at the level of cache lines) and requires very small and frequent messages. Hence, to achieve good performance the interconnection network must provide very low latency.

Fault tolerance in the interconnection network has traditionally been provided at the network level. Several proposals on how to do this are mentioned in section II. Ensuring the reliable transmission of all messages through the network imposes significant overheads in latency, power consumption and area. Differently from other authors, we propose to deal with transient faults in the interconnection network of CMPs at the level of the cache coherence protocol. This allows for more flexibility to design a high-performance on-chip network which can be unreliable. At the same time, the higher level information available to the coherence protocol enables it to achieve fault tolerance with lower overhead, avoiding acknowledgment messages in most cases, protecting only those messages which are critical to the correctness of the protocol. The few necessary acknowledgments are sent out of the critical path of coherence transactions to

minimize the effect of fault tolerance on performance.

In a previous work [5], [6], we showed that a token-based [12] coherence protocol can be extended to tolerate transient faults. Unfortunately, token coherence is not the cache coherence protocol of choice in current CMP proposals.

Tiled CMPs implement a point-to-point interconnection network which is best suited for directory-based cache coherence protocols. Furthermore, compared with snoopy-based or token-based protocols which usually require frequent broadcasts, directory-based ones are more scalable and energy-efficient. In this work, we apply some of the lessons learned there to guarantee fault tolerance in a directory-based cache coherence protocol.

A fault-tolerant cache coherence protocol needs to provide the following things: a fault detection mechanism, a fault recovery mechanism, and a mechanism to ensure that data is never lost or corrupted.

In both protocols, fault detection is achieved by means of a number of timeouts which detect deadlocks caused by discarded messages. This fault detection mechanism is reliable and valid for every coherence protocol where a discarded message can be either harmless or lead to a deadlock in the same or a subsequent memory transaction. This is the case of TOKENCMP (the non fault-tolerant token-based cache coherence protocol), where discarded transient requests are harmless and the rest of message types lead to deadlock; and in the case of DIRCMP (the non fault-tolerant directory-based cache coherence protocol) where every discarded message leads to a deadlock. However, not all cache coherence protocols have this property: for example, some protocols do not require acknowledgments for invalidation messages, hence discarding an invalidation message would lead to an incoherence instead of a deadlock. The number and precise function of timeouts depend on the way that each protocol works. Also, both protocols ensure the integrity of data when it travels through the network by means of explicit acknowledgments out of the critical path of cache misses. However, the recovery mechanisms used by each protocol are different.

Our two proposals do not add any requirements to the interconnection network so they are applicable to current and future designs. Actually, since the network does not need to guarantee correct delivery of every message, we expect that it could take advantage of a more aggressive design to reduce latency even at the cost of dropping a few messages, improving overall performance. Also, our proposals could be used in conjunction to other techniques which provide fault tolerance to individual cores and caches in the CMP to achieve full coverage against transient faults inside the chip.

Some parts of this work were presented in a preliminary version in [8] and [7]. The main contributions of this paper are:

- A cache coherence protocol which extends a standard directory-based coherence protocol with fault-tolerant measures and assumes a point-to-point unordered interconnection network. Unlike the protocol presented in [8], which assumed a point-to-point ordered interconnection network, this protocol can work with reconfigurable interconnection networks or with adaptive routing. These characteristics are highly desirable for being able to deal with permanent errors both in the interconnection network and in the processor cores and for other purposes beyond fault-tolerance.
- A modified set of token counting rules that ensure reliable ownership transference and formalize the modifications to the token coherence framework required by the protocol presented in [6].
- A comparison of the two fault-tolerant cache coherence protocols under a common framework and using a wider selection of benchmarks than in [7]; including scientific, multimedia and commercial applications. In this new evaluation, we explain how to adjust the fault detection timeouts and how to trade between the overhead introduced by request serial numbers and the desired rate of fault tolerance.
- We explain the fault model used in our previous works and extend it to model faults when they happen in bursts (instead of isolated).

The rest of the paper is organized as follows: In section II we review relevant previous work. The base architecture and cache coherence protocols are described in section III. Section IV explains our fault model. Sections V and VI explain the fault-tolerant coherence protocols. A performance evaluation is done in section VII. Finally, section IX concludes the paper.

II. RELATED WORK

Fault tolerance for multiprocessors has been thoroughly studied in the past. Most proposals deal with transient errors by means of checkpointing and recovery. For example, Pruvlovic *et al.* presented ReVive [21], which performs checkpointing, logging and memory based distributed parity protection with low overhead in error-free execution and is compatible with off-the-self processors, caches and memory modules. At the same time, Sorin *et al.* presented SafetyNet [22] which aims at the same objectives but has less overhead and uses custom caches.

Recently, Meixner *et al.* proposed error detection techniques [15], [16] for multiprocessors which can detect errors that lead to memory consistency or coherence violations, but do not provide any recovery mechanism. Also, Aggarwal *et al.* [1] proposed a mechanism to provide dynamic reconfiguration of CMPs which enables fault containment and reconfiguration, but does not directly address the problems caused by a faulty interconnection network in the coherence protocol.

Instead of ensuring fault tolerance at the cache coherence protocol level like we propose, a more straightforward way to solve the problem of transient faults in the on-chip interconnection network is making the network itself fault-tolerant. There are several proposals [2], [4], [19] exploring this approach.

Usually, these proposals achieve fault tolerance using error detection or correction codes and message retransmission [18]. Both end-to-end and switch-to-switch retransmission schemes are possible. In an end-to-end scheme, error detection codes are added to messages and network interfaces have additional message buffers to store messages which have been transmitted until it receives an acknowledgment signal. The messages can be retransmitted when a negative acknowledgment signal is received or when a timeout triggers at the sender. If timeouts are used, messages require sequence numbers to detect duplicates. Switch-to-switch schemes are similar, but error detection and retransmission hardware is added at each switch instead of at each network interface. Error correction can be done at message level or flit level also.

Other way to provide fault tolerance at the interconnection network level is using fault-tolerant routing techniques [20] relying on flooding. These techniques trade increased network traffic (and power consumption) for reliability.

A system could combine both interconnection level fault tolerance measures and cache coherence protocol level ones. This way, the fault rate that the coherence protocol would need to support would decrease, while part of the reliability of the interconnection network could be traded for increased performance.

Also, most interconnection network fault tolerance proposals rely on adding a certain amount of fault resiliency to the network interfaces and/or network switches by means of hardware redundancy or VLSI transient fault mitigation techniques to avoid single points of failure. Our approach could also benefit from these measures, although we have not identified any component of the interconnection network that would require it to guarantee correctness.

However, ensuring the reliable transmission of all messages through the network limits the flexibility of the network design and imposes significant overheads in latency, power consumption and area. Those overheads are constant per message since the interconnection network lacks information about the meaning of the communication that is taking place. In contrast, ensuring fault tolerance at the higher lever of the cache coherence protocol allows for more flexibility to design a high-performance on-chip network which can be not totally reliable, but have better latency and power consumption in the common case. Since the protocol has more information, it can ensure the reliable retransmission of those few messages that carry owned data and could cause data loss; achieving better performance overall as long as enough messages are transmitted correctly through the network.

