Universidad de Murcia
Departamento de Ingeniería y
Tecnología de Computadores

Fault-tolerant Cache Coherence Protocols for CMPs

A dissertation submitted in fulfillment of
the requirements for the degree of

Doctor en Informática

By
Ricardo Fernández Pascual

Advised by
José Manuel García Carrasco
Manuel Eugenio Acacio Sánchez

Murcia, June 2009

Universidad de Murcia
Departamento de Ingeniería y
Tecnología de Computadores

Protocolos de Coherencia Tolerantes a Fallos

Tesis propuesta para la
obtención del grado de

Doctor en Informática

Presentada por
Ricardo Fernández Pascual

Supervisada por
José Manuel García Carrasco
Manuel Eugenio Acacio Sánchez

Murcia, Junio de 2009

UNIVERSIDAD
DE MURCIA

DEPARTAMENTO DE INGENIERÍA Y TECNOLOGÍA DE
COMPUTADORES

D. José Manuel García Carrasco, Catedrático de Universidad del Área de Arquitectura y Tecnología de Computadores en el Departamento de Ingeniería y Tecnología de Computadores

y

D. Manuel Eugenio Acacio Sánchez, Profesor Titular de Universidad del Área de Arquitectura y Tecnología de Computadores en el Departamento de Ingeniería y Tecnología de Computadores

AUTORIZAN:

La presentación de la Tesis Doctoral titulada «*Fault-tolerant Cache Coherence Protocols for CMPs*», realizada por D. Ricardo Fernández Pascual, bajo su inmediata dirección y supervisión, y que presenta para la obtención del grado de Doctor por la Universidad de Murcia.

En Murcia, a de Junio de 2009.

Fdo: Dr. José Manuel García Carrasco

Fdo: Dr. Manuel Eugenio Acacio Sánchez

UNIVERSIDAD
DE MURCIA

DEPARTAMENTO DE INGENIERÍA Y TECNOLOGÍA DE
COMPUTADORES

D. Antonio Javier Cuenca Muñoz, Profesor Titular de Universidad del Área de Arquitectura y Tecnología de Computadores y Director del Departamento de Ingeniería y Tecnología de Computadores, INFORMA:

Que la Tesis Doctoral titulada «*Fault-tolerant Cache Coherence Protocols for CMPs*», ha sido realizada por D. Ricardo Fernández Pascual, bajo la inmediata dirección y supervisión de D. José Manuel García Carrasco y de D. Manuel Eugenio Acacio Sánchez, y que el Departamento ha dado su conformidad para que sea presentada ante la Comisión de Doctorado.

En Murcia, a de Junio de 2009.

Fdo: Dr. Antonio Javier Cuenca Muñoz

A María Dolores y José Ricardo

Abstract

Technology trends are making possible to put billions of transistors in a single chip, which has allowed computer architects to build single-chip multiprocessors (CMPs). Unfortunately, these trends of miniaturization also mean that the reliability of individual transistors decreases. This way, transient faults, which were once a serious problem only for memories and in extreme environments like aerospatial applications, are expected to be a problem for future many-core CMPs. One of the components prone to experience transient faults in CMP systems will be the on-chip interconnection network.

In this thesis, we propose a new way to deal with transient faults in the interconnection network that is different from the classic approach of building a fault-tolerant interconnection network. In particular, we propose to provide fault tolerance measures at the level of the cache coherence protocol so that it guarantees the correct execution of parallel programs even when the underlying interconnection network does not necessarily deliver all messages correctly. By doing this, we can take advantage of the different meaning of each message to implement fault tolerance with lower overhead than at the level of the interconnection network, which has to treat all messages alike with respect to reliability.

To demonstrate the viability of our approach, we design a number of fault-tolerant cache coherence protocols. First, we design FtTokenCMP, based on the token coherence framework. FtTokenCMP adds timeouts for fault detection and simple recovery mechanisms for transient faults to a previously proposed token-based cache coherence protocol. We also extend the general token counting rules to ensure reliable transference of owned data, to simplify the application of these fault tolerance measures to other cache coherence protocols based on the token framework. Secondly, we design FtDirCMP: a directory-based fault-tolerant cache coherence protocol which adds measures inspired by the previous work

in FtTokenCMP. Finally, the same ideas are used to design FtHammerCMP: a broadcast-based and snoopy-like fault-tolerant cache coherence protocol based on the cache coherence protocol used by AMD in their Opteron processors.

We evaluate the proposed protocols using full-system simulation of several parallel applications. The results of this evaluation show that, in absence of faults, our fault tolerance measures do not increase significantly the execution time of the applications and that their major cost is an increase in network traffic due to acknowledgment messages that ensure the reliable transference of ownership between coherence nodes, which are sent out of the critical path of cache misses. The results also show that a system using our protocols degrades gracefully when transient faults in the interconnection network actually happen, supporting faults rates which are much higher than expected in the real world with only a small increase in the execution time.

Acknowledgments

Completing this thesis has by no means been an easy task for me. However, the experience has been very good on the whole thanks to the help of many people that deserve credit for the final result. At least for the good parts.

First, I would like to thank my advisors, Prof. José Manuel García Carrasco and Dr. Manuel Acacio Sánchez for having me as a PhD student during almost five years. They have taught me how to perform research in computer architecture and they have always shown encouraging support for me and my work, helping me well beyond their professional duties. Prof. José Duato, coauthor of all my papers related with this thesis, has also provided very important technical advice, and he was the one who suggested this line of research in the first place.

I would like to thank Dr. Mikel Luján and Prof. Frank Wang for reviewing this thesis and providing useful comments.

I have had the luck to perform this work at the *Departamento de Ingeniería y Tecnología de Computadores* of the *Universidad de Murcia*. All the members of the department have always been ready to help me whenever I asked, and sometimes even before I asked. I am specially thankful for their help during my first years teaching computer architecture.

This thesis and my formation as a researcher have benefited greatly from the influence of the rest of the PhD students of my research group (GACOP). Amongst them, I should mention first Alberto Ros Bardisa, who has helped me in everything that I have done related to computer architecture since before I suspected that I would be one day writing a thesis in this area. I have also enjoyed many discussions (not only about computer architecture) with most of the students, specially those who joined the research group during my first years of PhD studies: Juan Manuel Cebrían González, José María Cecilia Canales, Daniel Sánchez Pedreño and Ruben Titos Gil. I also want to thank them for their work on keeping our computing cluster up and running.

My friends have helped me to avoid losing my sanity by keeping me away from my work from time to time. I have tried to make a list of those who I should mention by name, but I cannot come up with a list short enough to put it here. They know who they are, anyway.

In the end, this thesis has only been possible because my family has supported me during all these years. My parents, aunts and uncles deserve the largest share of the merits.

Contents

Abstract **11**

Acknowledgments **13**

Contents **15**

Table of contents in Spanish **19**

List of Figures **21**

List of Tables **25**

List of Acronyms **27**

Extended abstract in Spanish **29**

1 Introduction **49**
1.1 Tiled Chip Multiprocessors 50
1.2 Cache coherence and interconnection networks 51
1.3 Reliability of future electronic devices 54
1.4 Motivation 55
1.5 Contributions 57
1.6 Organization of this thesis 59

2 Related work **61**
2.1 Fault-tolerant multiprocessors 61
2.2 Fault tolerance in CMPs 64
2.3 Fault tolerance at the interconnection network level 66

3 General assumptions and fault tolerance requirements for cache coherence protocols 69
3.1 Base system assumptions 69
3.2 Requirements for a fault-tolerant cache coherence protocol 72

4 A token-based fault-tolerant cache coherence protocol 75
4.1 Problems caused by an unreliable interconnection network in TokenCMP 76
4.2 Fault tolerance measures introduced by FtTokenCMP 78
Token counting rules for reliable data transference 79
Fault detection 81
Avoiding data loss 81
Dealing with token loss 88
Dealing with faults in persistent requests 89
Token recreation process 92
4.3 Hardware overhead of FtTokenCMP 97

5 A directory-based fault-tolerant cache coherence protocol 101
5.1 Problems caused by an unreliable interconnection network in DirCMP 102
5.2 Fault tolerance measures introduced by FtDirCMP 104
Reliable data transmission 106
Faults detected by the *lost request timeout* 113
Faults detected by the *lost unblock timeout* 114
Faults detected by the *lost backup deletion acknowledgment timeout* . . 115
Faults detected with the *lost data timeout* 116
Reissuing requests and request serial numbers 118
Ping messages 123
Taking advantage of a point-to-point ordered network 125
5.3 Hardware overhead of FtDirCMP 126

6 A broadcast-based fault-tolerant cache coherence protocol 131
6.1 A cache coherence protocol for CMPs based on AMD's Hammer protocol 132
Problems caused by an unreliable interconnection network in HammerCMP 133
6.2 Fault tolerance measures introduced by FtHammerCMP 134

Summary of differences of the fault tolerance measures of FtHammerCMP and FtDirCMP . . . 135

7 Evaluation methodology 137
7.1 Failure model . . . 139
7.2 Configuration of the simulated systems . . . 141
7.3 Applications . . . 141
Server applications . . . 141
Apache . . . 141
SpecJbb . . . 144
Applications from SPLASH-2 . . . 144
Applications from ALTPBench . . . 146
Other scientific applications . . . 148

8 Evaluation results 149
8.1 Overhead without faults . . . 149
Execution time overhead . . . 150
Network traffic overhead . . . 153
8.2 Performance degradation under faults . . . 157
Execution slowdown . . . 157
Network traffic increase . . . 160
Effect of bursts of faults . . . 163
8.3 Adjusting fault tolerance parameters . . . 165
Adjusting the backup buffer size in FtTokenCMP . . . 165
Adjusting the fault detection timeouts . . . 167
Adjusting request serial number sizes for FtDirCMP and FtHammerCMP . . . 170
8.4 Hardware implementation overheads . . . 173

9 Conclusions and future ways 177
9.1 Conclusions . . . 177
9.2 Future ways . . . 180

A Base token-based protocol 183

B Base directory-based protocol 187

C Message dependency chains of our directory-based protocols 191

Bibliography **195**

Índice

Índice **19**

Lista de figuras **21**

Lista de tablas **25**

Lista de acrónimos **27**

Resumen en español **29**

1 Introducción **49**

2 Trabajo relacionado **61**

3 Suposiciones generales y requerimientos para los protocolos de coherencia tolerantes a fallos **69**

4 Diseño de un protocolo de coherencia basado en tokens tolerante a fallos **75**

5 Diseño de un protocolo de coherencia basado en directorio tolerante a fallos **101**

6 Diseño de un protocolo de coherencia basado en broadcast tolerante a fallos **131**

7 Metodología de evaluación **137**

8 Resultados de evaluación **149**

9 Conclusiones y vías futuras **177**

A Protocolo base basado en tokens **183**

B Protocolo base basado en directorio **187**

C Cadenas de dependencias de mensajes para nuestros protocolos basados en directorio **191**

Bibliografía **195**

List of Figures

0.1 Incremento en el tiempo de ejecución debida a las medidas de tolerancia a fallos en ausencia de fallos (resultados normalizados respecto a DirCMP) . . . 42
0.2 Tráfico en la red para cada tipo de mensaje medido en bytes normalizados respecto al protocolo no tolerante a fallos más similar . . . 43
0.3 Degradación del rendimiento bajo diferentes tasas de fallos (en mensajes corrompidos por millón de mensajes que viajan por la red) para cada protocolo tolerante a fallos respecto a su correspondiente protocolo base no tolerantes a fallos . . . 45

1.1 Tiled Chip Multiprocessors . . . 51

4.1 Message exchange example of a cache-to-cache transfer using owned data loss avoidance in FtTokenCMP . . . 84
4.2 Message exchange in FtTokenCMP without backup buffer for a miss including the required previous write-back . . . 86
4.3 Message exchange example of a token loss detected using the *lost token timeout* . . . 90
4.4 Message exchange in a token recreation process (used in this case to recover from the loss of an *ownership acknowledgment*) . . . 99

5.1 Message exchange for a cache-to-cache write miss . . . 110
5.2 Message exchange for an L1 write miss that hits in L2 . . . 111
5.3 L2 miss optimization of ownership transference . . . 112
5.4 Transaction where the *lost data timeout* detects data which has been wrongly discarded . . . 118
5.5 Transaction where request serial numbers are needed to avoid incoherency in a point-to-point ordered network . . . 128

5.6 Transaction where request serial numbers avoid using stale data in a point-to-point unordered network 129

8.1 Execution time overhead of the fault-tolerant cache coherence protocols in absence of faults 152
8.2 Network overhead for each category of message in terms of number of messages. Results are normalized with respect to the total number of messages of the most similar non fault-tolerant protocol 154
8.3 Relative network traffic in terms of messages for each protocol normalized with respect to DirCMP 155
8.4 Network overhead for each type of message in terms of number of bytes. Results are normalized with respect to the total number of bytes of the most similar non fault-tolerant protocol 156
8.5 Performance degradation with different fault rates (in messages corrupted per million of messages that travel through the network) for each fault-tolerant protocol with respect to its non fault-tolerant counterpart 158
8.6 Average performance degradation with different fault rates (in messages corrupted per million of messages that travel through the network) for each fault-tolerant protocol with respect to its non fault-tolerant counterpart 159
8.7 Network usage increase with different fault rates (in messages corrupted per million of messages that travel through the network) for each fault-tolerant protocol with respect to its non fault-tolerant counterpart 161
8.8 Average network usage increase with different fault rates (in messages corrupted per million of messages that travel through the network) for each fault-tolerant protocol with respect to its non fault-tolerant counterpart 162
8.9 Performance degradation with burst faults of several lengths for each fault-tolerant protocol with respect to its non fault-tolerant counterpart. The total fault rate is fixed to 125 corrupted messages per million of messages that travel through the network 164
8.10 Execution time of FtTokenCMP normalized with respect to TokenCMP when no faults occur and using different backup buffer sizes 166

