A fault tolerant coherence protocol for CMP architectures

Ricardo Fernández-Pascual, José M. García and Manuel E. Acacio
Universidad de Murcia
Grupo de Arquitectura y Computación Paralela
{r.fernandez,jmgarcia,meacacio}@ditec.um.es

Abstract— **It is a well known fact that transient failures will increase in chips designed in the near future due to several factors such as the increased integration scale. On the other hand, chip-multiprocessors (CMP) that integrate several processor cores in a single chip are nowadays the best alternative to more efficient use of the increasing number of transistors that can be placed in a single die. Hence, it is necessary to design new techniques to deal with these faults to be able to build sufficiently reliable Chip Multiprocessors (CMPs). In this work, we present a coherence protocol aimed at dealing with transient failures that affect the interconnection network of a CMP assuming that the network is no longer reliable.**

Keywords— **Fault tolerance, cache coherence, CMP, transient failures.**

I. Introduction

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

Moreover, the reliability of electronic components is never perfect. Electronic components are subject to several types of failures due to a number of causes. 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, also known as soft errors or single event upsets, occur when a component produces an erroneous output and it continues working correctly after the event. The causes of transient errors are multiple and include alpha-particle strikes, cosmic rays and radiation from radioactive atoms which exist in trace amounts in all materials and electrical sources like power supply noise, electromagnetic interference (EMI) or radiation from lightning. Any event which upsets the stored or communicated charge can cause soft errors in the circuit output.

Transient failures are much more common than permanent failures [1]. 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 [2]. Since the number of components in a single chip increases so much, it is no longer economically feasible to assume a worst case scenario when designing and testing the chips. Instead, new designs will target the common case and assume a certain rate of transient failures. Hence, transient failures will affect more components and more frequently and will need to be handled across all the levels of the system to avoid actual errors.

In message passing machines, message loss can be solved at the library level (usually MPI) or using reliable network protocols. In those machines, communication patterns are controlled by the application developer, so communication tends to be coarse-grained and most messages are larger than in shared-memory machines. Hence the cost of reliable transmission in message-passing architectures is assumable.

On the contrary, shared-memory machines do not rely in any library to deal with dropped messages. Communication is very fine-grained (at the level of cache blocks from 64 to 256 bytes), hence smaller and more frequent messages are used. In order to achieve the best possible performance it is necessary to use low-latency interconnections and avoid acknowledgement messages and other control-flow messages. For these reasons, dropped messages seriously limit the scalability of these machines.

In this work, we propose a way to deal with the transient failures that occur in the interconnection network of shared-memory multiprocessor systems. We can assume that these failures cause the loss of some messages, because either the interconnection network losses 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.

In order to build shared-memory machines as reliable and scalable as possible, we need to build large, reliable and low-latency interconnection networks both on chip and off chip, which are hard to design and expensive; an alternative is to add enough fault-tolerance to our system to cope with an unreliable interconnection network.

In this we paper attack this problem at the coherence protocol level. We propose a coherence protocol which assumes an unreliable interconnection network and guarantees 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, so it is applicable to current and future glueless designs. Up to the best of our knowledge, no previous proposal has addressed this topic yet.

Recently, a new kind of coherence protocols have been proposed. Token coherence protocols [3] avoid the need of a totally ordered network and the introduction of additional latency in the common case. Several variants of token coherence protocols are possible for small, medium and large scale multiprocessors. Token protocols decouple the correctness substrate from the performance policy, allowing greater flexibility and reducing the complexity of designing new protocols.

To take advantage of this decoupling, we have based our proposal on a token based coherence protocol for Multiple-CMPs [4] but we expect that these ideas are applicable to any token based protocol. Decoupling correctness and performance allows us to concentrate on fault-tolerance at the level of the correctness substrate and avoid penalizing the common case.

In this work, we have designed a new coherence protocol for CMPs and multiple CMPs (M-CMPs) based on token coherence which ensures correct behavior from the coherence and consistency point of view even if some coherence messages do not arrive to their destination. Also, we have measured the overhead introduced by the new protocol compared to a similar protocol without fault tolerance support using full-system simulations.

The rest of this paper is organized as follows. In section II we present some related works, including some proposals for fault tolerance using checkpointing for scalable multiprocessors and a summary of the base coherence protocol. In section III we describe a new approach for designing a fault-tolerant coherence protocol that tries to avoid checkpointing and does not need a reliable interconnection network. Section IV presents a preliminary evaluation of the protocol. Finally, in section V we present some conclusions.