III. Base CMP architectures and protocols

In this work, we assume a single CMP system built using a number of tiles [23]. Each tile contains a processor core, private L1 data and instruction caches and a bank of the L2 cache. The L2 cache is logically shared by all cores but it is physically distributed among tiles. Each tile has its network interface to connect to the on-chip interconnection network. We assume in-order processors since that seems the most reasonable approach to build power-efficient CMPs with many cores.

While we have assumed a tiled architecture and in-order processors, these choices are not constraints of the evaluated coherence protocols, whose functionality and correctness is not affected if out-of-order cores are used or a different arrangement is used instead of tiles.

We consider two base architectures whose main difference lies in the coherence protocol: one uses a token-based cache coherence protocol called TokenCMP [14] and another uses a more traditional directory-based protocol adapted for CMP systems that we will refer to as DirCMP.

TokenCMP is a cache coherence protocol based on token coherence which targets hierarchical multiple CMP systems and is well suited for single CMPs. Token coherence provides a framework for defining several particular coherence protocols by separating the protocol definition in a correctness substrate and a performance policy which define how the nodes exchange a fixed number of tokens among them. Most requests are *transient requests* which, in the case of TokenCMP and most other token coherence protocols, are sent using broadcast to all other nodes with no ordering guarantees and without even a guarantee of being successfully satisfied. *Token counting* rules ensure that coherency is maintained while *persistent requests* ensure forward progress by providing serialization when races between transient requests are detected. TokenCMP uses a performance policy similar to TokenB (*Token-using-broadcast*) with a distributed arbitration scheme for persistent requests.

DirCMP is a traditional MOESI-based directory cache coherence protocol which uses an on-chip directory to maintain coherence between several private L1 caches and a shared non-inclusive L2 cache. It uses a directory cache in L2 and the L2 effectively acts as the directory for the L1 caches. It uses per line busy states to defer requests to lines with outstanding requests. Hence, the directory will attend only one request for each line at the same time. Also, it uses three-phase writebacks to coordinate writebacks and the rest of requests.

The two base protocols implement a migratory sharing optimization in which a cache holding a modified cache line invalidates its copy when responding to a request to read that line (*GetS*), thus granting write permission to the requesting cache. This optimization substantially improves performance of many workloads with read-modify-write sharing behavior.

IV. Fault model

As mentioned in the introduction, a transient fault in the interconnection network of a CMP can have a number of causes. For example, any event that changes the value stored in a flip-flop which is part of a buffer or that affects the signal transmitted through a wire would cause a transient error. The actual effect of these errors is hard to predict, but we can assume that one or more messages are either corrupted or misrouted as the final consequence.

Corrupted messages will be discarded upon reception. This can be achieved by means of using an error detection code (CRC) in each message. The particular error detection code employed for this purpose is out of the scope of this paper, but we assume that all corrupted messages are detected and discarded[1]. In addition to checking the error detection code, nodes also check that they are the intended recipient of a message before responding. This way, a misrouted message that arrives uncorrupted to the wrong destination will be discarded. Even if a misrouted message were not detected as such, the protocols would handle it gracefully except for invalidation requests that could lead to incoherence in some cases.

Hence, from the point of view of the coherence protocol we assume that all errors cause the loss of any affected messages. That is, in our fault model we assume that the interconnection network will either deliver a message correctly or not at all. We also assume that caches and memories are protected by means of ECC.

In our evaluation we consider several fault rates expressed as “number of corrupted messages per million of messages that travel through the network”. This rate measures the probability that every message has of being affected by a transient fault while it is in the network. We consider two ways of distributing faults in time: in the first one faults are distributed uniformly among messages, while in the second one faults affect messages in bursts of a constant size.

V. A fault-tolerant directory coherence protocol

From now on, we consider a CMP system whose interconnection network is not reliable due to the potential presence of

[1]With our protocols, an undetected corrupted message could lead to incoherence or silent data corruption in some cases, but never to deadlock.

TABLE I
MESSAGE TYPES USED BY DIRCMP.

Type	Description
GetX	Request data and permission to write.
GetS	Request data and permission to read.
Put	Sent by the L1 to initiate a write-back.
WbAck	Sent by the L2 to let the L1 actually perform the write-back. The L1 will not need to send the data.
WbAckData	Sent by the L2 to let the L1 actually perform the write-back. The L1 will need to send the data.
WbNack	Sent by the L2 when the write-back cannot be attended (probably due to some race) and needs to be reissued.
Inv	Invalidation request sent to invalidate sharers before granting exclusive access. Requires an ACK response.
Ack	Invalidation acknowledgment.
Data	Message carrying data and granting read permission.
DataEx	Message carrying data and granting write permission (invalidation acknowledgments may still be pending).
Unblock	Informs the L2 that the data has been received and the sender is now a sharer.
UnblockEx	Informs the L2 that the data has been received and the sender has now exclusive access to the line.
WbData	Write-back containing data.
WbNoData	Write-back containing no data.

transient faults. In the rest of this section we explain the fault tolerance mechanisms of FTDIRCMP in detail, as an example of how to add fault tolerance to a cache coherence protocol.

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

Table I shows a simplified list of the main types of messages used by DIRCMP and a short explanation of their main function.

Directory protocols used in most cache coherent non-uniform memory access machines (cc-NUMAs) usually assume that the network is point-to-point unordered. That is, two messages sent from a node to another can arrive in a different order than they were sent. Unlike our previous work, the version of the FTDIRCMP protocol presented here does support unordered networks. This improvement requires the addition of a new timeout and two new message types (see section V-F).

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

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

Thanks to the fact that every message lost in DIRCMP leads to a deadlock, FTDIRCMP can use timeouts to detect potentially lost messages. It uses a number of timeouts to detect faults and start corrective measures. Table III summarizes these timeouts.

TABLE II
ADDITIONAL MESSAGE TYPES USED BY FTDIRCMP.

Type	Description
AckO	Ownership acknowledgment.
AckBD	Backup deletion acknowledgment.
UnblockPing	Asks confirmation that a cache miss is still in progress.
WbPing	Asks confirmation that a writeback is still in progress.
WbCancel	Confirms that a previous writeback has already finished.
OwnerPing	Asks confirmation of ownership.
NackO	Negative response to an OwnerPing.

Usually, when a fault occurs and a timeout triggers, FTDIRCMP reissues the request using a different serial number. These reissued requests need to be identified as such by the node that answers to them and not be treated like an usual request. In particular, a reissued request should not wait in the incoming request buffer to be attended by the L2 or the memory controller until a previous request is satisfied, because that previous request may be precisely the older instance of the request that is being reissued. Hence, the L2 directory needs to remember the blocker (last requester) of each line to be able to detect reissued requests. This information can be stored in the Miss Status Holding Register (MSHR) table or in a dedicated structure for the cases when it is not necessary to allocate a full MSHR entry.

A. Reliable data transmission

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

There is always one owner node[2] for each line which is responsible of sending data to other nodes to satisfy read or write requests or to perform writeback when the data is modified.

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

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

- **Backup (B)**: This state is similar to the Invalid (I) state, but the data is kept in the cache to be used for potential recovery (that is, when leaving the Modified, Owned or Exclusive states) and will abandon it once an *ownership acknowledgment* is received.
- **Blocked ownership (Mb, Eb and Ob)**: To prevent having more than one backup for a line at any given point in time, which is important to be able to recover in case of a fault, a cache that acquires ownership (entering the Modified, Owned or Exclusive states) will avoid transmitting the ownership to another cache until it receives a *backup deletion acknowledgment* message from the previous owner.