8.11 Performance degradation of the fault-tolerant protocols with a constant fault rate of 250 lost messages per million of messages travelling through the network and with different values for the fault detection timeouts, using the same value for all the timeouts of each protocol . 168
8.12 Maximum miss latency (in cycles) of each cache coherence protocol without faults 169
8.13 Required request serial number bits to be able to discard every old response to a reissued message when faults occur. In the fault-free case, no bits are required at all in any of our tests 172

C.1 Message dependency chains for the DirCMP protocol 193
C.2 Message dependency chains for the FtDirCMP protocol 194

List of Tables

0.1 Resumen de *timeouts* usados en FtTokenCMP 35
0.2 Resumen de *timeouts* usados en FtDirCMP 37
0.3 Características de las arquitecturas simuladas 41
0.4 *Benchmarks* y tamaños de problema usados en las simulaciones 42

4.1 Summary of the problems caused by loss of messages in TokenCMP . 76
4.2 Summary of timeouts used in FtTokenCMP 82
4.3 Summary of the problems caused by loss of messages in FtTokenCMP and the detection and recovery mechanisms employed 83

5.1 Message types used by DirCMP and the effect of losing them 103
5.2 Additional message types used by FtDirCMP 104
5.3 Summary of timeouts used in FtDirCMP 105
5.4 Timeout which detects the loss of each type of message in FtDirCMP 106

6.1 Summary of timeouts used in FtHammerCMP 135

7.1 Characteristics of simulated architectures 142
7.2 Benchmarks and input sizes used in the simulations 143

A.1 Correspondence of token counting states with MOESI states 185

B.1 Message types used by DirCMP 188

List of Acronyms

BER: Backward Error Recovery.

CARER: Cache Aided Rollback Error Recovery.

CCA: Core Cannibalization Architecture.

CMP: Chip Multiprocessor.

CRC: Cyclic Redundancy Code.

DCT: Discrete Cosine Transform.

DMR: Dual Modular Redundancy.

DRSM-L: Distributed Recoverable Shared Memory with Logs.

DUE: Detected but Unrecoverable Error.

ECC: Error Correcting Code.

EDC: Error Detecting Code.

EMI: Electromagnetic Interference.

FER: Forward Error Recovery.

FFT: Fast Fourier Transform.

FIFO: First In First Out.

GEMS: General Execution-driven Multiprocessor Simulator.

HTML: Hypertext Markup Language.

HTTP: Hypertext Transfer Protocol.

IDCT: Inverse Discrete Cosine Transform.

ILP: Instruction Level Parallelism.

LPC: Low Power Codes.

M-CMP: Multiple CMP.

MOESI: Modified Owned Exclusive Shared Invalid (set of cache coherence states).

MPEG: Moving Picture Experts Group.

MSHR: Miss Status Holding Register.

MSSG: MPEG Software Simulation Group.

MTTR: Mean Time To Recover.

RSM: Recoverable Shared Memory.

SDC: Silent Data Corruption.

SECDED: Single Error Correction / Double Error Detection.

SLICC: Specification Language Including Cache Coherence.

SMP: Symmetric Multiprocessors.

SMT: Simultaneous Multithread.

TCP: Transmission Control Protocol.

TLP: Thread Level Parallelism.

TMR: Triple Modular Redundancy.

TRUSS: Total Reliability Using Scalable Servers.

VLSI: Very Large Scale of Integration.

Resumen

Introducción

La mejora en la tecnología de semiconductores está haciendo posible que cada vez se puedan incluir más transistores en un solo chip. Los arquitectos de computadores deben decidir de qué forma usar estos transistores para obtener el mejor rendimiento posible. Tradicionalmente, se viene considerando que la faceta más importante del rendimiento es la velocidad de ejecución de las aplicaciones que finalmente se ejecutan en esos chips.

Sin embargo, el rendimiento no puede ser entendido solo desde el punto de vista de la velocidad de ejecución. Otros factores como la dificultad para programar el procesador resultante o su consumo energético también son importantes. El peso relativo de cada variable depende del uso final del chip.

Uno de los aspectos del rendimiento de un sistema es la fiabilidad del mismo. La misma tendencia de miniaturización de los transistores que permite disponer de mayor número de ellos, también hace que la fiabilidad de los mismos disminuya. Debido a esto, es necesario aplicar técnicas de tolerancia a fallos a varios niveles del sistema. Hasta hace poco, estas técnicas eran importantes solo en determinadas aplicaciones críticas. Sin embargo, actualmente son necesarias para todo tipo de sistemas para mantener niveles de fiabilidad aceptables.

Los fallos cuya frecuencia está aumentando más rápidamente debido a las tendencias tecnológicas son los fallos transitorios. Cuando ocurre un fallo transitorio, un componente produce una salida incorrecta pero continua funcionando correctamente después del fallo [87, 89].

Uno de los componentes de un CMP (*Chip Multiprocessor* [11, 50]) que se verá afectado por los fallos transitorios mencionados anteriormente es la red de interconexión. Ésta se usa para permitir el intercambio de los mensajes del

protocolo de coherencia de caché [27] entre procesadores, cachés y memorias; y su rendimiento es clave para el rendimiento general del sistema.

La red de interconexión es especialmente vulnerable a los fallos transitorios debido a que ocupa una porción importante de la superficie del chip, lo que aumenta la probabilidad de impacto de partículas. Además, está construida usando hilos más largos en promedio que el resto de componentes, lo que la hace más propensa a acoplamiento (*crosstalk*) e incrementa aún más las probabilidades de fallos transitorios [34]. Los fallos transitorios en la red de interconexión provocan la corrupción de los mensajes del protocolo de coherencia de caché, lo que en última instancia causa interbloqueos o corrupción de datos no detectada.

Una forma de solucionar los problemas causados por los fallos transitorios en la red de interconexión es construir una red de interconexión tolerante a fallos. Hay múltiples propuestas de redes de interconexión tolerantes a fallos que se mencionan en la sección de trabajo relacionado de esta tesis.

A diferencia de otros trabajos, en esta tesis se propone tratar con los fallos transitorios en la red de interconexión a nivel del protocolo de coherencia de caché, en lugar de a nivel de la propia red. Este enfoque tiene las siguientes ventajas:

- El protocolo de coherencia tiene más información sobre el significado de los mensajes y sobre los efectos de perder cada mensaje. De esta forma, se pueden utilizar opciones más inteligentes para obtener el mismo nivel o mayor de tolerancia a fallos con una sobrecarga menor. Por su parte, las medidas de tolerancia a fallos a nivel de la red de interconexión tienen que tratar a todos los mensajes por igual.

- La implementación de medidas de tolerancia a fallos a nivel de la red de interconexión limita la capacidad de maniobra del diseñador de la red de interconexión para optimizar agresivamente otros aspectos del rendimiento de la misma.

- Como se mostrará más adelante, un protocolo de coherencia tolerante a fallos se comporta la mayoría del tiempo casi igual que un protocolo de coherencia convencional. En ausencia de fallos, solo se observan diferencias cuando un mensaje crítico necesita ser transmitido.

Por contra, implementar las medidas de tolerancia a fallos a nivel del protocolo de coherencia de caché también tiene sus desventajas. La mayor desventaja es tener que modificar el propio protocolo de coherencia, cuyo rendimiento también

es clave para el rendimiento general del sistema y cuyo diseño suele considerarse complejo. En esta tesis mostramos que el aumento de complejidad es asumible y la sobrecarga en el rendimiento es pequeña.

En nuestro modelo de fallos, suponemos que, desde el punto de vista del protocolo de coherencia, la red de interconexión entrega correctamente un mensaje o bien no lo entrega en absoluto. Esto es fácil de conseguir añadiendo códigos de detección de errores [58, 89] que sean comprobados a la llegada del mensaje, y descartando los mensajes que se detecten corruptos. Los mensajes inesperados se supone que han sido mal enrutados y se descartan también.

Aunque en esta tesis suponemos que los mensajes son descartados debido a errores transitorios, las mismas técnicas se podrían utilizar si los mensajes se descartasen por cualquier otra razón como, por ejemplo, falta de espacio en buffers de la red de interconexión. Esta propiedad puede ser útil para los diseñadores de red.

En esta tesis se hacen las siguientes aportaciones:

- Se identifican los problemas causados por una red de interconexión no fiable en CMPs con coherencia de caché.

- Se proponen modificaciones a los protocolos de coherencia de caché que es más probable que se usen en *tiled CMPs* [127, 128, 14] para soportar el descarte de algunos mensajes por la red de interconexión sin añadir excesiva sobrecarga.

- Un protocolo de coherencia de caché que extiende un protocolo basado en *tokens* con medidas de tolerancia a fallos, y modificaciones a las reglas genéricas de conteo de *tokens* para asegurar la transferencia fiables de datos y propiedad de los datos.

- Un protocolo de coherencia de caché que extiende un protocolo de coherencia basado en directorio con medidas de tolerancia a fallos.

- Un protocolo de coherencia de caché que extiendexs un protocolo de coherencia tipo *snoopy* basado en broadcast con medidas de tolerancia a fallos.

- Dado que nuestros protocolos de coherencia imponen menos requerimientos a la red de interconexión que otros, esperamos que sean útiles para simplificar el diseño de la red y permitir la aplicación de técnicas agresivas para mejorar el rendimiento.

- Se evalúan los protocolos tolerantes a fallos usando simulación del sistema completo para comprobar su efectividad y para medir la sobrecarga introducida en ausencia de fallos y la degradación del rendimiento bajo distintas tasas de fallo.

Premisas generales y requerimientos para los protocolos de coherencia tolerantes a fallos

Para desarrollar y evaluar las propuestas de esta tesis, partimos de una arquitectura basada en *tiled CMPs* [127,128,14]. La gran cantidad de transistores disponibles en un solo chip ha obligado a replantearse cuál es la mejor forma de organizarlos. Mientras que hasta hace pocos años se trataba de obtener el mayor rendimiento posible aumentando la frecuencia del circuito y explotando el paralelismo a nivel de instrucción (ILP), los límites físicos y la gran complejidad necesaria para obtener pequeñas mejoras hacen estas vías poco atractivas actualmente. Hasta la fecha, la forma más práctica que se ha propuesto para hacer uso de estos transistores es construir multiprocesadores en un solo chip (CMPs) que exploten el paralelismo a nivel de hilo (TLP) además, o incluso en lugar, de a nivel de instrucción. Para reducir aún más la complejidad de los CMPs, los *tiled CMPs* se construyen replicando *tiles* (baldosas) idénticos entre sí y conectados mediante una red de interconexión punto a punto. Cada *tile* está formado por un núcleo de procesador, una caché privada, una parte de una caché compartida y un interfaz de red. Los *tiled CMPs* son una arquitectura muy probable para futuros diseños de chips con gran número de procesadores.

Para la evaluación, se asume una red de interconexión punto a punto con topología de malla y enrutamiento determinista, aunque los protocolos en sí no requieren ninguna característica concreta de la red. En particular, no se hace ninguna suposición sobre el orden de entrega de los mensajes.

Aunque en esta tesis se asume la arquitectura anteriormente descrita como base, las propuestas que se presentan son también aplicables a cualquier tipo de CMP con red de interconexión punto a punto e incluso para sistemas multiprocesadores en varios chips.

Nuestras propuestas presuponen que la memoria y las cachés están protegidas mediante ECC (*Error Correction Codes*) u otros medios. Es decir, los datos solo se tienen que proteger mientras viajan por la red.

Nuestro objetivo es diseñar protocolos de coherencia de caché que garanticen la correcta ejecución de aplicaciones paralelas incluso sobre una red de

interconexión no fiable, sin tener que realizar nunca *rollback* debido a errores en dicha red. Así mismo, evitaremos la introducción generalizada de mensajes de reconocimiento punto a punto, para evitar incrementar innecesariamente la latencia de las transacciones de coherencia, el tráfico de la red, o el tamaño de los buffers.

Una red de interconexión no fiable causa una serie de problemas en un *tiled CMP*: corrupción silenciosa de datos, violación no detectada de la coherencia de memoria, interbloqueos (*deadlock*), errores detectables pero no recuperables (interbloqueos, corrupción o pérdida de datos) o degradación del rendimiento.

Para evitar los problemas mencionados anteriormente, un protocolo de coherencia de caché tolerante a fallos debe ofrecer las siguientes características:

Evitar corrupción de datos: El sistema debe ser capaz de detectar mensajes que han sido corrompidos y descartarlos. Para ello, se pueden utilizar códigos de detección de errores (EDC). Algunos sistemas ya usan EDC en los mensajes de la red de interconexión para evitar la corrupción no detectada de datos aunque no dispongan de otras medidas de tolerancia a fallos en el protocolo de coherencia.