II. Related Work

A. Fault tolerance for shared-memory multiprocessors

There have been several proposals for dealing with fault tolerance in cache-coherent multiprocessor systems, either for those using a shared bus or an unordered interconnection network. Most of these proposals use variations of checkpointing and recovery. In most cases, some amount of hardware redundancy is necessary too, at least to ensure the reliability of the fault-tolerant support hardware itself.

A large body of literature exists concerning checkpointing and recovery, primarily for message-passing loosely coupled distributed systems rather than tightly coupled shared-memory multiprocessors [5] [6] [7]. Recovery strategies, using rollback and replay of input messages, for message-passing distributed systems are not generally applicable to recovery from transient errors in shared-memory multiprocessors.

However, there have been several proposals targeting shared-memory multiprocessors: R.E. Ahmed *et al.* developed Cache-Aided Rollback Errors Recovery (CARER) [8], Wu *et al.* [9] developed error recovery techniques using private caches for recovering from processor transient faults in multiprocessor systems, Banâtre *et al.* propose a *Recoverable Shared Memory* (RSM) which deals with processor failures on shared-memory multiprocessors using snoopy protocols [10] [11], while Sunada *et al.* propose *Distributed Recoverable Shared Memory with Logs* (DRSM-L) [12]. More recently, Pruvlovic *et al.* presented ReVive, which performs checkpointing, logging and memory based distributed parity protection with low overhead in error-free execution and is compatible with off-the-shelf processors, caches and memory modules [13]. At the same time, Sorin *et al.* presented SafetyNet [14] which aims at similar objectives but has less overhead, requires custom caches and can only recover from transient faults.

B. Token coherence

Token coherence [15] is a framework for designing coherence protocols whose main asset is that it decouples the correctness substrate from several different performance policies. This allows a great flexibility, making possible to adapt the protocol for different purposes easily [3] since the performance policy can be modified without worrying about infrequent corner cases, whose correctness is guaranteed by the correctness substrate.

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

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

Token coherence provides the framework for designing several particular coherence protocols. Building upon the correctness substrate, a variety of *performance policies* may be designed specifying the precise behavior of each processor and memory module to different coherence messages.

To date, only a few performance policies have been designed: *Token-using-broadcast* (TokenB) is a performance policy to simultaneously achieve low-latency cache-to-cache transfer misses and is faster than both traditional snooping protocols and directory protocols, although it requires more bandwidth [15]. *Token-based-directory* (TokenD) [3] emulates a directory based protocol using the token framework. TokenM [3] is a performance policy that seeks a compromise between bandwidth usage and latency by means of prediction. Finally, TokenCMP [4] is a performance policy similar to TokenB which targets hierarchical multiple CMP systems.

III. A fault tolerant token coherence protocol

We currently consider errors resulting in loss of protocol messages, either losing an isolated message or a burst of messages. Instead of detecting faults and return to a consistent state previous to the occurrence of the fault, our aim is to design a coherence protocol that can guarantee the correct semantics of program execution over an unreliable interconnection network without ever having to perform a checkpointing or rollback. We do not try to address the full range of errors that can occur in a CMP system. We only concentrate on those errors that affect directly the interconnection network and which can be tolerated modifying the coherence protocol. Hence, other mechanisms must be used to complement our proposal in order to achieve full fault tolerance.

Our proposal modifies the correctness substrate of the token protocol assuming an unreliable interconnection network. Thanks to decoupling correctness from performance, the performance policies can still be used with minimal or no changes. This minimizes the performance impact in the common fault-free case.

A. Possible faults

Loosing some of the coherence messages in a token coherence protocol, like 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.

Coherence messages in a token protocol can contain one or more tokens which would be lost if a message is lost. The total number of tokens in the whole system must remain constant to ensure the correct behavior of the system. More precisely, if the number of tokens decreases, no processor will be able to write to that block of memory anymore. On the other hand, if the number of tokens increases, a processor would be able to write to the memory block while another processor keeps a readable copy, violating the memory coherence model.

Hence, if a coherence message containing a token were lost, a deadlock would occur. We propose a mechanism for detecting this fact and recovering the lost token or tokens.

Also, coherence messages can contain data. In particular, if a coherence message contains a dirty owner token, then it must also carry the memory block. If a message containing data but not the owner token is lost, the requester will eventually time out and ask again for the data (with a retried transient request or with a persistent request). However, if the owner token is lost too, no processor (or memory module) would send the data and a deadlock and possibly data loss would occur.