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

TABLE III
SUMMARY OF TIMEOUTS USED IN FTDIRCMP.

Timeout	When is it activated?	Where is it activated?	When is it deactivated?	What happens when it triggers?
Lost request	When a request is issued.	At the requesting L1 cache.	When the request is satisfied.	The request is reissued with a new serial number.
Lost unblock	When a request is answered (and writebacks).	At the responding L2 or memory.	When the unblock / write-back message is received.	An *UnblockPing* / *WbPing* is sent.
Lost data	When owned data is sent through the network.	At the node that sends owned data.	When the *AckO* message is received.	An *OwnerPing* is sent.
Lost backup deletion acknowledgment	When the *AckO* message is sent.	At the node that sends the *AckO*.	When the *AckBD* message is received.	The *AckO* is reissued with a new serial number.

For achieving this, we have added blocked versions of the Modified, Exclusive and Owned states. While a line is in one of these states, the cache will not attend external requests to that line which require ownership transference.

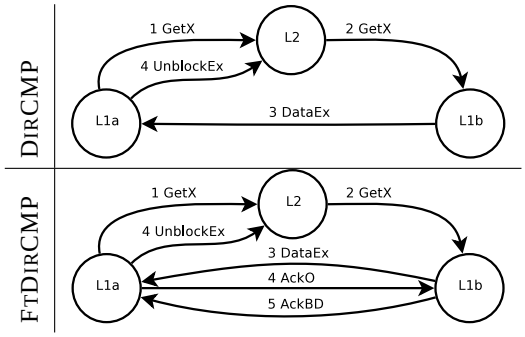
Using the states described above, the transmission of owned data between two nodes works as follows:

1) When a node sends owned data to another node, it does not transition to an *Invalid* state. Instead, it enters a *Backup* state in which the data is still kept for recovery, although no read or write permission on the line is retained. Depending on the particular case, the data may be kept in the same cache block, in a backup buffer [5] or in a writeback buffer. The cache will keep the data until it receives an *ownership acknowledgment*, which can be received as a message by itself or piggybacked along with an *UnblockEx* message.
2) When the data message is received by the new owner, it sends an *ownership acknowledgment* to the node that sent the data. Also, it does not transition to an M, O or E state. Instead it enters one of the blocked ownership states (Mb, Eb or Ob) until it receives the *backup deletion acknowledgment*. While in these states, the node will not transfer ownership to another node. This ensures that there is never more than one backup copy of the data. However, at this point the node has received the data (and possibly write permission to it) and the miss is already satisfied. The *ownership acknowledgment* will carry a serial number also, which can be the same than the data carrying message just received.
3) When the node that sent the data receives the *ownership acknowledgment*, it transitions to an *Invalid* state and sends a *backup deletion acknowledgment* to the other node with the same serial number as the received ownership acknowledgment.
4) Finally, once the *backup deletion acknowledgment* is received, the node that received the data transitions to an M, O or E state and can now transfer the ownership to another node if necessary.

Figure 2 shows an example of how a cache-to-cache transfer miss which requires ownership change is handled in FTDIRCMP and compares it with DIRCMP.

The ownership acknowledgment can be piggybacked in the *UnblockEx* message when the data is sent to the requesting L1 by the L2 (or to L2 by the memory). In that case, only an extra message (the backup deletion acknowledgment) needs to be sent. An example of this situation can be seen in figure 3.

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



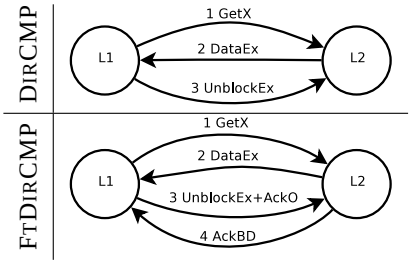
Initially, for both protocols, L1b has the data in modifiable (M), exclusive (E) or owned[3] (O) state and L1a requests write access to L2 (1) which forwards the request to L1b (2). In DIRCMP, L1b sends the data to L1a (3) and transitions to invalid state. Subsequently, when L1a receives the data, it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

Fig. 2. Message exchange for a cache-to-cache write miss.

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

1) Optimizing ownership transference from memory to L1 caches: The rules explained above ensure the reliable transmission of owned data in all cases without adding any message to the critical path of cache misses in most cases. However there are potential performance problems created by the blocked ownership states, since a node (L1 cache, L2 cache bank or memory controller) cannot transfer the recently received owned data until the *backup deletion acknowledgment* message is received.

This is not a problem when the data is received by an L1 cache since the node can already use the data while it waits for said acknowledgment. However, in the case of L2 misses, the L2 cannot answer the L1 request immediately after receiving the data from memory because, according to the rules described above, it first needs to send an *ownership acknowledgment* to memory and wait for the *backup deletion acknowledgment*. Hence, in the case



In both protocols, the L1 cache sends a *GetX* message (1) to the L2, which has the data in an exclusive state (M or E). In DIRCMP, when the L2 receives the request, it sends the data to L1 (2) which, once it receives it, transitions to modificable (M) state and answers to L2 with an *UnblockEx* message. In FTDIRCMP, when the L2 receives the request, it sends the data to the L1 (2) but it also keeps a backup copy of the data. When the L1 receives the data, it transitions to blocked ownership and modifiable (Mb) state and answers with a message (3) which serves both as the *UnblockEx* and as the *AckO* messages used in figure 2. The backup copy in L2 will be kept until the *UnblockEx+AckO* message send by L1 is received and the *AckBD* message (4) is sent to the L1, which transitions to the modifiable (M) state when receives it.

Fig. 3. Message exchange for an L1 write miss that hits in L2.

of L2 misses, these rules would add two messages in the critical path of misses.

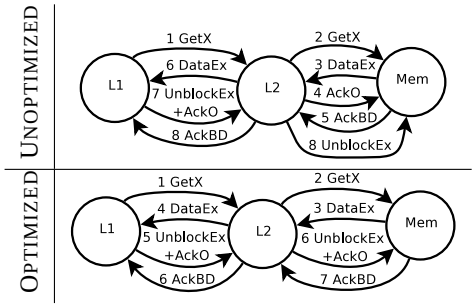
To avoid increasing the latency of L2 misses, we relax the rules in these cases. We allow the L2 to send the data directly to the requesting L1 just after receiving it, keeping a backup until it receives the ownership acknowledgment from the L1. In fact, the L2 does not send the ownership acknowledgment to memory until it receives it from the L1 (most times piggybacked on an unblock message) since this way we can piggyback it with an *UnblockEx* message. Figure 4 shows an example of how an L2 miss would be resolved without and with this optimization.

To implement this behavior, we modify the set of states for the L2 cache so that a line can be either internally blocked or externally blocked, or both (which would correspond to the blocked states already described).

A line enters an externally blocked state when the L2 receives data from memory and leaves it when it receives the backup deletion acknowledgment from memory. While in one of those states, the L2 cannot send the data to the memory again[4], but it can send it to an L1 cache keeping a backup until the respective ownership acknowledgment is received. This ensures that there is at most one backup of the data out of the chip, although there may be another in the chip. This is enough to guarantee correctness in case of faults.