Evitar pérdida de datos: Debido a que los mensajes pueden ser descartados, es necesario que se evite enviar datos a través de la red de interconexión sin conservar al menos una copia de los mismos para que puedan ser recuperados en caso necesario.

Evitar incoherencias: Un protocolo de coherencia tolerante a fallos necesita asegurar que la coherencia de caché no se puede llegar a violar debido a que un mensaje no llega a su destino. En particular, los mensajes de invalidación deben requerir el envío de mensajes de reconocimiento, ya sea al peticionario o al directorio. Estos mensajes de reconocimiento ya son necesarios en la mayoría de los protocolos de coherencia para evitar condiciones de carrera.

Detectar y recuperar interbloqueos: Cuando un mensaje de coherencia es descartado, esto provoca en la mayoría de las ocasiones un interbloqueo. Es por tanto necesario incluir mecanismos de detección y resolución de interbloqueos.

Para poder ser útil, en ausencia de fallos, un protocolo de coherencia tolerante a fallos debe, además de incluir las características arriba mencionadas, ofrecer un rendimiento muy similar al de un protocolo de coherencia no tolerante a fallos que usara una red de interconexión fiable.

Diseño de un protocolo de coherencia de caché basado en *tokens* tolerante a fallos

Nuestro primer protocolo de coherencia tolerante a fallos, FtTokenCMP, se basa en la metodología de conteo de *tokens* [68,67]. En concreto, se basa en el protocolo TokenCMP [70]. Decidimos usar esta metodología debido a la mayor simplicidad proporcionada por la separación entre el *substrato de corrección* y la *política de rendimiento*, y el uso de peticiones transitorias y *timeouts* por parte de TokenCMP, ya que esto minimiza las modificaciones necesarias para la detección de fallos. Las medidas de tolerancia a fallos propuestas para FtTokenCMP son también aplicables (con ligeras modificaciones) a cualquier otro protocolo de coherencia basado en *tokens*.

FtTokenCMP utiliza unas reglas de transferencia de *tokens* modificadas que evitan la pérdida de datos en caso de pérdida de mensajes. Las modificaciones incluyen la adición de un *backup token* y reglas para su transferencia que aseguran que siempre hay una copia válida de los datos y un máximo de una copia de *backup*. Con estas reglas, se garantiza que siempre existe al menos una copia de los datos de una línea de memoria además de la copia que viaja por la red de interconexión. Para ello, en algunos casos se utiliza un par de mensajes de reconocimiento: el *reconocimiento de propiedad* y el *reconocimiento de eliminación de backup*. Estos mensajes se envían fuera del camino crítico de los fallos de caché.

Se muestra que en TokenCMP todas las pérdidas de mensajes son, o bien inocuas, o bien causan un interbloqueo (además de una posible pérdida de datos). Por tanto, FtTokenCMP puede utilizar un conjunto de *timeouts* para detectar cualquier fallo. El conjunto de *timeouts* se resume en la tabla 0.1.

FtTokenCMP posee dos mecanismos para recuperarse tras un fallo transitorio detectado por alguno de los cuatro *timeouts*. Estos mecanismos entrarán en acción de forma muy infrecuente, por lo que su eficiencia no es un objetivo prioritario.

El más sencillo de los mecanismos consiste en el envío de un mensaje de comprobación (*ping*) cuando se ha detectado la (posible) pérdida de un mensaje de desactivación de petición persistente. La recepción de este mensaje provoca el reenvío del mensaje de desactivación si es necesario.

El segundo mecanismo, y el más frecuentemente utilizado, es la *recreación de tokens*. Este mecanismo es utilizado cuando se disparan la mayoría de los *timeouts* y es orquestado de forma centralizada por el controlador de memoria. Tras la ejecución de una recreación de *tokens* para una línea de memoria, se garantiza que existe una sola copia de los datos y el número de *tokens* necesario para el

Tabla 0.1: Resumen de *timeouts* usados en FtTokenCMP

Timeout	**¿Cuándo se activa?**	**¿Dónde se activa?**	**¿Cuándo se desactiva?**	**¿Qué ocurre cuando se dispara?**
Token perdido	Se activa una petición persistente.	La caché que la activa.	Se satisface o desactiva la petición persistente.	Se solicita una recreación de *tokens*.
Datos perdidos	Se entra en estado de backup (se envía el token propietario).	La caché que tiene el backup.	Se abandona el estado de backup (llega el *reconocimiento de propiedad*).	Se solicita una recreación de *tokens*.
Reconocimiento de eliminación de backup perdido	Se entra en estado de propiedad bloqueada.	La caché que posee el token propietario.	Se abandona el estado bloqueado (llega el *reconocimiento de eliminación de backup*).	Se solicita una recreación de *tokens*.
Desactivación persistente perdida	Se activa una petición persistente de una caché distinta.	Todas las cachés (en la tabla de peticiones persistentes).	Se desactiva la petición persistente.	Se envía un *ping de petición persistente*.

correcto funcionamiento del protocolo. Los datos se recuperan usando una copia de *backup* si es necesario.

Dado que el tiempo necesario para completar una transacción de coherencia no puede ser acotado de forma estricta, los *timeouts* pueden dispararse en ocasiones cuando no ha ocurrido ningún fallo si un mensaje tarda mucho más de lo normal en alcanzar su destino. Es decir, nuestro sistema de detección de fallos puede producir falsos positivos. Por tanto, nuestro método de recuperación en caso de error debe de tener esto en cuenta. Para ello, se introducen los *números de serie de tokens*. Cada línea de memoria tiene asociado un *número de serie de tokens*, que se incluye en cada mensaje que transfiera *tokens* de una caché a otra. El mecanismo de *recreación de tokens* incrementa el número de serie asociado con la línea de memoria afectada y los mensajes que se reciben con un número de serie incorrecto se descartan. Estos números de serie valen inicialmente cero, y solo es necesario almacenarlos cuando su valor es distinto del inicial. Por tanto,

una pequeña tabla asociativa en cada caché es suficiente para almacenar aquellos números de serie distintos de cero.

El hardware extra necesario para implementar FtTokenCMP con respecto a TokenCMP es pequeño. Es necesario añadir un solo contador por cada tabla de peticiones persistentes, ya que se reutiliza para la mayoría de los *timeouts* el contador ya requerido por el *timeout* de las peticiones transitorias. El mayor coste es debido a la tabla de *números de serie de tokens* que es necesario añadir en cada caché (en nuestra pruebas utilizamos 2 bits para representar cada número de serie, y 16 entradas en cada tabla). También es conveniente añadir una pequeña *caché de backups* (o un *writeback buffer*) en cada caché L1 para evitar incrementar el tiempo de ejecución.

Diseño de un protocolo de coherencia de caché basado en directorio tolerante a fallos

Mientras que la coherencia basada en *tokens* no ha sido utilizada todavía en ningún procesador real, los protocolos de coherencia de caché basados en directorios son una de las opciones más populares en las nuevas propuestas [72]. Por tanto, hemos estudiado cómo añadir técnicas de tolerancia a fallos similares a las de FtTokenCMP a un protocolo basado en directorios. El resultado de ello es FtDirCMP, que añade tolerancia a fallos a un protocolo de coherencia de caché basado en directorios (DirCMP).

El diseño de FtDirCMP está muy influenciado por el de FtTokenCMP. Ambos asumen el mismo modelo de fallos, aseguran la transferencia fiable de datos usando mensajes de reconocimiento solo para unos pocos mensajes, utilizan *timeouts* para detectar los fallos y utilizan mensajes de *ping* para recuperarse de algunos casos de interbloqueo. El conjunto de *timeouts* usado por FtDirCMP se muestra en la tabla 0.2.

Sin embargo, algunos mecanismos de tolerancia a fallos son significativamente diferentes entre FtDirCMP y FtTokenCMP. Las principales diferencias son las siguientes:

- FtDirCMP no tiene un mecanismo de recuperación centralizado como la *recreación de tokens*. Cuando se detecta un potencial fallo, FtDirCMP usa mensajes de *ping* o reenvía la petición afectada.
- FtDirCMP usa *números de serie de petición* para evitar crear incoherencias debido a las respuestas viejas a peticiones reenviadas en caso de falsos

Tabla 0.2: Resumen de *timeouts* usados en FtDirCMP

Timeout	**¿Cuándo se activa?**	**¿Dónde se activa?**	**¿Cuándo se desactiva?**	**¿Qué ocurre cuando se dispara?**
Petición perdida	Se envía una petición.	En la caché L1 peticionaria.	Se satisface la petición.	La petición se reenvía con un nuevo número de serie.
Unblock perdido	Se contesta una petición (incluso *write-backs*).	La caché L2 o controlador de memoria que contesta.	El *unblock* (o *write-back*) se recibe.	Se envía un *UnblockPing*/*WbPing* a la caché que debería haber enviado el *Unblock* o *write-back*.
Reconocimiento de eliminación de backup perdido	Se entra en estado de propiedad bloqueada.	La caché que posee el token propietario.	Se abandona el estado bloqueado (llega el *reconocimiento de eliminación de backup*).	Se solicita una recreación de *tokens*.
Datos perdidos	Se envía datos en propiedad a través de la red.	El nodo que envía los datos.	El mensaje *AckO* se recibe.	Se envía un *OwnershipPing*.

positivos. Su función es análoga a los *números de serie de tokens*, pero los *números de serie de petición* son más escalables y fáciles de implementar, ya que están asociados a cada petición y solo necesitan ser conocidos por los nodos que intervienen en la petición y hasta que la petición se resuelve, mientras que los *números de serie de tokens* están asociados a cada dirección de memoria, necesitan ser conocidos por todos los nodos del sistema, y tienen duración indefinida.

Para evitar la pérdida de datos en caso de pérdida de mensajes, FtDirCMP usa un mecanismo que es igual en esencia al utilizado por FtTokenCMP. Para implementarlo, FtDirCMP añade nuevos estados al conjunto básico de estados MOESI [27]: estado de *backup* (B) y estados con propiedad bloqueada (Mb, Ob y Eb). Una línea se conserva en la caché en estado de *backup* cuando la caché

cede la propiedad de la línea a otra y se mantiene en dicho estado hasta que se comprueba, mediante la recepción de un *reconocimiento de propiedad*, que los datos se han transferido correctamente. Los estados de propiedad bloqueada sirven para asegurar que nunca hay más de una copia de *backup*, lo cual complicaría significativamente la recuperación en caso de fallo. Una línea de memoria se mantiene en uno de dichos estados desde que se reciben los datos en propiedad hasta que se recibe un *reconocimiento de eliminación de backup*.

FtDirCMP asume que la red de interconexión no es ordenada punto a punto, como la mayoría de los protocolos de directorio. Sin embargo, es posible simplificar las medidas de tolerancia a fallos si se asume que, como es muy habitual, la red de interconexión mantiene el orden de mensajes punto a punto (aunque no mantenga el orden total de los mensajes). En ese caso, no es necesario implementar el *timeout de datos perdidos*.

FtDirCMP necesita algo más de hardware que DirCMP. Es necesario añadir contadores para implementar los *timeouts* de detección de fallos, añadir campos en los *registros de estado de fallo* (MSHR) para almacenar los *números de serie de petición*, y se necesita también que el nodo que contesta almacene la identidad del receptor de los mensajes de datos en propiedad hasta que se reciba un *reconocimiento de propiedad*. Aunque el protocolo utiliza hasta cuatro *timeouts* en cada transacción de coherencia, nunca hay más de uno activo en el mismo nodo y para la misma petición. Finalmente, el número de redes virtuales puede que tenga que ser incrementado para evitar interbloqueos, dependiendo de la topología de red particular y de la estrategia que use para evitar interbloqueos.

Diseño de un protocolo de coherencia de caché basado en broadcast tolerante a fallos

El último de los protocolos de coherencia tolerantes a fallos presentado en esta tesis es FtHammerCMP. Este protocolo extiende con medidas de tolerancia a fallos a HammerCMP, un protocolo de coherencia basado en el utilizado por AMD en sus procesadores Opteron [5, 54, 93] y que tiene características de los protocolos basados en fisgoneo (*snoopy*) aunque funciona sobre una red punto a punto, como los protocolos basados en directorios. El protocolo original de AMD estaba enfocado a sistemas multiprocesadores de pequeño o medio tamaño. HammerCMP es una adaptación de dicho protocolo a entornos CMP y con pequeñas optimizaciones.

Estrictamente, HammerCMP se podría clasificar como un protocolo basado

en directorios, pero sin información de directorio (Dir_0B) [2]. Por tanto, las modificaciones para soportar tolerancia a fallos que introduce FtHammerCMP están basadas en las de FtDirCMP. FtHammerCMP usa los mismos mecanismos para recuperarse cuando se detecta un fallo transitorio, para asegurar la transferencia fiable de datos y para detectar los fallos.

FtHammerCMP usa solo tres de los *timeouts* empleados por FtDirCMP (ver tabla 0.2): el *timeout de petición perdida*, el *timeout de unblock perdido* y el *timeout de reconocimiento de eliminación de backup perdido*. No se requiere el *timeout de datos perdidos*, ya que se asume una red ordenada punto a punto.

Debido a que el protocolo HammerCMP es más sencillo que el DirCMP, la sobrecarga hardware introducida por las medidas de tolerancia a fallos es relativamente mayor en el caso de FtHammerCMP que en el de FtDirCMP.