Finally, while a persistent request is in process, we have to deal with errors in the persistent request messages as well as the same errors as in the usual case. Losing a persistent request or persistent request deactivation would create inconsistencies amongst the persistent request tables at each node in a distributed arbitration scheme.

We present a summary of all the possible problems due to loss of messages in table I. Only the messages that are found in TokenCMP with a distributed arbitration for persistent requests are shown. Later, we show how to prevent or solve each one of those situations.

TABLE I
Summary of possible problems due to loss of messages.

Message lost	Effect
Transient read/write request	Harmless
Response with tokens	Deadlock
Response with tokens and data	Deadlock
Response with a clean owner token	Deadlock
Response with a dirty owner token and data	Deadlock and data loss
Persistent read/write requests	Deadlock
Persistent requests	Deadlock

B. Adding fault-tolerance to the base protocol

To prevent adding a significant overhead to the fault-free case and to keep the flexibility of choosing any particular performance policy, we should try to avoid modifying the usual behavior of transient requests. For example, we should avoid placing point-to-point acknowledgements in the critical path as much as possible.

Since the base protocol does not have unacknowledged invalidations, loosing a message cannot lead to an incoherence. In every problematic case shown in table I, loosing a message would lead to a deadlock. Hence, a possible way to detect faults is using timeouts for transactions. The timeout may be triggered for the same coherence transaction that losses the message or for a subsequent transaction. We call this timeout the *"lost token timeout"* and it will start when a persistent request is activated and will stop once the miss is satisfied or the persistent request is deactivated.

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

In some cases, we will need to recreate possibly lost tokens. When doing this, we must respect the *Conservation of Tokens* invariant [3]. So, to avoid increasing the total number of tokens for a memory block even in the case of a false positive, we need to ensure that all the old tokens are discarded. To achieve this we define a *token serial number* conceptually associated with each token and each memory block.

All tokens of the same memory block should have the same serial number. Every node in the system must know the serial number associated with each memory block and should discard every message received containing an incorrect serial number. The serial number will be transmitted within every coherence response.

The overhead associated with the token serial number is small. In the first place, we will need to increase it very infrequently, so a counter with a small number of bits should be enough (we assume a two bit wrapping counter). Secondly, most memory blocks will keep the initial serial number unchanged, so we only need to store those ones which have changed and assume the initial value for the rest. Thirdly, the comparisons required to check the validity of received messages can be done out of the critical path. To avoid filling the *token serial number* table, serial numbers can be reset after some time.

To store the token serial number of each block we propose a small associative table present at each node. Only blocks with an associated serial number different than zero must keep an entry in that table. The information of the tables must be identical in all the nodes, so we must ensure a reliable protocol for updating it. Since updates to this table should be very infrequent, we can use point-to-point acknowledgments in this case.

B.1 Dealing with token loss

When a message containing one or more tokens is lost, the total number of tokens in the system decreases. As stated above, this would lead to deadlock because no processor will be able to write anymore to that block.

When a processor tries to write to a memory block which has lost a token, it will eventually timeout and issue a persistent request. Eventually, after the persistent request gets activated, all the available tokens in the whole system for the memory block will be received by the starving node. Also, if the owner token was not lost, the node will receive it too together with data.

If the starving node fails to acquire the necessary tokens within certain time after the persistent request has been activated, the *lost token timeout* will trigger. In that case, we will assume that some token carrying message has been lost and we will start the recovery process.

Once the problem has been detected, it is safe for the starving node to recreate the missing tokens, because it is guaranteed that no other processor or memory module has any token for that memory block. Once the tokens are restored, the operation can continue normally.

B.1.a **Token recreation process**. 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. The process works as follows:

The starving node that detects the problem sends a *recreate tokens* request to the memory controller responsible for that line. That request also informs whether the requester node has valid data for the line or not. The memory will then increase the *token serial number* associated to the line and send a *set token serial number* message to every node. When receiving that message, each node updates the *token serial number*, destroys any token that it could have and sends an acknowledgment to the memory. Additionally, if the node has a backup of the line (see section III-B.2 to see when this happens), it is sent to the directory with the acknowledgment. Once the memory receives all the acknowledgments, it will send a *destruction done* message to the starving node. If the starver did not have the data when requesting the token recreation and the memory has received one[1] backup copy of the data, it will send it in the same message. Once the node receives this message, it recreates all the tokens and continues.