Conversely, a line enters an internally blocked state when the L2 cache receives data from an L1 cache and leaves it when the corresponding backup deletion acknowledgment is received. While in an internally blocked state, ownership cannot be transferred to another L1 cache.

[4]In a multiple CMP setting, it would not be able to send it to other L2 in different chips either. In other words, the ownership of the line cannot leave the chip.



In both cases, the L1 sends a *GetX* message (1) to L2 which, since it does not have the data, forwards (2) it to the memory controller. The memory controller fetches the data and sends it to L2 using a *DataEx* message (3). Now, in the unoptimized case, when the L2 receives the owned data, it sends an ownership acknowledgment (4) to the memory controller and waits for the backup deletion acknowledgment (5) before answering to L1 with the data (6). Once L1 receives the data, it sends a message to L2 (7) carrying the ownership acknowledgment and the unblock. When the L2 receives this message, it will send a backup deletion acknowledgment to L1 and an unblock message to the memory controller (8). On the other hand, in the optimized version, when the L2 receives the *DataEx* (3) message from memory, it sends another *DataEx* message (4) to L1. Notice that now the critical path of the miss requires only 4 hops instead of 6. Once the L1 has received the data, it will send a message (5) with the unblock and the ownership acknowledgment to L2 which will then send the unblock to memory and the backup deletion acknowledgment to L1 (6). Finally, when the memory controller receives the ownership acknowledgment, it will answer with a backup deletion acknowledgment message (7).

Fig. 4. L2 miss optimization of ownership transference.

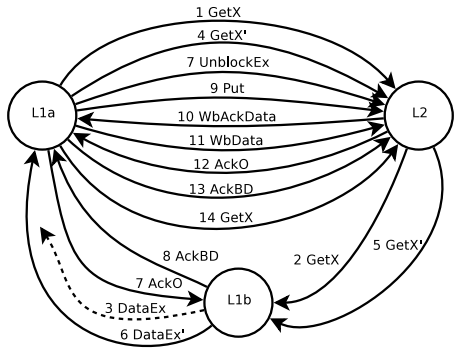
B. Request serial numbers

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

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

Every request and every response message carries a serial number. Request serial numbers are chosen by the L1 cache that issues the request (or by the L2 in case of writebacks from L2 to memory). Responses or forwarded requests will carry the serial number of the request that they are answering to. When a request is reissued, it will be assigned a new serial number which will allow to distinguish between responses to the old request and responses to the new one.

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



L1a makes a request (1) to L2 which forwards it to L1b (2). L1b sends the data (3) to L1a, but this message gets delayed in the network for such a long time that the *request timeout* triggers and L1a reissues the request (4) which is forwarded again to L1b (5) which has a backup copy of the data and resends it (6). This time, the message arrives to L1a which sends (7) an unblock message to L2 and an ownership acknowledgment to L1b. L1b answers with a (8) backup deletion acknowledgment to L1a. After modifying the data, L1a performs a writeback to L2 (messages 9 to 13) and after that, it issues another request (14). If the first data message (3) arrives at this moment and is not discarded using its serial number, it would allow L1a to use the old data.

Fig. 5. Transaction where request serial numbers avoid using stale data.

number or from an unexpected sender. This information needs to be updated when a reissued request arrives. Discarding any message in FTDIRCMP is always safe (even if it could be not strictly necessary in some cases) since the protocol already has provisions for lost messages of any type.

Figure 5 show a case where not using request serial numbers to discard a message that arrives too late would lead to incoherency or using stale data. The example assumes that the interconnection network is not point-to-point ordered, but similar situations are also possible on point-to-point ordered networks [8].

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

C. *Faults detected by the* lost request timeout

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

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

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

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

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

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

D. *Faults detected by the* lost unblock timeout

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

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

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

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

Unblock messages are also exchanged between the L2 and the memory controller in an analogous way. Hence, FTDIRCMP uses

an unblock timeout and *UnblockPing* in the memory controller too.

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

E. Faults detected by the lost backup deletion acknowledgment timeout

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

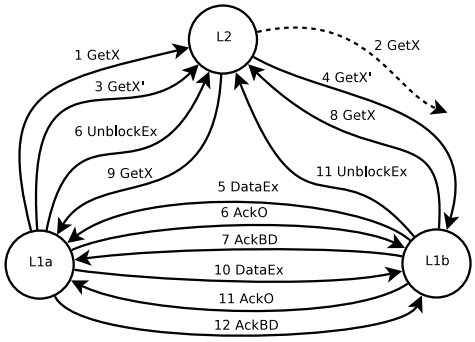
For these reasons, we intoduce the *lost backup deletion acknowledgment* timeout which is started when an ownership acknowledgment is sent and is stopped when the backup deletion acknowledgment arrives. This way, it will trigger if any of these acknowledgments is lost or arrives too late. When it triggers,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

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

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

F. Faults detected with the lost data timeout

The rules described in section V-A guarantee that when an owned data carrying message is lost or discarded due to a wrong serial number, the data will always be in backup state in some node. Usually, when this happens the node that requested the data in the first place will reissue the request after its *lost request timeout* triggers and the data will be resent using the backup copy. Alternatively, if the data ownership transference was due to a writeback, the *lost unblock timeout* will trigger in the L2 (or memory) and the data will be resent when the *WbPing* message is received.



L1a makes a request (1) to L2 which forwards it to L1b. The forwarded message (2) gets delayed in the network and hence the *lost request timeout* triggers in L1a. When this happens, L1a reissues the request (3) which is forwarded again by L2 (4). This time, the request arrives to L1b which sends the data (5) to L1a. When L1a receives the data, it sends (6) an unblock message to L2 and an ownership acknowledgment to L1b which responds with a backup deletion acknowledgment (7) to L1a. Later, L1b makes a new request which is handled in a similar way (messages 8 to 12), so it has the only copy of the data again with exclusive access. If the first forwarded request (2) arrives now to L1b, it will send the data to L1a which will discard it (since it does not expect a response with that serial number anymore). In this situation no node will have the data nor expect it, so future accesses to the line would cause a deadlock.

Fig. 6. Transaction where the lost data timeout detects data which has been wrongly discarded.

In the previous situations, the fault was detected (either with the *lost request timeout* or the *lost backup deletion acknowledgment timeout*) because there was still some node which expected the data to arrive and for that reason had a timeout enabled to detect the situation.

However, if request messages can be reordered while traveling through the network, it can happen that the owned data is sent to some node which does not expect the data and hence will discard it. Since in that case the data will be kept only in backup state in the sender node, no node will be able to access it and this will lead to a deadlock the next time that the line is accessed.

To be able to detect this situation, we have added the *lost data timeout* which is started whenever an owned data carrying message is sent and stopped once the ownership acknowledgment is received. The *lost data timeout* is not activated when the *unblock timeout* is activated in the same node, since the latter can detect the same faults too. This timeout is not necessary if the network guarantees point-to-point ordering of messages.

Figure 6 shows a case where a message carrying owned data is discarded by a node which receives it unexpectedly and the *lost data timeout* is needed to detect the situation.