Metodología de evaluación

Se ha realizado una evaluación experimental de los protocolos de coherencia presentados en esta tesis. Los objetivos de esta evaluación han sido:

- Comprobar la efectividad de las medidas de tolerancia a fallos. Es decir, que las aplicaciones paralelas se ejecutan correctamente en presencia de fallos transitorios en la red de interconexión.

- Medir la sobrecarga introducida por las medidas de tolerancia a fallos cuando no ocurren fallos. Hemos medido tanto el incremento en el tiempo de ejecución como el incremento en el tráfico de red. Estimamos que el incremento en consumo de energía de nuestros protocolos estará dominado por el incremento en el tráfico de red.

- Medir la degradación del rendimiento de los programas ejecutándose en un sistema que use nuestros protocolos de coherencia bajo distintas tasas de fallos transitorios en la red de interconexión.

- Determinar valores adecuados para los parámetros de configuración introducidos por nuestros protocolos.

Nuestra metodología se basa en simulación de sistema completo. Hemos implementado todos los protocolos de coherencia mencionados en una versión modificada del simulador Multifacet GEMS [69] de la Universidad de Winsconsin.

GEMS es un entorno de simulación basado en Virtutech Simics [65], al que complementa con modelos detallados del sistema de memoria y del procesador.

El simulador descrito anteriormente lo utilizamos para simular la ejecución de un conjunto de aplicaciones paralelas. Debido al no determinismo de la ejecución paralela, realizamos varias ejecuciones con distintas semillas aleatorias para cada experimento y calculamos el intervalo con 95 % de confianza de los resultados. Solo medimos la parte paralela de las aplicaciones.

Nuestro modelo de fallos supone que los fallos transitorios corrompen los mensajes y estos son descartados cuando se reciben después de comprobar su código de detección de errores, por lo que desde el punto de vista del protocolo, los mensajes se entregan correctamente o no se entregan. En estas condiciones, cuando realizamos inyección de fallos, los mensajes son seleccionados de forma aleatoria y descartados de manera que se obtenga una tasa media de fallos fija, medida en mensajes descartados por millón de mensajes que viajan por la red de interconexión. También se evalúa el efecto de la aparición de ráfagas de fallos transitorios que afectan a varios mensajes consecutivos.

Además, se ha realizado una validación funcional de los protocolos de coherencia usando tests aleatorios, para buscar casos muy infrecuentes que no se presentan en la simulación de las aplicaciones. Estas pruebas incluyen inyección de fallos.

La configuración de los sistemas simulados se puede ver en la tabla 0.3. Para la evaluación se han usado las aplicaciones y tamaños de problema que se muestran en la tabla 0.4.

Resultados de evaluación

Hemos medido el efecto que las medidas de tolerancia a fallos tienen en el tiempo de ejecución de las aplicaciones en ausencia de fallos transitorios en la red de interconexión. En la figura 0.1 se muestra el tiempo de ejecución normalizado de los seis protocolos mencionados en esta tesis. Como se puede observar, el tiempo de ejecución de las aplicaciones con los protocolos tolerantes a fallos es prácticamente el mismo que el de sus correspondientes protocolos no tolerantes a fallos. Esto es así porque, mientras no ocurran fallos, la única diferencia observable entre el comportamiento de los protocolos tolerantes a fallos con respecto a los no tolerantes a fallos es el envío de los mensajes de reconocimiento usados para garantizar la transferencia segura de datos en propiedad, y estos mensajes se envían fuera del camino crítico de los fallos de cache.

Tabla 0.3: Características de las arquitecturas simuladas

Sistema *tiled* CMP de 16 nodos	
Parámetros de procesador	
Frecuencia del procesador	2 GHz
Parámetros de caché	
Tamaño de línea de caché	64 bytes
Caché L1:	
Tamaño	32 KB
Asociatividad	4 vías
Tiempo de acierto	3 ciclos
Caché L2 compartida:	
Tamaño	1024 KB
Asociatividad	4 vías
Tiempo de acierto (mismo *tile*)	15 ciclos
Parámetros de memoria	
Tiempo de acceso a memoria	160 ciclos
Parámetros de red	
Topología	Malla 2D
Tamaño de mensaje sin datos	8 bytes
Tamaño de mensaje con datos	72 bytes
Ancho de banda por enlace	64 GB/s

(a) Parámetros comunes a todas las configuraciones.

Parámetros específicos para FtTokenCMP	
Timeout de token perdido	2000 ciclos
Timeout de datos perdidos	2000 ciclos
Timeout de reconocimiento de eliminación de backup perdido	2000 ciclos
Timeout de desactivación persistente perdida	2000 ciclos
Tamaño de los números de serie de token	2 bits
Tamaño de la tabla de números de serie de token	16 entradas
Tamaño de buffer de backups	2 entradas
Parámetros específicos para FtDirCMP	
Timeout de petición perdida	1500 ciclos
Timeout de *unblock* perdido	1500 ciclos
Timeout de reconocimiento de eliminación de backup perdido	1500 ciclos
Timeout de datos perdidos	1500 ciclos
Bits de números de serie de petición por mensaje	8 bits
Parámetros específicos para FtHammerCMP	
Timeout de petición perdida	1500 ciclos
Timeout de *unblock* perdido	1500 ciclos
Timeout de reconocimiento de eliminación de backup perdido	1500 ciclos
Bits de números de serie de petición por mensaje	8 bits

(b) Parámetros específicos a cada protocolo de coherencia tolerante a fallos

Tabla 0.4: *Benchmarks* y tamaños de problema usados en las simulaciones

Benchmark	**Tamaño de problema**
	***Benchmarks* de servidores**
Apache	10 000 transacciones HTTP
SpecJbb	8 000 transacciones
	***Benchmarks* de SPLASH-2**
Barnes	8 192 cuerpos, 4 pasos
FFT	64K números complejos de doble precisión
LU	Matriz de 512×512
Ocean	Océano de 258×258
Radix	1M de claves, raíz 1024
Raytrace	10MB, escena teapot.env
WaterNSQ	512 moléculas, 4 pasos
	Otros *benchmarks* científicos
Em3d	38 400 nodos, grado 2, 15 % remotos y 25 pasos
Unstructured	Mesh.2K, 5 pasos
	***Benchmarks* de ALPBench**
FaceRec	Entrada de entrenamiento de ALPBench
MPGdec	525_tens_040.mv2
MPGenc	Salida de MPGdec
SpeechRec	Entrada por defecto de ALPBench

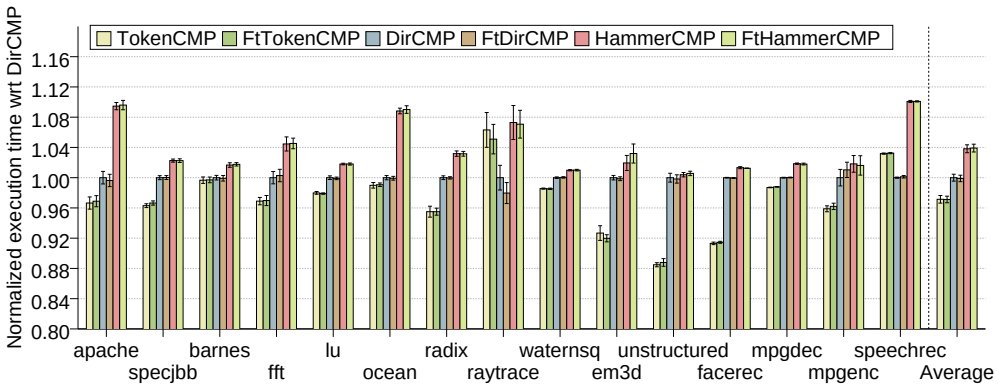
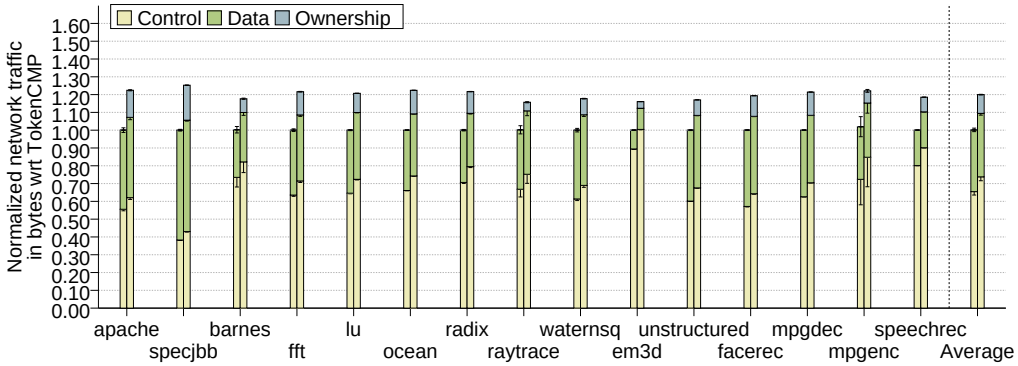
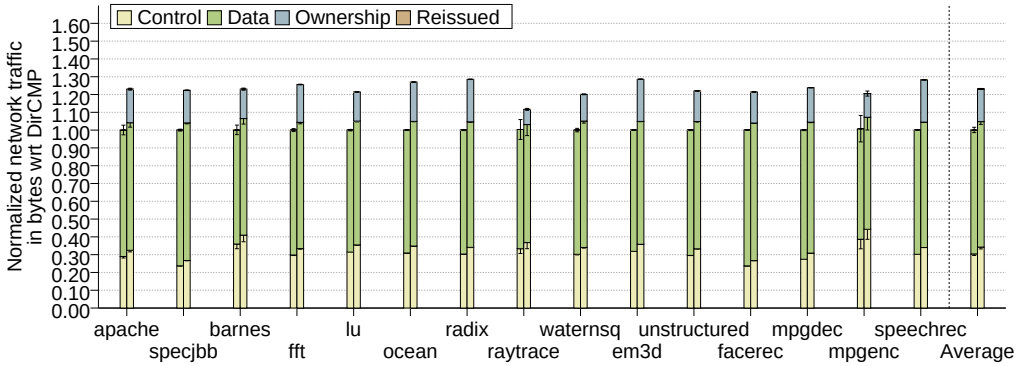


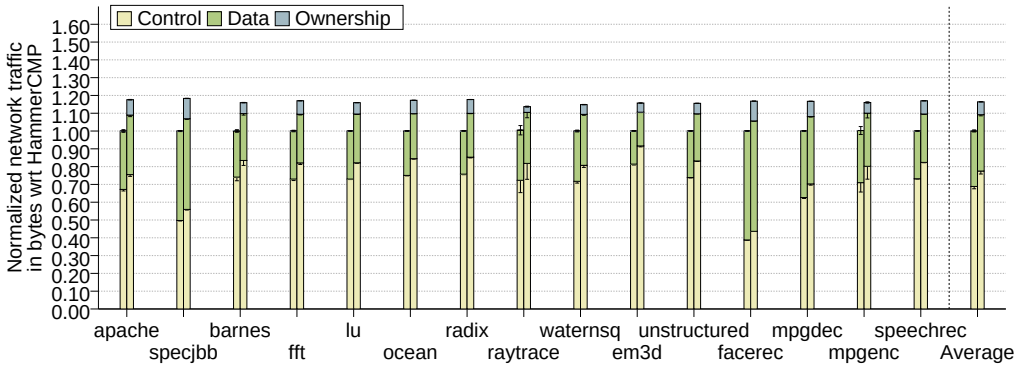
Figura 0.1: Incremento en el tiempo de ejecución debida a las medidas de tolerancia a fallos en ausencia de fallos (resultados normalizados respecto a DIRCMP)



(a) TokenCMP y FtTokenCMP



(b) DirCMP y FtDirCMP



(c) HammerCMP y FtHammerCMP

Figura 0.2: Tráfico en la red para cada tipo de mensaje medido en bytes normalizados respecto al protocolo no tolerante a fallos más similar

Sin embargo, este intercambio de reconocimientos provoca un incremento en el tráfico que circula por la red de interconexión. Este incremento, que es el mayor coste de nuestras medidas de tolerancia a fallos, provoca un incremento en el consumo de energía y podría tener efectos adversos en el tiempo de ejecución si la red de interconexión no tuviera suficiente ancho de banda disponible. En la figura 0.2 se muestra esta sobrecarga, medida en bytes, clasificada por tipo de mensaje y normalizada respecto a los protocolos no tolerantes a fallos. La sobrecarga media varía para cada protocolo, desde el 17 % de FtHammerCMP al 25 % de FtDirCMP. Cuanto más eficiente fuera el protocolo original en cuanto a tráfico de red, mayor es la sobrecarga. Esto es debido a que el número extra de mensajes depende de la frecuencia de los cambios de propiedad de los bloques de caché, que es un valor que depende mucho más de la aplicación que del protocolo de coherencia utilizado.