If there was no real failure but a token carrying message was delayed on the network due to congestion, it will be discarded when received by any node because the *token serial number* will not match.

B.2 Avoiding data loss

The rules governing the owner token ensure that there is always at least a valid copy of the memory block which travels along with it every time the owner token is transmitted. So, losing the owner token means that it is possible to totally lose the data of a memory block.

[1]If the memory has received more than one backup of the line and the node requested the data, there is no way to recover from the failure since it is not possible to know which backup data is the newest version of the block. To get to this unlikely situation, a number of particular messages (including several acknowledgments and a data message) need to have been lost repeatedly.

To avoid losing data, a node that has to send the owner token will keep the line in *backup* state. A line in backup state will not be evicted from the cache until an *ownership acknowledgment* is received, even if every token is sent to other nodes. This acknowledgment is sent by every node in response to a message carrying the owner token.

While a line is in *backup* state its data may be invalid. Hence, the node will no be able to read from that line unless it receives valid data (and a token) from other node. This data will be used to answer to a *set serial number* request if necessary as described in paragraph III-B.1.a.

Notice that this mechanism also affects replacements (from L1 to L2 and from L2 to memory) and will increase their latency.

B.2.a **Handling the loss of an ownership acknowledgment or a data carrying message**. If an *ownership acknowledgment* message is lost, the line in backup state will never be evicted. Also, this would make possible to have more than one backup copy for recovering from data loss. To avoid this, a node which holds a line in *backup* state for more than certain amount of time will issue a persistent write request for that line. If the persistent request is eventually satisfied, that means that the *ownership acknowledgment* was actually lost, but the data is safe and the system can continue to work normally. If the persistent request can not be satisfied, that means that the data carrying message was lost, hence the node can start the recovery process described in paragraph III-B.1.a using its backup copy as the new data. The timeout used for the *ownership acknowledgment* must be significantly shorter than that for the *lost token timeout.*

B.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 requests tables at each node synchronized. Losing one of these messages will lead to an inconsistency amongst the different tables. This inconsistency can be fixed reissuing the persistent request or request deactivation. For this to work correctly, the logic of the persistent request table needs to be modified to handle duplicated requests or unexpected deactivations.

B.3.a **Dealing with the loss of a request**. If a node holding at least one token for the requested line does not receive the persistent request, it will not activate it and will not send the tokens and data. To avoid this situation, the starving node can reissue the persistent request after a certain timeout. This timeout should be significantly shorter than the *lost token timeout.*

If the node that does not receive the persistent request did not have tokens necessary to satisfy the miss, it will eventually receive an unexpected deactivation message which it should ignore.

TABLE II
CHARACTERISTICS OF SIMULATED MACHINE

4-Way CMP System	
Processor Parameters	
Processor speed	2 GHz
Max. fetch/retire rate	4
Cache Parameters	
Cache block size	64 bytes
L1 cache:	
Size, associativity	32 KB, 2 ways
Hit time	2 cycles
Shared L2 cache:	
Size, associativity	512 KB, 4 ways
Hit time	15 cycles
Memory Parameters	
Memory access time	300 cycles
Memory interleaving	4-way
Network Parameters	
Topology	Fully connected
Non-data message size	2 flits
Channel bandwidth	64 GB/s

B.3.b **Dealing with the loss of a deactivation**. If a persistent request deactivation message is lost, the request will be permanently activated at some node. To avoid this, the node will send a *persistent request ping* to the starver after a certain amount of time. A node receiving a *persistent deactivation ping* will answer it with a persistent request or persistent request deactivation message whether it has a pending persistent request for that line or not, respectively.

IV. Evaluation

We have done a preliminary implementation of our protocol using the GEMS simulator and compared it against the original TokenCMP [4] protocol. We model a 4-way chip multiprocessor whose more relevant characteristics are shown in II. We use an in-order processor model for simplicity, but the processor frequency is four times as fast as the memory system frequency to approximate a 4-way superscalar model. We have used a number of scientific benchmarks to perform the evaluation.

In figure 1 we show the execution time overhead incurred by our fault-tolerant protocol in the fault-free case compared to a base TokenCMP protocol without any fault tolerance support. As we can see, this overhead is near to 5% in average and is as high as 11% in the worst case.

The overhead comes from two issues: the extra bandwidth consumed by the acknowledgements and the increased replacement latency mentioned in section III-B.2. The second issue is significantly more important, since it increases the average latency of misses, and we are currently studying ways to minimize its impact.