When the *lost data timeout* triggers, an *OwnerPing* message is sent to the node that was sent the data before. Upon receiving this message, a node will react as described below:

- If the node does not have the ownership of the line, it will answer with a *NAckO* message with the same serial number than the received ping message. Additionally, if it has a pending request for that address, it should reissue it with a new serial number to avoid gaining ownership after sending the *NackO* due to some data message currently delayed in

the network.

- If the node has the ownership in a blocked state (i.e., it already sent an ownership acknowledgment and is waiting for the corresponding backup deletion acknowledgment) it will reissue the ownership acknowledgment with a new serial number. Basically, it will act as if the *lost backup deletion acknowledgment timeout* had triggered.
- In other cases (the node has ownership but it is not blocked), the ownership ping should be ignored.

The node that has the backup will regain ownership if it receives a *NackO* message with the expected serial number, avoiding the potential deadlock if owned data had been discarded. It will forget the serial number of the issued *NackO* (hence canceling the ping) if it receives an *AckO* or a new reissued request.

VI. A token-based fault-tolerant cache coherence protocol

Prior to the design of FtDirCMP, we had already designed and evaluated a token-based fault-tolerant cache coherence protocol which we call FtTokenCMP [5]. In this section we briefly describe FtTokenCMP and compare it with FtDirCMP.

The fault tolerance measures of both protocols are similar in their intent and functionality and differ mostly in the implementation. Table IV shows a summary of the timeouts used by FtTokenCMP.

FtTokenCMP uses a mechanism similar to the one of FtDirCMP to avoid data loss, ensuring reliable transmission of owned data as described in section V-A. Section VI-A formalizes that mechanism with a modified version of token counting rules.

The fault recovery mechanism is different for each protocol. In FtTokenCMP, fault recovery is achieved by means of a centralized mechanism called the *token recreation process* arbitrated by the memory controller. This process works as long as there is a valid copy of data in some cache or one and only one backup copy (which is guaranteed by the rules in section VI-A). The memory controller attends token recreation requests in FIFO order to avoid livelock and it works sending messages to every cache asking it to invalidate all tokens and send back to memory any data that it may have. Once the memory receives the data or invalidation acknowledgments from every cache, it sends it to the cache which requested the recovery with a new set of tokens.

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

The *token serial numbers* used in FtTokenCMP serve a similar purpose to the *request serial numbers* used in FtDirCMP (e.g.: being able to discard stale messages after fault recovery which could cause an incoherence), but the latter are easier to implement and more scalable. *Token serial numbers* are associated with each cache line and need to be updated in a coordinated fashion during the *token recreation process*. Hence, they require an additional structure in each cache to store them (only for those hopefully few lines that had a token serial number different than 0, but even for lines which are not currently in any cache). On the other hand, *request serial numbers* are associated with individual requests and so they are short-lived information which can be stored in the MSHR. However, *token serial numbers* do not need to be carried in request messages (only in responses) while *request serial numbers* are sent with every request and need to be propagated with every message which is sent as consequence of the request.

In some cases, FtTokenCMP achieves deadlock recovery issuing ping messages when a timeout triggers to force the reissue of a message which is expected to finish a coherence transaction, like a *Persistent Request Deactivation*. These ping messages are analogous to the *UnblockPing* messages used by FtDirCMP.

A. Fault-tolerant token-counting rules

The main observation of the token framework is that simple token counting rules can ensure that the memory system behaves in a coherent manner. To implement fault tolerance, we have modified the token counting rules to ensure that data cannot be lost when some message fails to arrive to its destination. The following token counting rules are based of those introduced by Martin [11], and extend them to ensure reliable ownership transference (modification with respect to the original rules are *emphasized*):

- **Conservation of Tokens**: Each line of shared memory has a fixed number of T + *1* 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. *Another token is the backup token.*
- **Write Rule**: A component can write a block only if it holds all T tokens for that block *which are not the backup token* 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 *different than the backup token* for that block and has valid data.
- **Data Transfer Rule**: If a coherence message carries a dirty owner token, it must contain data.
- ***Owner Token Transfer Rule****: If a coherence message carries the owner token, it must not carry the backup token also.*
- ***Backup Token Transfer Rule****: The backup token can only be sent to another node that already holds the owner token.*
- ***Blocked Ownership Rule****: The owner token cannot be sent to other component until the backup token has been received.*
- **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 *different than the backup token*. A component clears the valid-data bit when it no longer holds any tokens *or when it holds only the backup token*. 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.

The above token counting rules along with the starvation and deadlock avoidance measures implemented by persistent

TABLE IV
SUMMARY OF TIMEOUTS USED IN FTTOKENCMP.

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 deactivated.	Request a token recreation.
Lost data	When the owner token is sent.	At the cache that holds the backup.	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 backup deletion acknowledgement arrives.	Request a token recreation.
Lost persistent deactivation	When an external persistent request is activated.	At every cache (by the persistent request table).	When the persistent request is deactivated.	Send a persistent request ping.

requests and the token recreation process compose the correctness substrate of FTTOKENCMP. These rules enforce the same global invariants than the original rules and additionally they enforce the following invariant: "For any given line of shared memory at any given point in time, there will be at least one component holding a valid copy of the data, or one and only one component holding a backup copy of it, or both". In other words: when the data is sent through the network (where it is vulnerable to corruption), it is guaranteed to be stored also in some component (where it is assumed to be safe) either as a valid and readable cache block or as a backup block to be used for recovery if necessary.

We have modified the *conservation of tokens* rule to add a special backup token. We have also modified the *write rule* and the *read rule* so that, unlike the rest of the tokens, this token does not grant any permission to its holder. Instead, a cache holding this token will keep the data only for recovery purposes. The new *owner token transfer rule* ensures that whenever a cache has to transfer the ownership to another cache, it will keep the backup token (and the associated data as a backup). The new *backup token transfer rule* ensures that the backup token is not transferred until the owner token (and hence the data) has been received by another cache. Of course, this implies that the component holding the owner token has to communicate that fact to the component that holds the backup token, usually by means of an *ownership acknowledgement*. It also implies that a cache receiving the backup token has always received the data before. Finally, the new *blocked ownership rule* ensures that there is at most one backup copy of the data, since there is only one backup token. The reliable ownership transference mechanism used by TOKENCMP (explained in [5]) complies with these rules.

VII. EVALUATION

The goals of this section are to measure the overheads introduced by our fault tolerant measures in the fault-free case and the performance degradation due to faults when they occur.

We have also performed an extensive functional validation of these measures using randomized testing and manually checking many possible cases. The randomized testing stresses protocol corner cases by issuing requests that simulate very contended accesses to a few memory lines, using random latencies for message delivery, and performing fault injection with very high fault rates. The tester also issues many concurrent requests, like a very aggressive out-of-order processor would do. Hence, although we have not done a formal verification of the protocols, we are fairly confident of the correctness of our fault-tolerant measures.

TABLE V
CHARACTERISTICS OF SIMULATED SYSTEMS.