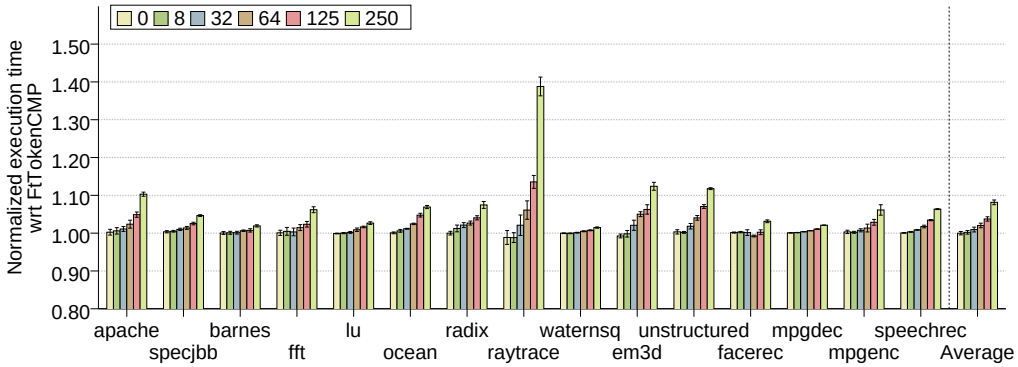
Cuando se presentan fallos transitorios en la red de interconexión, nuestros protocolos garantizan la correcta ejecución de los programas paralelos. El efecto de los fallos, en lugar de un bloqueo del sistema, será una cierta degradación de rendimiento. En la figura 0.3 se muestra la magnitud relativa del incremento en el tiempo de ejecución. Como se observa, incluso para las mayores tasas de fallo que hemos probado, la degradación media del rendimiento varía entre menos del 10 % y el 25 %, dependiendo del protocolo.

Las tasas de fallos que hemos probado son mucho mayores que las tasas de fallos transitorios que cabe esperar en sistemas reales [23]. Por tanto, cabe esperar que en condiciones normales la degradación del rendimiento no sería significativa.

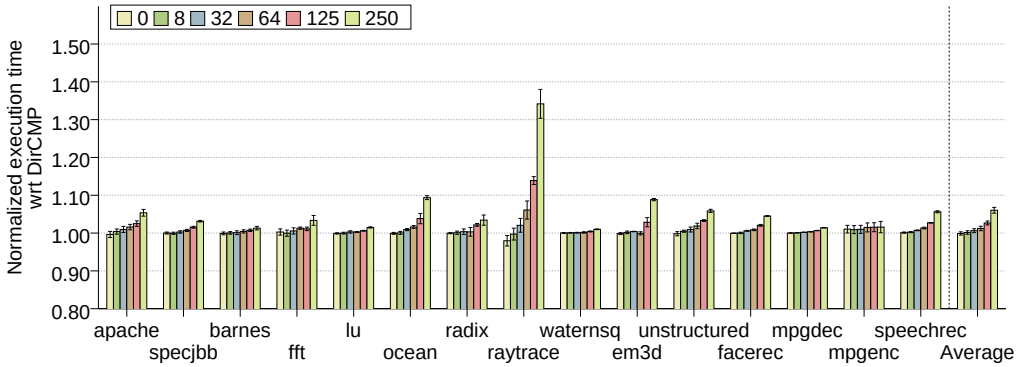
Por último, los resultados mostrados en esta sección dependen del valor de los parámetros de configuración de los protocolos tolerantes a fallos, mostrados en la tabla 0.4b. Estos valores pueden ajustarse para soportar mayores tasas de fallo o reducir la degradación de prestaciones, a costa de una mayor sobrecarga de hardware.

Conclusiones y vías futuras

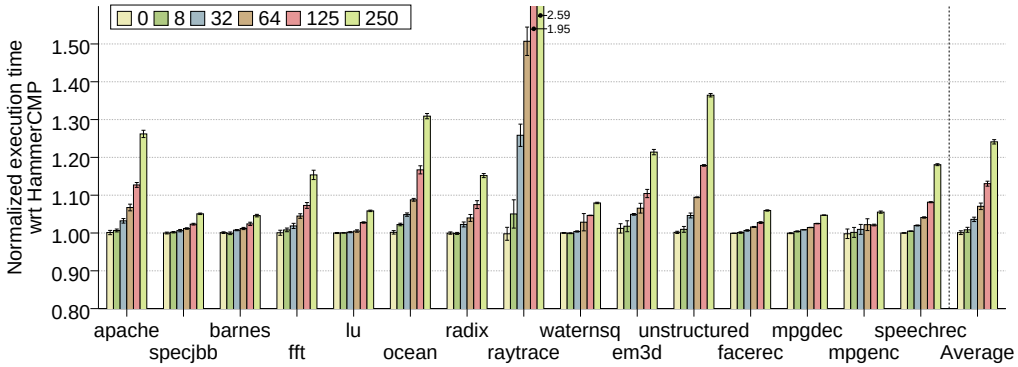
Los fallos transitorios serán cada vez más frecuentes según aumente la escala de integración de los circuitos VLSI. En esta tesis, hemos propuesto una nueva forma de evitar los problemas causados por dichos fallos en la red de interconexión de CMPs, que es uno de los componentes propensos a sufrir fallos transitorios. En lugar de construir una red de interconexión tolerante a fallos, proponemos usar



(a) TokenCMP y FtTokenCMP



(b) DirCMP y FtDirCMP



(c) HammerCMP y FtHammerCMP

Figura 0.3: Degradación del rendimiento bajo diferentes tasas de fallos (en mensajes corrompidos por millón de mensajes que viajan por la red) para cada protocolo tolerante a fallos respecto a su correspondiente protocolo base no tolerantes a fallos

redes de interconexión no fiables y añadir medidas de tolerancia a fallos a los protocolos de coherencia de caché que utilizan las redes de interconexión.

Para mostrar la viabilidad de este enfoque, se han diseñado tres protocolos de coherencia de caché tolerantes a fallos (FtTokenCMP, FtDirCMP y FtHammerCMP) basados en diferentes tipos de protocolos de coherencia de caché no tolerantes a fallos. Hemos mostrado que las medidas de tolerancia a fallos necesarias para garantizar la correcta ejecución de aplicaciones paralelas ante fallos transitorios en la red de interconexión suponen un coste asumible de hardware y no incrementan el tiempo de ejecución de los programas en ausencia de fallos. El mayor coste de nuestras propuestas es un incremento moderado en el tráfico de la red de interconexión. En caso de fallos, la degradación media de rendimiento es pequeña incluso para tasas de fallo más altas de lo que cabe esperar en realidad.

De esta forma, nuestras propuestas son una solución con muy baja sobrecarga para los fallos transitorios en las redes de interconexión de CMPs que pueden ser combinadas con otras técnicas de tolerancia a fallos para construir CMPs fiables.

Los resultados presentados en esta tesis abren nuevas vías de investigación:

- Explorar la aplicación de medidas de tolerancia a fallos a nuevas propuestas de coherencia de caché prometedoras [71,107]. Se espera que, en la mayoría de los casos, las técnicas aquí descritas necesitarán pequeñas modificaciones.

- Diseño de un protocolo de coherencia de caché con soporte para memoria transaccional [49,83]. Pensamos que los mecanismos de necesarios para soportar la memoria transaccional podrían aprovecharse para mejorar la tolerancia a fallos.

- Pensamos que, en lugar de diseñar protocolos de coherencia tolerantes a fallos basados en protocolos ya existentes no tolerantes a fallos, sería conveniente diseñar un protocolo tolerante a fallos desde el principio, usando los mecanismos de tolerancia a fallos para simplificar otras partes del diseño.

- El protocolo de coherencia también podría ayudar a tratar los fallos intermitentes y permanentes. Los mismos mecanismos de detección de fallos transitorios se podrían adaptar para detectar otros tipos de fallos e iniciar una reconfiguración de la red de interconexión.

- Finalmente, ya que la principal característica de nuestros protocolos es que son capaces de garantizar la correcta ejecución de los programas paralelos aunque la red de interconexión no entregue todos los mensajes, se puede

utilizar esta propiedad para simplificar el diseño de la red de interconexión, más allá de la tolerancia a fallos. De esta forma, la red de interconexión podría descartar algunos mensajes cuando fuera necesario. Por ejemplo, esto permitiría mecanismos sencillos de recuperación en caso de interbloqueos o el uso de enrutamiento sin buffers. [47, 48, 86].

Chapter 1

Introduction

Improvements in semiconductor technology have provided an ever increasing number of transistors available per chip. Popularly referred to as "Moore's Law", the widely known prediction made by Gordon E. Moore in 1965 [82] is still regarded as true. It stated that the number of transistors per integrated circuit doubled every two years approximately. Nowadays, billions of transistors are available in a single chip [74].

Computer architects are responsible for deciding how to effectively organize these transistors. Usually, given a number of transistors and a number of other constraints such as target power consumption, reliability of the components and time, we try to design computers with the best possible performance, defined as the speed at which programs execute. Many techniques have been invented since the first microprocessor was designed to reduce execution time.

However, performance cannot be defined only with respect to the execution time. Other factors need to be also considered, such as how hard the microprocessor will be to program, or how much power it will need (and hence, how long its batteries will last in a mobile environment). The relative importance of each of the different facets of performance depend on the specific goals and constraints of the project at hand.

In particular, system reliability needs to be considered. In many applications high availability and reliability are critical requirements, but even for commodity systems reliability needs to be above a certain level for the system to be useful for something. Technology trends are making that level harder to achieve without applying fault tolerance techniques at several levels in the design.

1.1 Tiled Chip Multiprocessors

Traditionally, computer architects have taken advantage of the increment in number of transistors to build increasingly complex systems. Performance of single chip systems has been improved each generation in two ways: increasing the frequency of the processor and exploiting instruction level parallelism (ILP). Both approaches have proved useful in the past, but they are no longer profitable due to physical limits that make increasing the frequency impractical and the excessive complexity and reduced benefits of exploiting ILP beyond a certain point.

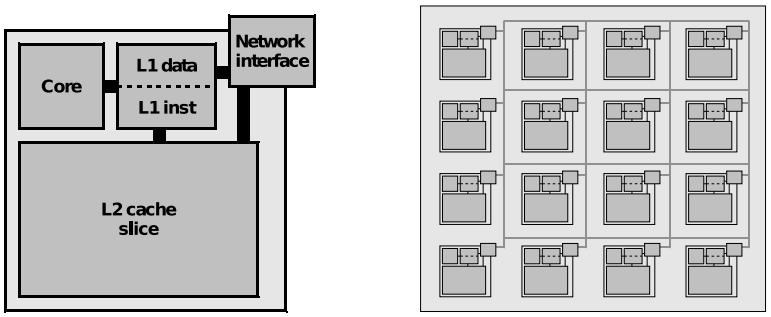
To date, the best way that has been proposed to use effectively the large number of transistors available in a single chip is to build several processors on the same chip instead of a more complex single processor. Compared to other options, Chip Multiprocessors (CMPs) [11,50] offer a way to utilize these resources to increase performance in an energy-efficient way while keeping complexity manageable.

For execution time performance, CMPs exploit thread level parallelism (TLP) in addition to ILP (and sometimes in detriment of exploiting ILP), which means that the execution time of single sequential programs cannot directly benefit from this approach. On the other hand, like most popular multiprocessor systems, CMPs provide the traditional shared-memory programming model which is already familiar to many programmers.

To further reduce complexity, tiled CMPs [127,128,14] have been proposed. Tiled architectures are built by replicating several identical *tiles*. Usually, each tile includes a processor core, private cache, part of a shared cache and an interconnection network interface. Tiles are connected among them via a point-to-point interconnection network. Figure 1.1 shows a diagram of these architectures. Recent proposals include more than one processor core per tile connected amongst them with a crossbar and sharing part of the tile resources. Although we assume a single core per tile, the conclusions of this thesis are valid if more cores per tile are used.

In addition to making complexity more manageable by regularizing the design, tiled architectures 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.

We consider tiled CMPs as the base architecture for explaining the ideas in this thesis. However, although tiled CMPs are the motivating architecture and they are assumed for evaluation purposes, the main ideas of this work can



(a) Diagram of an individual tile

(b) 16 way tiled CMP with 2D mesh interconnection network

Figure 1.1: Tiled Chip Multiprocessors

be also applied to any kind of cache coherent CMP, or even to multiple chip multiprocessor systems.

1.2 Cache coherence and interconnection networks

In shared-memory systems, the communication between different processors (either in different chips or in the same chip) is done implicitly from the point of view of the programmer through a shared physical address space. All processors can read and write to the same shared memory pool and all processors will eventually see changes done by other processors, according to the particular memory consistency model [1] of the architecture.

This programing model is easy to understand by programmers and is flexible enough to efficiently implement other programming models on top of it, like message passing programming. We think that it is safe to assume that it will remain the dominant programming model for a long time. Even if better programming models emerge in the future, shared memory will still need to be efficiently supported if only due to the sheer quantity of software already developed using it.

Unfortunately, the implementation of the shared memory programming model is complicated by the existence of private caches, which are essential to achieve good performance. Private caches capture the great majority of data accesses

reducing the average latency of memory instructions and reducing the number of messages that need to travel through the network.

Caches are critical in CMPs due to the high latency required to access to off-chip memory and the limited bandwidth of off-chip pins. Off-chip memory access latency is due to the growing disparity in speed between processors and main memory.

Since processors copy data to their cache before reading or modifying it, there will be several copies of those memory lines which are being used by several processors at the same time. If any processor modifies its copy of the data while it is stored in its private cache, the other processors will not notice it. Hence, different processors will have different ideas about the content of that memory line, creating an incoherency.

To restore the convenience of the shared memory programming model in presence of private caches, cache coherence is automatically enforced in hardware by most shared memory systems. The alternative is to force the programmer to manually manage the communication between processors by means of carefully flushing caches as required. This manual cache management is very hard to do correctly without severely degrading performance.