V. Conclusions

The rate of transient failures in new chips will increase in the near future due a number of factors like the increased scale of integration, the lower voltages used and changes in the design process. This will create problems for CMPs that will need new techniques

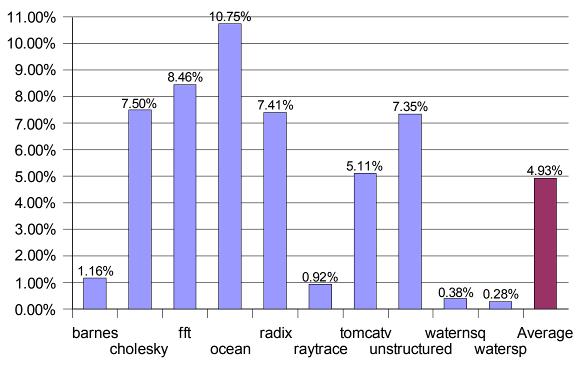

Fig. 1. Execution time overhead of our protocol compared to TokenCMP.

to avoid errors. One important source of problems will be faults in the interconnection network used to communicate between the cores and the caches. In this work, we have proposed a new coherence protocol which can work over an unreliable network aimed at dealing with those faults.

We have shown some preliminary results evaluating the execution time overhead introduced in the error-free case. We have not measured the impact of faults in the execution time yet, but we have tested the fault tolerance of most parts of the proposed protocol.

The hardware overhead required to implement the fault-tolerance measures proposed for our protocol is minimal: just a small associative table at each cache to store the *token serial number*.

Unfortunately, the overhead introduced in the fault-free case is currently higher than desirable and we are studying ways to reduce it, like using victim caches to store the lines in backup states.

Acknowledgements

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

References

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

[2] Atul Maheshwari, Wayne Burleson, and Russell Tessier, "Trading off transient fault tolerance and power consumption in deep submicron (DSM) VLSI circuits," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 12, no. 3, pp. 299–311, March 2004.

[3] Milo M.K. Martin, *Token Coherence*, Ph.D. thesis, University of Wisconsin-Madison, December 2003.

[4] Michael R. Marty, Jesse D. Bingham, Mark D. Hill, Alan J. Hu, Milo M. K. Martin, and David A. Wood, "Improving multiple-cmp systems using token coherence," in *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, Washington, DC, USA, 2005, pp. 328–339, IEEE Computer Society.

[5] Richard Koo and Sam Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering*, vol. 13, no. 1, pp. 23–31, January 1987.

[6] Yuval Tamir and Carlo H. Sequin, "Error recovery in multicomputers using global checkpoints," in *13th International Conference on Parallel Processing*, August 1984, pp. 32–41.

[7] E.N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, September 2002.

[8] R.E. Ahmed, R.C. Frazier, and P.N. Marinos, "Cache-aided rollback error recovery (CARER) algorithm for shared-memory multiprocessor systems," in *Fault-Tolerant Computing. FTCS-20.*, June 1990, pp. 82–88.

[9] K.L. Wu, W.K. Fuchs, and J.H. Patel, "Error recovery in shared memory multiprocessors using private caches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 231–240, April 1990.

[10] Michel Banâtre, Alain Gefflaut, Philippe Joubert, Peter A. Lee, and Christine Morin, "An architecture for tolerating processor failures in shared-memory multiprocessors," Tech. Rep. 1965, INRIA, March 1993.

[11] Michel Banâtre, Alain Gefflaut, Philippe Joubert, Christine Morin, and Peter A. Lee, "An architecture for tolerating processor failures in shared-memory multiprocessors," *IEEE Transactions on Computers*, vol. 45, no. 10, pp. 1101–1115, October 1996.

[12] Dwight Sunada, Michael Flynn, and David Glasco, "Multiprocessor architecture using an audit trail for fault tolerance," in *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, June 1999, pp. 40–47.

[13] Milos Prvulovic, Zheng Zhang, and Josep Torrellas, "ReVive: Cost-effective architectural support for rollback," in *29th Annual International Symposium on Computer Architecture*, May 2002, pp. 111–122.

[14] Daniel J. Sorin, Milo M.K. Martin, Mark D. Hill, and David A. Wood, "SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *29th Annual International Symposium on Computer Architecture*, May 2002, pp. 123–134.

[15] Milo M.K. Martin, Mark D. Hill, and David A. Wood, "Token coherence: A new framework for shared-memory multiprocessors," *IEEE Micro*, vol. 23, no. 6, pp. 108–116, November/December 2003.