16-Way Tiled CMP System	
Processor parameters	
Processor speed	2 GHz
Cache parameters	
Cache line size	64 bytes
L1 cache:	
Size, associativity	32 KB, 4 ways
Hit time	3 cycles
Shared L2 cache:	
Size, associativity	1024 KB, 4 ways
Hit time	15 cycles
Memory parameters	
Memory access time	160 cycles
Memory interleaving	4-way
Network parameters	
Topology	2D Mesh
Non-data message size	8 bytes
Data message size	72 bytes
Channel bandwidth	64 GB/s

A. Methodology

We have used full system simulations of a mix of applications with fault injection with the aims of determining adequate values for some protocol parameters, assess the fault tolerance capability of each protocol and measure the overhead introduced by the fault tolerance measures. For this, we have used a custom version of Multifacet GEMS [13] detailed memory model and Virtutech Simics [10]. Every simulation has been performed several times using different random seeds to account for the variability of multithreaded execution, such variability is represented by the error bars in the figures which enclose the resulting 95% confidence interval of the results. We have simulated tiled CMP systems as described in section III. Table V shows the relevant parameters common to all the simulations. We have performed experiments to adjust some protocol-specific parameters, as shown in sections VII-E and VII-F.

We have used a mix of scientific, multimedia and commercial applications for the evaluation: Apache (10000 http transactions) is version 2.2.4 of the http server serving static pages of different sizes. SpecJbb (8000 transactions) is a Java server workload based on SPEC JBB 2000. Barnes (8192 bodies, 4 time steps), FFT (256K complex doubles), Ocean (258×258 ocean), Raytrace (10Mb, teapot.env scene) and Water-SP (512 molecules, 4 time steps) are from the SPLASH-2 [25] benchmark suite. Unstructured (Mesh.2K, 5 time steps) is a computational fluid dynamics application. FaceRec (ALPBench training input), MPGdec (525_tens_040.m2v) and SpeechRec (ALPBench default input) are from the ALPBench [9] benchmark suite. The experimental results reported here correspond to the parallel phase of each

program only.

B. Execution time overhead

We have measured the execution time of each one of the fault-tolerant protocols using the fault tolerance parameters determined above with several message loss rates and compared it to the execution time of the non fault-tolerant protocols in a fault-free scenario. The results are shown in figure 7(a). Fault rates are expressed in number of messages discarded per million of messages that travel trough the network and all results are normalized with respect to the execution time of the DIRCMP protocol. Of course, results for non fault-tolerant protocols are only shown in the fault free cases.

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

These results assume that there is enough bandwidth so that the extra messages required by our protocols (see section VII-C) do not increase the latency of the network. This is likely to be the case in a CMP environment.

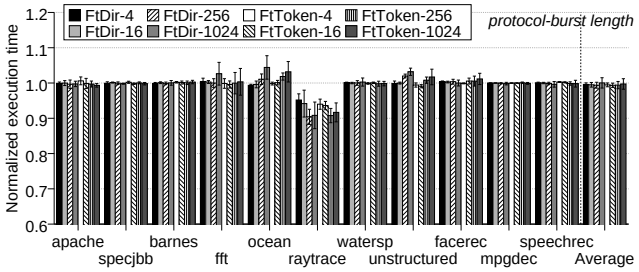
As the fault rate increases, so does the execution time. However, this performance degradation is very moderate, and only one application (Raytrace) suffers a performance degradation higher than 5% even with 125 corrupted messages per million.

C. Network overhead

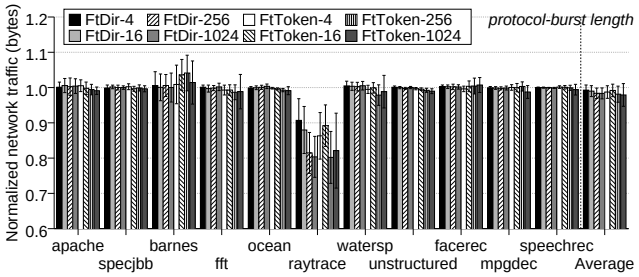
In absence of faults, the most important difference in the behavior of our protocols with respect to their non fault-tolerant counterparts is the exchange of acknowledgments to ensure that owned data is transferred safely and avoid data loss. Although they are sent out of the critical path of cache misses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

We have measured the network overhead of our proposal in terms of the relative increase in the number of messages and the number of bytes transmitted. We have increased one byte the message sizes of the fault tolerant protocols with respect to the non fault-tolerant ones to accommodate the *request serial numbers* and *token serial numbers*. This means 1.14% increase in size for data messages and 12.5% increase for control messages. The results of these measurements are shown in figures 7(b) and 7(d). To allow a comparison with simple interconnection level fault tolerance measures, we also include bars showing the network overhead of using the base protocols with end-to-end reliable delivery (DIRCMP+NLFT and TOKENCMP+NLFT), using an acknowledgment for each message.

We can see that, in terms of message traffic, the overhead of the fault-tolerant protocols comes entirely from the acknowledgments used to ensure reliable data transmission ("*Ownership*" part of each bar). This overhead is less than 40% on average for our fault-tolerant protocols. Moreover, the overhead drops considerably when it is measured in terms of bytes, even considering that every message is one byte longer in the fault-tolerant protocols.



(a) Execution time overhead



(b) Network traffic overhead

Fig. 8. Relative execution time with burst faults of several lengths with respect to single message faults. The fault rate is fixed to 125 corrupted messages per million.

FTDIRCMP has a higher relative overhead because network traffic is much lower for DIRCMP than for TOKENCMP.

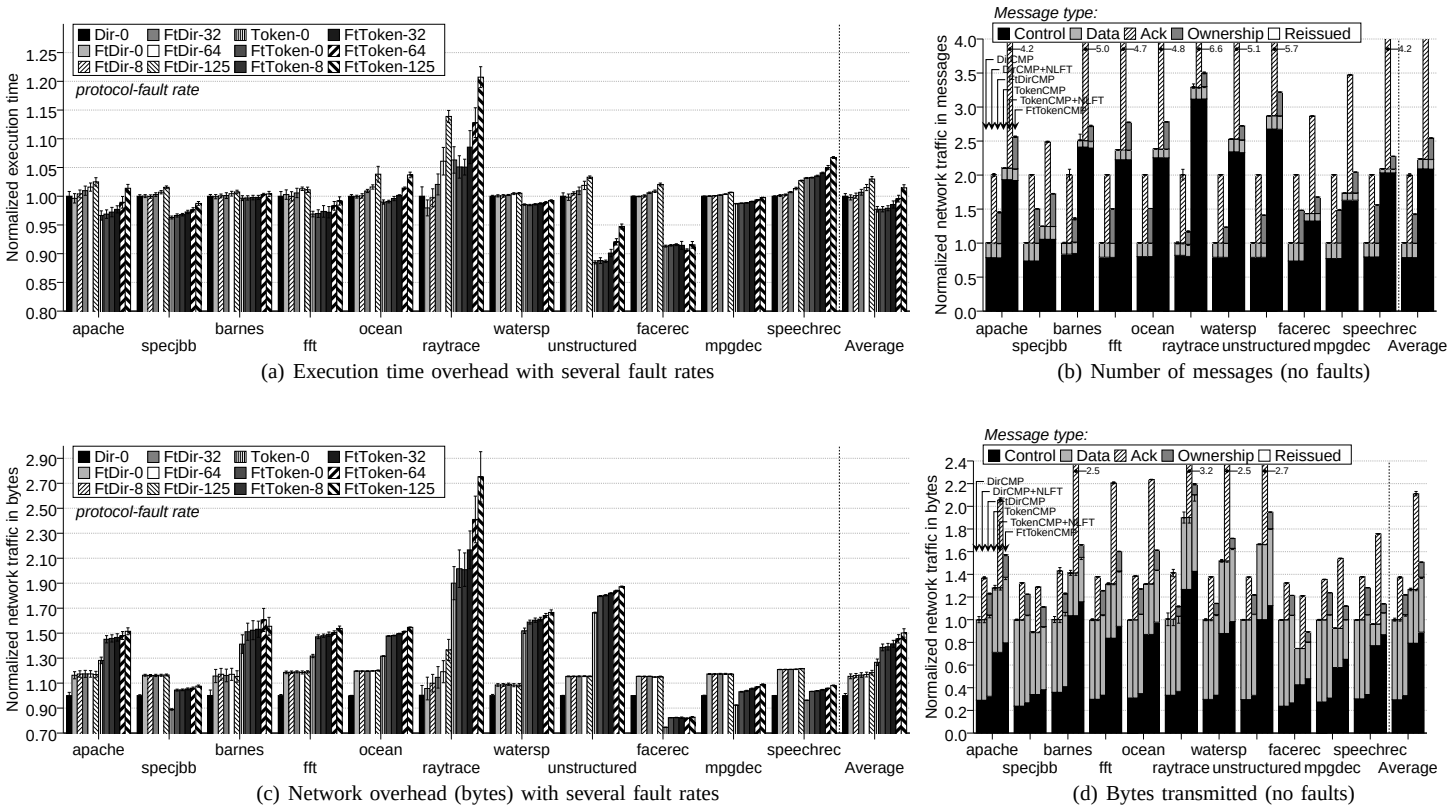
Figure 7(c) shows the network overhead under several fault rates. The network traffic increases slowly with the fault rate due to the reissued messages or the token recreation messages. In the case of FTDIRCMP, the increase is almost unmeasurable for the fault rates shown for all applications except Raytrace.