Cache coherence ensures that writes to shared memory are eventually made visible to all processors and that writes to the same location appear to be seen in the same order by all processors [41] even in presence of caches. Cache coherence is enforced by means of cache coherence protocols [27] which arbitrate the exchange of data and access permissions between processors, caches and memories.

In the rest of this work, we will call *coherence node* or simply *node* to every device connected to the on-chip network that can take part in a coherence transaction. In practice, a coherence node in a CMP can be either a L1 cache, an L2 cache bank or a memory controller/directory.

There are a number of coherence protocols available to enforce coherence. Deciding the best coherence protocol for a system depends on the characteristics of the system. There are three main approaches to designing a cache coherence protocol:

Snooping-based protocols: All requests are broadcast to all coherence nodes using an interconnect which guarantees total order[1] of messages, like a

[1]Total order of messages means that messages are received by all destinations in the same order that they are sent, keeping the relative order even for messages which are sent to different destinations or from different sources.

shared bus. The total order property of the interconnect is used to serialize potentially conflicting requests, since all nodes "snoop" the bus and see all the requests in the same order.

Directory-based protocols: Requests are sent to a single node (which may be different for different addresses) which forwards it to all the nodes that need to be involved because they have data and they need to invalidate it or send it to the requestor. That single node has the information about the current sharers of the data in a "directory" and serves as the serialization point for requests to the same address. Directory-based protocols do not need a totally ordered network and are appropriate for point-to-point interconnects[2] but the extra indirection introduced by the directory increases the latency of misses.

Token-based protocols: Token coherence [67] provides a way to avoid the indirection introduced by directory-based protocols while using an interconnect which does not provide total order of requests. Most requests in a token coherence protocol are not serialized by any means. Instead, coherence is ensured by a set of "token counting" rules. Token counting rules are enough to ensure that coherence is kept but they are not enough to ensure that requests are actually satisfied. For ensuring this, token-based cache coherence protocols need to provide additional mechanisms that ensure forward progress even in presence of request races. Currently proposed token-based protocols ensure forward progress by means of *persistent requests* when races are detected. These persistent requests are serialized by a centralized or distributed arbiter.

Some coherence protocols impose additional requirements to the interconnection network that they use, like cache coherence protocols that are designed for point-to-point interconnection networks with ring topology [71].

Of all the above, only directory-based and token-based protocols are suitable for tiled CMPs in general, since they can be used along with a point-to-point interconnection network. Hence, this thesis presents fault tolerance measures for token-based and directory-based cache coherence protocols.

[2] Although point-to-point interconnects do not guarantee total order of messages, they may guarantee point-to-point order of messages. That is, two messages are delivered in the same relative order as they are sent as long as both have the same source and destination. Many point-to-point interconnects provide point-to-point ordering guarantees, but not all.

Since cache coherence protocols ensure coherence by exchanging messages between coherence nodes through an interconnection network, their performance (and, in the end, the performance of the whole system) depends on the performance of the interconnection network. As the number of cores in a CMP increases, the importance of efficient communication among them increases too. Hence, the design of the on-chip interconnection network becomes very important in a many-core CMP. For these reasons, it is desirable to give the interconnection network designer as much freedom as possible to concentrate on the performance of the network for the common case, not having to worry too much about other requirements like message delivery order or fault tolerance.

Due to the nature of cache coherence protocols, communication between processors in a CMP is very fine-grained (at the level of cache lines), hence small and frequent messages are used. In order to achieve the best possible performance it is necessary to use low-latency interconnections and avoid acknowledgment messages and other control-flow messages as much as possible.

1.3 Reliability of future electronic devices

The reliability of electronic components is never perfect. Electronic components are subject to several types of *faults* due to a number of sources like defects, imperfections or interactions with the environment. Fortunately, many faults have no user visible effects either because they do not affect the output of the computation in any significant way or because they are detected and corrected by some fault tolerance mechanism. When a fault has user visible effects, it produces an *error*. In fact, fault tolerance mission is avoiding errors caused by faults.

A common classification of faults attends to their duration and repeatability. Faults can be either permanent, intermittent or transient. Permanent faults require the replacement of the component to restore the functionality of the system and, in the context of semiconductor devices, are caused by electromigration among other causes. Intermittent faults appear and disappear, are mainly due to voltage peaks or falls, and partial wearout; and are often forewarns of future permanent faults. Transient faults occur when a component produces an erroneous output but it continues working correctly after the event.

The causes of transient faults are multiple and varied. They include transistor variability, thermal cycling due to temperature fluctuations, current noise, electromagnetic interference (EMI) and radiation from lightning.

Radiation induced transient faults, also known as soft errors [87,89] or single

event upsets, are an increasingly common type of transient fault. The most important sources of soft errors in current semiconductor devices are radiation-induced faults which are caused by alpha particles from radioactive atoms which exist in trace amounts in all packaging materials, and by neutrons present in the atmosphere due to cosmic rays. Neutron flux varies with altitude. It increases with altitude until 15 km (Pfotzer point) and then decreases due to the rareness of the atmosphere. For example, a chip in a plane traveling at 10 km of altitude would have 100 times more chances of suffering a soft error due to a neutron impact than at sea level.

Transient faults are much more common than permanent faults [119]. Currently, transient fault rates are already significant for some devices like caches and memories, where error correction codes are routinely used to deal with them. However, the same technology trends of increased scale of integration which make many-core CMPs possible will make transient faults more common. Also, the lower voltages used for power-efficiency reasons make transient faults even more frequent. Hence, the importance of transient faults will increase for all semiconductor devices [66, 13, 19], and architects need to assume a certain rate of transient faults for every design and incorporate the adequate fault tolerance measures to meet the reliability requirements of the final device.

Moreover, 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 need to target the common case and assume a certain rate of transient faults. Hence, transient faults will affect more components and more frequently and will need to be handled across all the levels of the system to avoid actual errors.

For every system, there is a minimum reliability target that has to be met to make the system usable. Hence, to enable more useful chip multiprocessors to be designed, several fault-tolerant techniques must be employed in their construction.

1.4 Motivation

One of the components in a CMP which will be affected by the transient faults described above is the on-chip interconnection network, which is used to exchange coherence messages between processors, caches and memory controllers.

The interconnection network occupies a significant part of the chip real estate, so the probability of a particle hitting it and causing a transient fault is high. Also,

since it is built with longer wires on average than the rest of the components of the chip, it is more prone to coupling (crosstalk) which increases the chances of transient faults [34].

At the same time, the performance of the interconnection network is critical to the performance of the whole system. It handles the communication between the cores and caches, which is done by means of a cache coherence protocol. This means that communication is usually very fine-grained (at the level of cache lines) and requires very small and frequent messages. Hence, to achieve good performance the interconnection network must provide very low latency and should avoid acknowledgment messages and other flow-control messages as much as possible.

Transient faults in the interconnection network will cause corruption of messages of the cache coherence protocol. If these faults are not detected and corrected, they will lead to deadlocks or silent data corruption, either because the fault corrupts a data carrying message or because coherence cannot be correctly maintained.

One way to deal with transient faults in the interconnection network is to build a fault-tolerant interconnection network, that is: an interconnection network that guarantees correct delivery of every message despite any transient fault. Several proposals on how to do this are mentioned in chapter 2.

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 approach has three main advantages:

- The cache coherence protocol is in a better position to know which messages are critical and which ones can be lost with benign consequences. Hence, it can make smarter choices to achieve fault tolerance with less overhead, both in term of execution time and network overhead. On the other hand, fault tolerance measures at the interconnection network level have to treat all messages as critical, imposing an overhead roughly proportional to the number of messages.

- Implementing fault tolerance measures at the interconnection network level limits the design flexibility for the architects that have to design the network itself. Since the performance of the interconnection network is key to the performance of the whole chip it is desirable to aggressively optimize the interconnection network to reduce latency in the common case and increase bandwidth. Fault tolerance measures make aggressive optimizations harder to implement.

- As we will show, most of the time a fault-tolerant cache coherence protocol behaves almost like a non fault-tolerant one, hence the effect of the fault tolerance measures in performance when no faults occur is very low. The fault-tolerant protocol behaves differently only when critical messages need to be transferred and when a fault is detected and needs to be recovered. On the other hand, fault tolerance measures at the interconnection network level affect the behavior of the interconnection network all the time whether faults actually occur or not.

Of course, handling interconnection network fault tolerance at the level of the cache coherence protocol has its drawbacks too. The main disadvantage is the need to modify the coherence protocol itself, whose performance is also key to the performance of the whole system and which is often regarded as a complex piece of the design. In this thesis we demonstrate that the added complexity is assumable and the performance overhead is minimal.

1.5 Contributions

In this work, we propose a way to deal with the transient faults that occur in the interconnection network of CMPs. In our failure model we assume that, from the point of view of the cache coherence protocol, the interconnection network will either deliver a message correctly to its correct destination or not deliver it at all. This can be easily achieved by means of using an error detection code (EDC) in each message and discarding corrupted messages upon arrival. Unexpected messages are assumed to be misrouted and are also discarded. We also assume that caches and memories are protected by means of ECC or some other mechanism, so that data stored there is safe.

In this thesis, we assume that messages are discarded due to being corrupted. However, the techniques described here are also applicable if messages are discarded for other reasons. For example, messages could be discarded due to lack of buffering space in switches. This property may be useful for network designers, but we have not evaluated it.

For simplicity, we only consider traffic due to accesses to coherent memory and ignore for now accesses to non-coherent memory like memory-mapped I/O. We expect that enabling fault tolerance for those messages would be easy to do following the same ideas discussed here.

We can assume that these faults cause the loss of some cache coherence messages, because either the interconnection network loses them, or the messages

reach the destination node (or other node) corrupted. Messages corrupted by a soft error will be discarded upon reception using error detection codes. Our proposal adds only those acknowledgments which are absolutely required and does so without affecting the critical path of most operations.

We attack this problem at the cache coherence protocol level. In particular, we assume that the interconnection network is no longer reliable and extend the cache coherence protocol to guarantee correct execution in presence of dropped messages. That is, while traditional cache coherence protocols assume that messages sent through the network will eventually arrive to their correct destination, ours just assume that messages will arrive correctly most times, but sometimes they will not.

Our proposals only modify the coherence protocol and do not add any requirement to the interconnection network, so they are applicable to current and future designs. We protect dirty data with acknowledgment messages out of the critical path of cache misses and provide mechanisms for recovering from lost data and control messages.

Since the coherence protocol is critical for good performance and correct execution of any workload in a CMP, it is important to have a fast and reliable protocol. Our protocols do not add a significant execution time overhead when compared to similar non fault-tolerant protocols but add a small amount of extra network traffic.

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

In summary, the main contributions of this thesis are:

- We have identified the different problems that the use of an unreliable interconnect poses to cache coherent CMPs.
- We have proposed modifications to the most important types of protocols which are likely to be used in tiled CMPs to cope with messages dropped by the interconnection network without adding excessive overhead.
- A cache coherence protocol which extends a token-based cache coherence protocol with fault tolerance measures; and modifications to the token

coherence framework to ensure reliable transference of ownership and owned data.

- A cache coherence protocol which extends a standard directory-based coherence protocol with fault tolerance measures.
- A cache coherence protocol which extends a broadcast-based snoopy protocol similar to the protocol used by AMD Hammer processors with fault tolerance measures.
- We have evaluated our proposals using full-system simulation to measure their effectiveness and overhead and we have found that although the cache coherence protocol is critical to the performance of parallel applications, the fault tolerance measures introduced in our protocols add minimal overhead in terms of execution time. The main cost of our proposals is a slight increase in network traffic due to some extra acknowledgments.

Our proposals do not add any requirement to the interconnection network so they are applicable to current and future designs. Actually, since we remove the requirement that the network need to guarantee correct delivery of every message, we expect that network designers can propose more aggressive interconnect designs to reduce the average latency even at the cost of dropping a few messages, improving overall performance. In this thesis we concentrate on messages lost due to transient failures and leave the latter as future work.

Parts of this thesis have been published as articles in international peer reviewed conferences [36,39,38] or peer reviewed journals [37].

1.6 Organization of this thesis

The rest of this document is organised as follows. Chapter 2 makes a survey of previous work on fault tolerance related with this thesis. In chapter 3 we present the base non fault-tolerant architecture that we want to protect against transient faults in its interconnection network and identify the requirements for doing so at the level of the cache coherence protocol. Chapter 4 presents our first fault-tolerant cache coherence protocol, that is based on token coherence. Then, chapter 5 presents another fault-tolerant protocol, this time based on directory coherence, while chapter 6 presents our last fault-tolerant protocol based on a broadcasts-based cache coherence protocol inspired by the one used by AMD Hammer processors. In chapter 7 we present our experimental methodology,

including details about the simulator that we have used, the applications that we use as benchmarks and the configurations that we have employed. We evaluate our protocols in chapter 8, both from the point of view of the overhead introduced by the fault tolerance measures when no faults occur and from the point of view of the performance degradation that happens with a number of fault rates. Finally, in chapter 9 we present the conclusions of this thesis and future lines of work.

Chapter 2

Related work

In this chapter we show a summary of previous work related with this thesis. Fault tolerance is a vast area, hence we cannot cover all the aspects required to build reliable CMPs. Instead, we mention the most relevant works related to fault tolerance in tightly-coupled symmetric multiprocessors (SMP) and CMPs when regarded as a distributed system of processor cores which communicate with each other through a shared address space. We do not talk extensively about how to build reliable processor cores, nor other components of a CMP except the interconnection network.

2.1 Fault-tolerant multiprocessors

Fault tolerance for multiprocessors has been thoroughly studied in the past, specially in the context of multiple chip multiprocessors.

Usually, fault recovery mechanisms are classified as either Forward Error Recovery (FER) or Backward Error Recovery (BER). BER is more commonly known as *rollback recovery* or *checkpointing*.

FER implies adding hardware redundancy to the system is such a way that the correct result of a computation can be determined even if some of the components fail. In some cases, cost-effective FER strategies can be designed for individual devices (like a processor, for example). However, generic FER cost is too high. The most common FER strategy are Error Correcting Codes (ECC) and Triple Modular Redundancy (TMR) [99].

ECC adds redundancy to information which is being transmitted or stored

to be able to recover the original data if some bits of the transmitted or stored information change their values. ECC is extensively used in memories and caches to deal with transient faults.

In TMR each operation is performed by three components and the correct result is determined using majority vote. The area overhead of TMR is usually too high except for some critical components or in specific situations.

Dual modular redundancy (DMR) is a less costly scheme in which only two components perform each operation. In case of differing outputs, some recovery mechanism is invoked. The recovery mechanism can be a rollback to a previous checkpoint or, if the faulty component can be identified, the system may switch to a degraded mode without redundancy.

Several high-end commercial systems have been built that provide fault tolerance through system-level or coarse-grain redundancy and targeting high-availability needs [12], like systems offered by Stratus [135], Sequoia [16], Tandem (now HP) NonStop systems [15] or IBM zSeries [112,119]. These systems often employ high levels of redundancy end even lockstep execution, making them too expensive for general use.

In checkpointing, the state of a computation is periodically saved to stable storage (checkpointed) and used to restart the computation in case of a failure. The checkpoint may be saved incrementally (storing only differences with respect to a previous checkpoint). BER does not require hardware redundancy per se, but it requires additional resources and it impacts system performance even in the common case of fault-free execution due to the need to periodically create checkpoints.

Checkpointing implemented in software is the most common approach to building fault-tolerant cache coherent systems using common server hardware. These approaches require modifications to either the operating system [57,60,109], virtual machine hypervisor [22], or user applications [21,64].

The memory hierarchy is often utilized for supporting checkpointing in hardware, making it transparent (or almost transparent) to the operating system and user programs, and incurring less performance overhead. Since the checkpointing overhead is lower, checkpoints can be made much more often and this reduces the mean time to recover when a fault is detected (MTTR). Hunt and Marinos [52] proposed *Cache Aided Rollback Error Recovery* (CARER), a checkpointing scheme for uniprocessors in which checkpoints are updated every time that a write-back happens.

In the case of a multiprocessor, implementing checkpointing efficiently is significantly more complicated because it requires coordination among all the

processors to create a system-wide checkpoint or a set of consistent processor-local checkpoints. Creating a system wide checkpoint [73,84,85,97] is the simplest approach, but it has the highest overhead.

Many proposals avoid global checkpoints and propose synchronized local checkpoints. Ahmed *et al.* [6] extend CARER for multiprocessors. In their scheme, all processors synchronize whenever any of them need to take a checkpoint. Wu *et al.* [138] also extend CARER, and perform a global checkpoint when the ownership of any modified memory line changes. Banâtre *et al.* [10] proposed *Recoverable Shared Memory* (RSM) which deals with processor failures on shared-memory multiprocessors using standard caches and standard snoopy cache coherence protocols. In RSM, when a processor needs to make a new checkpoint, only those processors which have communicated with it since the last checkpoint need to synchronize and make a new checkpoint too.

Differently from other authors, Sunada *et al.* [123] propose *Distributed Recoverable Shared Memory with Logs* (DRSM-L) that performs unsynchronized checkpoints. In DRSM-L, processors establish local checkpoints without communication with other processors and log the interactions with other processors. In case of rollback, the logged interactions are used to determine which processors need to rollback.

More recently, Pruvlovic *et al.* [100] 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. At the same time, Sorin *et al.* [117] presented *SafetyNet* which aims at similar objectives but has less overhead, although it requires custom caches and can only recover from transient faults. In particular, *SafetyNet* is shown to be able to recover from transient faults that affect the interconnection network of a multiprocessor system.

Most checkpointing proposals deal only with the memory state contents of the system and do not address how to checkpoint and rollback in presence of I/O, which is a requirement for any realistic system. The common approach to I/O is to log the input for replaying in case of rollback and delay the actual output until a new checkpoint is performed, so that committed output never needs to be undone or repeated [73]. Nakano *et al.* [91] propose an efficient scheme to be able to transparently buffer I/O operations that builds on *ReVive*.

Gold *et al.* [43] propose the *TRUSS* architecture. In *TRUSS*, each component of the multiprocessor server (processor cores, memory or I/O subsystems) must detect and recover from errors without involvement of the rest of components.

Fault containment between components is achieved by the *Membrane* abstraction. *TRUSS* requires a reliable interconnection network.

A multiprocessor can suffer faults in the interconnection network, cache coherence controllers or other components that would lead to errors in the execution of a parallel workload that expects a particular consistency model and cache coherence behavior. Sorin *et al.* [116] dynamically check end-to-end invariants of the cache coherence protocol and the interconnection network applicable to snooping coherence protocols implemented on top of a point-to-point interconnection network. Meixner and Sorin proposed error detection techniques [77, 78, 79] applicable to any cache coherence protocol which can detect faults that lead to incorrect ordering of memory accesses (both consistency or coherence violations) assuming sequential consistency, including transient errors in the interconnection network. However, they do not provide any recovery mechanism. The work of Meixner and Sorin is based on checking conservative system invariants and may report false positives. Chen *et al.* [25] propose a different approach based on constraint graph checking [24] which is more precise.

2.2 Fault tolerance in CMPs

Many of the techniques used for building fault-tolerant general multiprocessor systems can be easily adapted for building fault-tolerant chip multiprocessors (CMPs). However, from the point of view of fault tolerance, CMPs present new problems like the increased complexity in achieving isolation, and new opportunities like the greater availability of on-chip resources for redundancy.

At the core processor level, a CMP can use the same techniques used by commercial high availability processors [112], processor level error checking or avoidance techniques [8, 104, 115, 132, 101], redundant process execution implemented using SMT [108, 105, 131, 115] or separate cores [124, 88, 44, 114, 113, 59, 103], or redundancy in the core internal structures [80, 111, 20, 121, 18, 81, 75], both adding redundancy for the explicit purpose of fault tolerance and leveraging the inherent redundancy present in superscalar out-of-order processors. However, since CMP designs tend toward simpler cores than previous superscalar processors, there is less inherent redundancy inside the core and hence the relative cost of previously proposed fault tolerance measures is higher. In [76], Meixner *et al.* propose *Argus* as a high-level error detection method applicable to any processor core and suitable for simple in-order cores.

Sylvester *et al.* [126] propose *ElastIC* as an architecture that continuously

monitors the performance, power and reliability of each core and adaptively throttles or disables failing cores.

Aggarwal *et al.* [4] analyze five commodity CMP architectures from the point of view of their reliability. They find that transient fault detection is limited to storage arrays (using ECC or parity checking and retries in caches, register files and memories) and that the major shortcoming of these architectures is the lack of proper fault isolation. The lack of isolation reduces the effectiveness of fault tolerance techniques like running programs in DMR or TMR configurations. In [3], they propose *configurable isolation*, a mechanism to provide dynamic reconfiguration of CMPs which enables fault containment to deal with errors, and show how to use that support to implement a DMR configuration similar to a NonStop system [15].

Gupta *et al.* [45] propose *StageNet*, another reconfigurable CMP architecture. However, in contrast with previous authors who proposed tile-level or core-level reconfigurability, *StageNet* proposes reconfigurability at a much finer granularity of pipeline stages connected through a reconfigurable interconnection network. *StageNet* tries to minimize the redundancy added exclusively for fault tolerance purposes and could be used in single cores as well [46]. Romanescu and Sorin [106] propose the *Core Cannibalization Architecture* (CCA), which also allows reconfigurability at the pipeline stages but has a much less radical design. In CCA, only a subset of the pipeline stages can be reconfigured, and has a set of *cannibalizable* cores which lend their functional units when an error is detected in other cores.

In [133] Wells *et al.* argue that using an overcommitted system with a thin firmware layer (an hypervisor) is more effective to adapt to intermittent faults than pausing the execution of the thread scheduled in the affected core, using spare cores or asking the operating system to temporarily avoid using the affected core. In [134], they study the implications of running several applications in a CMP with reliability support where some applications require redundancy and use *Reunion* [113] for DMR execution alongside with applications that do not require redundancy, and also propose using an hypervisor to abstract the management of reliability features from system software.

2.3 Fault tolerance at the interconnection network level

The main proposal of this thesis is dealing with transient faults in the interconnection network of CMPs at the level of the cache coherence protocol that uses such interconnections networks. On the other hand, a more traditional and arguably more straightforward way to achieve protection against transient faults in the on-chip interconnection network is making the network itself fault-tolerant. There are several proposals exploring reliable networks on chip (NoC).

Soft faults that affect an interconnection network can be grouped in two main categories: link errors that occur during the traversal of flits from router to router, and errors that occur within the router.

Link errors can happen due to coupling noise and transient faults [120]. They are usually considered the predominant source of transmission errors in interconnection networks and error correcting codes (ECC) or error detecting codes (EDC) plus retransmission are being used extensively in on-chip communication links as a form of protection against link errors [141, 129]. Bertozzi *et al.* [17] report that retransmission strategies are more effective than correction ones, although the latency penalty of retransmission schemes is higher. Error correction can be done either at message level or flit level.

ECC can be also complemented with retransmission [90], since many coding techniques can detect errors in a greater number of bits than those that they can correct, like the popular Single Error Correction and Double Error Correction codes (SECDED). In this case, faults that affect only one bit are corrected and retransmission is requested only in case of faults that affect two bits.

Retransmission can be either end-to-end or switch-to-switch (also known as hop-to-hop). Both schemes require adding EDC to messages (or flits) and require dedicated buffers to hold the messages (or flits) until the receiver or the next switch acknowledges their reception. The acknowledgment can be done using a new message, a dedicated line or piggybacked with other messages. In some cases, only negative acknowledgments (NACK) are necessary [94]. 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.

In an end-to-end scheme, retransmission buffers are added to the network interfaces and error checking and acknowledgment is done by the receiver node. On the other hand, in a switch-to-switch scheme buffering needs to be added

to each switch and error checking and acknowledgment is done by each switch. Hence, upon a fault the latency to detect it and retransmit the message is higher in the case of end-to-end schemes.

For example, [30] and [29] describe simple switch-to-switch retransmission schemes. The Unique Token Protocol is used in [30] to achieve link-level retransmission without requiring acknowledgment packets. Ali *et al.* [7] propose a fault-tolerant mechanism to deal with packet corruption due to transient faults using error detection codes and end-to-end retransmission and a dynamic routing mechanism to deal with permanent link or router failures.

Murali *et al.* [90] present a general design methodology and compare several error recovery schemes in an interconnection network with static routing and wormhole flow control. Lehtonen *et al.* [61] propose a link-level transient fault detection scheme and interleaving.

In addition to link errors, transient faults can also affect the router logic. The effect of these errors include data corruption which can be recovered using ECC as in the case of link errors. However, soft errors in the router logic can also cause other problems like misrouted packets, deadlocks or packet duplication. A fault-tolerant router can deal with link-level faults, faults in its internal logic, or both.

Kim *et al.* [55] propose a router architecture that can tolerate transient faults within the router's individual components. Park *et al.* [94] generalize the previous work and provide solutions relevant to generic router architectures. Constantinides *et al.* [26] introduce an on-chip router capable of tolerating silicon defects, transient faults and transistor wearout using domain specific fault tolerance techniques. Pereira *et al.* [40] propose fault tolerance techniques to protect NoC routers against the occurrence of soft errors and crosstalk.

Another way to provide fault tolerance at the interconnection network level is using fault-tolerant routing techniques. These methods rely on probabilistic flood (gossip algorithms) [33,95] or redundant random walks [96]. These techniques trade increased network traffic and power consumption for reliability. Their main drawbacks are that they are not deterministic and the high network traffic overhead.

Other techniques try to mitigate the probability of transient faults. Since the low voltages that are required for low-power devices reduce the noise margins and increase the probability of transient faults, it is necessary to seek a trade-off between power consumption and reliability [51,139,125]. Worm *et al.* [137] propose dynamically varying the supply voltage according to the error rate on the links in a low-power interconnect system which also includes error detection

codes and retransmission. Li *et al.* [62] propose to dynamically choose the most energy efficient error protection scheme that maintains the error rate below a preset threshold. Ejlali and Al-Hashimi [35] propose the use of energy recovery techniques to construct low-power and reliable on-chip interconnects.

Capacitive crosstalk on global system-on-chip buses increases power consumption, propagation delay and the probability of transient errors due to noise [110,51], specially when low power codes (LPC) [122,102] are used. Several coding schemes have been proposed that mitigate these effects [31,56,130,140,118] avoiding adversarial switching patterns to decrease crosstalk between wires.