D. Effect of bursts of faults

Until now, we have assumed that all faults are distributed evenly in time and that each fault only affects one message. However, it is possible for a single fault to affect more than one message. For example, a fault may hit a buffer holding several messages and make it discard all of them. Note that usually those messages will be part of different transactions involving different addresses.

We have performed experiments to determine the effect of the burst length in the performance of our fault tolerant cache coherence protocols. Figure 8 shows how the execution time and network overhead vary with the length of burst. When performing fault injection of bursts of length L, we determine for each message whether it has been corrupted or not based on the probability given by the fault rate divided by L. If the message is determined to have been corrupted, it and the next $L-1$ messages to arrive will be discarded. This ensures that the total number of corrupted messages is the same for the same amount of traffic and fault rate, independently of the burst size.

As can be seen in figure 8(a), on average and for most applications the length of the burst of dropped messages has little effect in the execution time. Some applications (like ocean and unstructured) show a modest slowdown as the fault rate increases. More surprisingly, some applications seem to benefit from longer bursts. This can be explained due to the fact that since the total



(a) Execution time overhead with several fault rates

(b) Number of messages (no faults)

(c) Network overhead (bytes) with several fault rates

(d) Bytes transmitted (no faults)

Fig. 7. Execution time and network overhead of each cache coherence protocol compared to DIRCMP.

number of messages that get corrupted is approximately the same (125 messages per million of messages that travel trough the network), longer bursts actually mean fewer faults (each fault affects more messages). Since the recovery of each message usually happens in parallel to the recovery of other messages, the overhead of the recovery process may actually be reduced over the whole execution of the program. Raytrace is also the application which is most affected by single message faults, as can be seen in figure 7(a).

The effect in network traffic is similar, as can be seen in figure 8(b). It is important to note that the effect of burst length in performance is more dependent on the application than on the particular coherence protocol: the behavior for each application is very similar with FTDIRCMP and with FTTOKENCMP.

E. Adjusting the fault detection timeouts

As explained above, all fault-tolerant protocols achieve fault detection by means of a number of timeouts. Each protocol requires up to four timeouts which are active at different places and times during a memory transaction or cache replacement. The value of these timeouts determines the latency of fault detection, hence shorter values help to achieve lesser performance degradation in presence of faults since fault recovery will start earlier. For example, for the two fault-tolerant protocols considered in this work, figure 9 shows how the execution time increases more than 15% on average when the value of these timeouts vary from 1500 cycles to 4000 cycles under a fixed fault rate of 250 corrupted messages per million of messages that travel through the network.

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

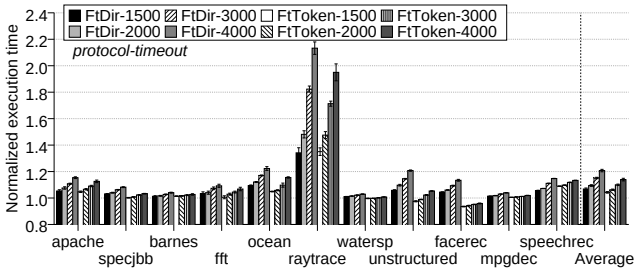


Fig. 9. Relative execution time with respect to DIRCMP without faults for each fault-tolerant protocol with 250 corrupted messages per million using different values for the fault detection timeouts.

the timeout values should be large enough to allow every memory transaction to finish, assuming that no fault occurs. Figure 10 shows the measured maximum latency in CPU cycles of each protocol when no faults occur and disabling all the timeouts.

Looking at figure 10, we can see that the maximum latency of the fault-tolerant protocols is almost the same as that of their corresponding non fault-tolerant counterpart[5]. This is expected, since the behavior of the fault-tolerant protocols when no timeout triggers is almost the same as that of the non fault-tolerant ones, except for the ownership acknowledgments which are sent out of

[5]In some cases, the maximum latency of the fault tolerant protocol is slightly lower that its non fault tolerant counterpart, which may be surprising. This is accidental and is not due to any optimization. Slight changes in the behavior of the protocols produce these variations due to the non deterministic nature of parallel applications.

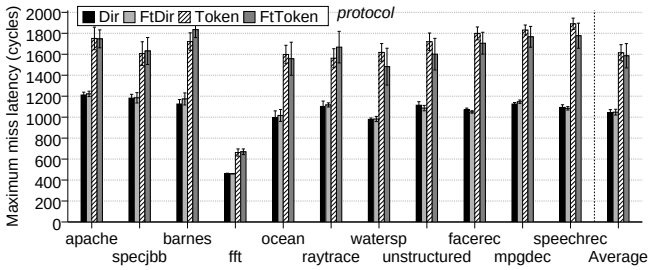


Fig. 10. Maximum miss latency (in cycles) of each protocol without faults.

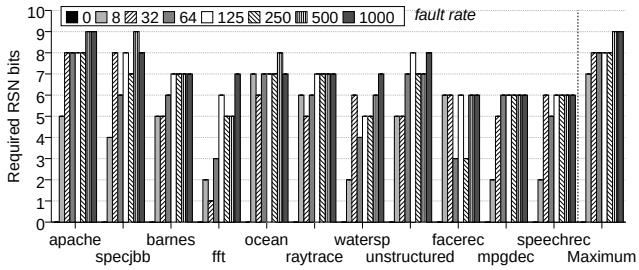


Fig. 11. Required RSN bit length to discard every old response to a reissued message in FTDIRCMP.

the critical path of cache misses.

This latency is less than 1200 cycles for the FTDIRCMP protocol and less than 1900 cycles for the FTTOKENCMP protocol. Hence, we can choose any value greater than those for the timeouts to avoid having any false positive for these workloads. Using shorter values is still possible but would increase the number of false positives and could degrade performance and increase network traffic due to the retried requests or token recreation requests. However, if the chosen values are too low (lower than the time required to finish the transaction), the recovery mechanism would be invoked too frequently preventing forward progress.

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

We have chosen a value of 2000 cycles for all timeouts in the FTTOKENCMP protocol and 1500 cycles in the FTDIRCMP protocol. These values are large enough to avoid false positives in every case and, as shown below, achieve very low performance degradation when faults actually occur.

F. Effect of the request serial number size in fault tolerance

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

To measure this, we have performed simulations of FTDIRCMP using a wide variety of fault rates. We have used 32-bit request serial numbers in our simulator to encode the request serial number for these simulations but we have recorded how many lower order bits were required to distinguish all the request serial numbers that needed to be compared (every time that two request serial numbers are compared, we record the position of the least significant bit which is different in both numbers). Then, we assume that the maximum of all these measures is an upper bound of the number of bits required to ensure correctness for each fault rate. These results are shown in figure 11.

As can be seen, when using the FTDIRCMP protocol 9 bits are enough for all the tested fault rates and 8 bits suffice for fault rates up to 250 corrupted messages per million. Hence, we have chosen to use 8 bits to encode the request serial numbers in the rest of our experiments which is enough to achieve fault tolerance up to 250 corrupted messages per million, which is already an unrealistic and unreasonably high fault rate. For the rest of the evaluation, we will show fault rates only up to 125 corrupted messages per million. This fault rate should be supported by both protocols with the configuration described above.

VIII. HARDWARE IMPLEMENTATION OVERHEADS

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

On the other hand, *request serial numbers* do not need to be kept once the memory transaction has completed. They can be stored in the MSHR or in a small associative structure in cases where a full MSHR is not needed. As shown in section VII-F, using 8 bits to encode request serial numbers is enough to achieve tolerance to very high fault rates.

To be able to detect reissued requests in FTDIRCMP, the identity of the requester currently being serviced by the L2 or the memory controller needs to be recorded, as well as the receiver when transferring ownership from one L1 cache to another.

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

We have analyzed FTDIRCMP from the point of view of its implementation using deterministic routing on a 2D-mesh. Due to the exchange of ownership acknowledgments to ensure reliable data transmission, the worst case message dependence chains of FTDIRCMP are one message longer than those of DIRCMP.

Hence, a correct implementation requires an additional virtual network to ensure deadlock free operation.

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

Finally, the design complexity of the cache coherence protocol increases due to the fault tolerance measures. However, the additional complexity is assumable, and the fault tolerance measures may simplify the handling of some corner cases.

IX. Conclusion

We have shown that it is possible to deal with transient faults in the interconnection network of CMPs at the cache coherence protocol level. For this task, we have designed a fault-tolerant directory-based coherence protocol which ensures the correct execution of programs even if the network is subject to transient faults and does not correctly deliver all the coherence messages, and we have presented a set of fault-tolerant token counting rules and a fault-tolerant token based protocol that uses them.

We have compared and evaluated the performance of the two protocols using full system simulation and performing fault injection to check the correctness of the protocol and to measure the performance degradation caused by several fault rates. We have shown that the overhead imposed in the execution time due to the fault-tolerant measures is negligible. Further, we have shown that the performance impact of moderate fault rates in the interconnection network is insignificant when using our protocols.

We have explained how to tune the fault tolerance parameters of the protocols to achieve the desired level of fault tolerance, performance degradation in presence of faults and overhead in absence of faults. We have shown that, even for fault rates which are unrealistically high, the hardware overhead of our proposals is low. The main cost of our fault tolerance measures is a moderate increase in network traffic.

We have found that the network usage of our protocols increases with the fault rate and hence network capacity can be a limiting factor for fault tolerance. Due to the efficient network usage of directory-based protocols, we think that FtDirCMP is a good cache coherence protocol for large scale tiled CMPs.

As future work, we would want to explore whether similar techniques can be used to deal with intermittent faults. Also, since the main feature of our protocol that it does not assume that every coherence message arrives to its destination while still guaranteeing correct program execution, we want to try to take advantage of this ability to allow the interconnection network to occasionally drop messages if that helps with performance in the common case or enables simpler interconnection network designs.

References

[1] N. Aggarwal, P. Ranganathan, N.P. Jouppi, and J.E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. In *34th Int'l Symposium on Computer Architecture (ISCA 2007)*, June 2007.

[2] M. Ali, M. Welzl, and S. Hessler. A fault tolerant mechanism for handling permanent and transient failures in a network on chip. In *Proc. of the Int'l Conference on Information Technology (ITNG 2007)*, pages 1027–1032, 2007.

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

[4] K. Constantinides, S. Plaza, J. Blome, B. Zhang, V. Bertacco, S. Mahlke, Todd Austin, and M. Orshansky. BulletProof: a defect-tolerant CMP switch architecture. In *12th Int'l Symposium on High-Performance Computer Architecture (HPCA'06)*, pages 3–14, February 2006.

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

[6] R. Fernández-Pascual, J.M. García, M.E. Acacio, and J. Duato. Extending the TokenCMP cache coherence protocol for low overhead fault tolerance in CMP architectures. *IEEE Transactions on Parallel and Distributed Systems*, 19(8):1044–1056, 2008.

[7] R. Fernández-Pascual, J.M. García, M.E. Acacio, and J. Duato. Fault-tolerant cache coherence protocols for CMPs: evaluation and trade-offs. In *Proc. of the International Conference on High Performance Computing (HiPC 2008)*, December 2008.

[8] R. Fernández-Pascual, J.M. García, M.E. Acacio, and J. Duato. A fault-tolerant directory-based cache coherence protocol for shared-memory architectures. In *Proc. of the International Conference on Dependable Systems and Networks (DSN 2008)*, June 2008.

[9] M. Li, R. Sasanka, S.V. Adve, Y. Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *Proc. of the IEEE Int. Symp. on Workload Characterization*, pages 34–45, 2005.

[10] 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.

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

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

[13] 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*, 33(4):92–99, September 2005.

[14] 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, February 2005.

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

[16] Albert Meixner and Daniel J. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. In *Proc. of the Int'l Conference on Dependable Systems and Networks (DSN 2006)*, pages 73–82, June 2006.

[17] 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-11)*, February 2005.

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

[19] Dongkook Park, Chrysostomos Nicopoulos, Jongman Kim, N. Vijaykrishnan, and Chita R. Das. Exploring fault-tolerant network-on-chip architectures. In *Proc. of the 2006 Int'l Conference on Dependable Systems and Networks (DSN'06)*, pages 93–104, 2006.

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

[21] 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.

[22] 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. In *29th Annual Int'l Symposium on Computer Architecture (ISCA'02)*, pages 123–134, May 2002.

[23] M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, J. Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The RAW microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, May 2002.

[24] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman,

Y. Hoskote, and N. Borkar. An 80-tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *IEEE Int'l Solid-State Circuits Conference*, 2007.

[25] S.C. Woo, M. Ohara, E. Torrie, J.P. 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.