Level-encoded Dual-rail has been the dominant encoding scheme for interconnection networks that connect different clock domains. Since dual-rail encoding is vulnerable to single-error upsets, phase-encoding communication schemes have been proposed [28,92] which are resilient to transient faults. Also, Halak and Yakovlev propose fault-tolerant techniques to mitigate the effect of crosstalk in phase-encoding communication channels.

In addition to the specific techniques, 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, relaxed scaling rules or other VLSI transient fault mitigation techniques to avoid single points of failure.

Many of the proposals mentioned in this section could be complemented with a fault-tolerant cache coherence protocol to achieve greater protection against transient faults. Since no fault tolerance technique can guarantee full coverage against all faults and since a fault-tolerant interconnection network is bound to be more expensive, slower or more power-hungry than a less reliable one, by using a fault-tolerant cache coherence protocol architects can trade-off some of the fault tolerance of the interconnection network for a reduced cost, better performance or better power efficiency.

In addition, ensuring the reliable transmission of all messages through the network limits the flexibility of the network design. 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 interconnection network which can be not totally reliable, but has better latency and power consumption in the common case. The protocol itself ensures the reliable retransmission of those few messages that carry owned data and could cause data loss; and provides better performance as long as enough messages are transmitted correctly through the network.

Chapter 3

General assumptions and fault tolerance requirements for cache coherence protocols

In this chapter, we describe in more detail the general assumptions that we make about the base architecture and what are the possible errors that can appear due to transient faults in the interconnection network of such architecture.

3.1 Base system assumptions

As mentioned above, we target tiled CMP architectures (see figure 1.1) since we expect that future many-core CMP designs will be based on that philosophy in order to keep complexity manageable.

We assume that the chip is comprised by a number of tiles laid out in a grid. Each tile will include at least the following components:

- A processor core.
- A private L1 cache split in a data cache and a read-only instruction cache. A cache line can be in either of them, but not on both at the same time.
- A slice of a shared L2 cache (both for instructions and data). Hence, the L2 cache is logically shared by the tiles but it is physically distributed among them. Each cache line has a *home tile* that is the tile which includes the

bank of L2 cache where that line can be cached. We assume that the L2 cache is not inclusive with respect to the L1 cache: a memory line may be present in L1, L2 or both, although our proposal is equally applicable to other inclusion policies (exclusive or inclusive).

- A network interface, which connects the tile to the on-chip interconnection network.

The L1 cache and the L2 cache are protected using error correction codes (ECC) or some other technique, as well as off-chip memory. Hence, our cache coherence protocols assume that the data stored in them is safe from corruption. In other words, only the data traveling through the network has to be protected by the fault-tolerant protocols.

The tiles which are part of the chip are connected among them by means of a point-to-point interconnection network. For evaluation purposes, we use a 2D-mesh topology with deterministic dimension-order routing. However, the protocols presented in this thesis do not make any assumptions about the particular characteristics of the network, and can be used with adaptive routing or with other network topologies. That is, no assumption is made about the order in which messages are delivered, even for two messages with the same source and destination.

In the architecture described above, cache coherence is maintained at the level of the L1 caches. That is, the L2 cache could be seen as filling the role of the memory in a cache coherence protocol designed for SMPs, since such protocols keep coherence at the level of the L2 cache.

Several CMPs like the one described above can be connected among them to build a *multiple CMP* (M-CMP). In a M-CMP, coherence has to be kept both at the level of the L1 caches within the chips and at the level of the chips themselves (L2 caches). For this task, one could use a hierarchical cache coherence protocol or a protocol specifically designed for this situation [70]. The protocols presented in this work have not been designed with a M-CMP system in mind, but the modifications required for fault tolerance are mostly orthogonal to the implementation of hierarchical coherence. The protocol presented in chapter 4 is based on a protocol designed for M-CMPs and has been verified to work in that scenario, although we have not performed an exhaustive evaluation.

We have focused on transient failures that occur on the on-chip interconnection network and have not explored failures in the off-chip interconnection network because the first ones are more likely to appear due to current technology trends. However, our techniques can be applied to coherence among several chips too.

Abstracting the interconnection network we can assume that, from the point of view of the coherence protocol, a coherence message either arrives correctly to its destination or it does not arrive at all. In other words, we assume that no incorrectly delivered or corrupted message is ever processed by a coherence controller. To guarantee this, error detection codes (EDC) are used to detect and discard corrupted messages. There are a number of coding schemes which can be used for this purpose [58,89].

Upon arrival, the EDC is checked using specialized hardware and the message is discarded if it is wrong. To avoid any negative impact on performance, the message is assumed to be correct because this is by far the most common case and the EDC check is done in parallel to the initial processing of the message (like accessing to the cache tags and to the MSHR to check the line state).

In summary, from now on we consider tiled CMP systems whose interconnection network is not reliable due to the potential presence of transient faults. We assume that these faults cause the loss of messages (either an isolated message or a burst of them) since they directly disappear from the interconnection network or arrive to their destination corrupted and then they are discarded after checking their error detection code.

Instead of detecting faults and returning to a consistent state previous to the occurrence of the fault using a previously saved checkpoint, 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 rollback due to a fault in the interconnection network.

The most obvious solution to the problems depicted above is to ensure that no message is lost while traveling through the interconnection network by means of reliable end-to-end message delivery using acknowledgment messages and sequence numbers in a similar way to TCP [98], like several of the proposals discussed in section 2. However, this solution has several drawbacks:

- Adding acknowledgments to every message would increase the latency of cache misses, since a cache would not be able to send a message to another cache until it has received the acknowledgment for the previous message.

- That solution would significantly increase network traffic. The number of messages would be at least doubled (one acknowledgment for each message). Since the interconnection network does not have as much insight about the meaning of the messages as the cache coherence protocol, it cannot protect only those messages that are critical for correctness.

- Extra message buffers would be needed to store the messages until an acknowledgment is received in case they need to be resent.
- The flexibility of the network designer would be reduced, since fault tolerance would have to be considered. This reduced flexibility would make harder to design a high performance interconnection network.

We do not try to address the full range of faults that can occur in a CMP system. We only concentrate on those faults that affect directly the interconnection network. Hence, other mechanisms should be used to complement our proposal to achieve full fault tolerance for the whole CMP.

3.2 Requirements for a fault-tolerant cache coherence protocol

Loss of coherence messages can cause a number of problems in a tiled CMP like the one described in section 3.1. Some of the problems depend on the particular cache coherence protocol that is being used. The following list includes all the problems identified in the base non fault-tolerant protocols presented in this thesis:

Silent data corruption: A corrupted data carrying message, if accepted, could cause silent data corruption (SDC) if the header of the message is not affected by the corruption. This can be avoided augmenting the coherence messages with error detection codes or error correction codes.

Undetected memory incoherence: In some protocols, a lost message can lead to cache incoherence. A cache coherence violation occurs when some node has write permission to a cache line while at least another has read permission for the same line. This can happen, for example, in a directory protocol that does not require acknowledgments for invalidation messages [42,53]. In such a protocol, if an invalidation message is lost, a sharer node will continue having read permission after write permission is granted to a different node that has requested it since neither the directory nor the requestor would notice that the invalidation message has been lost.

All the base protocols described in this thesis require acknowledgments for invalidation messages to avoid race conditions. In these protocols, no message loss can lead to cache incoherence.

Deadlock: The most frequent effect of a message loss in most cache coherence protocols is a protocol deadlock. The deadlock can happen in the same transaction that suffered the message loss or in a subsequent coherence transaction. Deadlocks happen easily because when a requestor sends a request message, it will wait until the response arrives. If either the response message, the request message itself or any intermediate message are lost, the requestor will wait indefinitely for the response.

Detected unrecoverable error (deadlock, data loss, or data corruption): Some systems may be able to detect some faults but have no means of recovering from the situation. In that case, a detected fault would be signalled allowing the OS to crash the application or resort to some higher level recovery mechanism. If the OS has no recovery mechanism in place, the final result would be a detected but unrecoverable error (DUE).

DUEs can happen in systems which employ error detection codes to avoid SDC even if their cache coherence protocol is not fault-tolerant. Also, in case of a deadlock, a fault can be detected by means of a watchdog timer even if it is unrecoverable.

Even if the fault is detected, it may not be possible to recover from it without loss of data if it was caused by a discarded message which contained data and there was no other updated copy of that cache line.

Harmless performance degradation: Some protocols include messages which can be discarded without serious consequences. This is the case of hint messages, messages used to update soft-state in protocols that use prediction [107] or transient requests in token based [68] cache coherence protocols. Losing any of these messages will not have any impact in the correctness of the execution of the program, although it could have a very minor effect in the execution time. In fact, is it very unlikely that losing a moderate number of any of these messages could have a measurable effect in any cache coherence protocol.

To avoid the problems mentioned above, a fault-tolerant protocol needs to have the following features to be able to ensure correct operation using a faulty interconnection network:

Avoid data corruption: The system needs to be able to detect corrupted messages and discard them. For this purpose, error detection codes (EDC) need

to be used. Only error detection is necessary, since the fault-tolerant protocol can provide recovery mechanisms that handle the loss of any message. As mentioned previously, some systems use EDC to avoid SDC despite not having a fault-tolerant cache coherence protocol.

Avoid data loss: Since messages can be corrupted and discarded, it is necessary to avoid sending data through the interconnection network without keeping a backup copy of it if there is no other updated copy of the data which can be used for recovery.

Avoid cache incoherence: The fault-tolerant cache coherence protocol needs to ensure that no cache incoherence can happen even if the network fails to deliver a message. In particular, invalidation messages need to be acknowledged (by sending an acknowledgment message either to the directory that sent the invalidation or to the original requestor). Many cache coherence protocols already require these acknowledgments since they are needed to avoid race conditions in unordered interconnection networks.

Detect and recover from deadlocks: When a coherence message is dropped by the interconnection network, it results almost in all cases in a deadlock. Fault-tolerant cache coherence protocols need to be able to detect these deadlocks and be able to recover from them, or avoid them altogether (although this is not practical most times). When a deadlock is detected, some recovery mechanism has to be invoked. Deadlocks are the easier symptom of faults to identify and usually the other problems mentioned above are accompanied by them, hence the recovery mechanism invoked when a deadlock is detected can be used to deal with most other problems if necessary.

To be useful at all, in addition to these properties, a fault-tolerant cache coherence protocol must provide a performance very similar to the performance of a comparable non fault-tolerant cache coherence protocol. The overhead introduced by the fault tolerance measures must be no higher than the overhead of increasing the reliability of the interconnection network used by the protocol.

The fault-tolerant cache coherence protocols described in this thesis implement solutions to the requirements exposed above which are adequate to add fault tolerance to the base protocols without impacting their performance too much.

Chapter 4

A token-based fault-tolerant cache coherence protocol

In this chapter we present our first fault-tolerant cache coherence protocol, which we have called call FtTokenCMP. This protocol is based on the token coherence framework [67] (we assume familiarity with token-coherence in this chapter, see appendix A for a short introduction to token coherence). We based our first fault-tolerant cache coherence protocol on token coherence due to several reasons:

- Token coherence is a framework for designing coherence protocols which provides a separation between the mechanisms used to enforce coherence and the mechanisms used to perform the exchange of data and access permissions between processors. The *correctness substrate* ensures that coherence is not violated by means of token counting rules, while the *performance policy* governs the communication between nodes optimizing the performance of the common case. This separation eases the development of cache coherence protocols since it simplifies the handling of corner cases [68]. In particular, to add fault tolerance measures to a token based cache coherence protocol we only have to modify the correctness substrate while impacting the performance policy as little as possible to reduce the overhead in terms of execution time.

- Token coherence protocols make use of transient requests. These requests are issued by the nodes when they require data or additional access permissions to a cache line, but they can fail sometimes due to race conditions.

The coherence protocol has to provide a mechanism to handle this situation, either reissuing the transient request or issuing a persistent requests (which are guaranteed to success). The provisions for transient requests already provide a basis for minimal support for fault tolerance.

- Most importantly, token coherence protocols already rely on timeouts to detect unsatisfied transient requests. Since we use timeouts for fault detection, this reduces the extra hardware modifications required to implement fault tolerance.

Although we have based our fault-tolerant cache coherence protocol on TokenCMP [70], a token based coherence protocol for Multiple-CMPs, we expect that these ideas will be applicable to any token based protocol with only minor modifications.

4.1 Problems caused by an unreliable interconnection network in TokenCMP

There are several types of coherence messages in a token based cache coherence protocol that can be lost, which translate into a different problem or problems of those mentioned in section 3.2. Table 4.1 shows a summary of the problems caused by the loss of each type of message in TokenCMP.

Table 4.1: Summary of the problems caused by loss of messages in TokenCMP

Fault / Type of lost message	**Effect**
Transient read/write request	Harmless
Response with tokens	Deadlock
Response with tokens and data	Deadlock
Response with the owner token and data	Deadlock and data loss
Persistent read/write requests	Deadlock
Persistent request deactivations	Deadlock

Differently than in a traditional directory-based protocol, in a token-based cache coherence protocol there are no invalidation messages. Their role is filled by transient and persistent requests to write, which in TokenCMP are broadcast to every coherence node. Every coherence node that receives one of these requests

will send every token that it holds (and data if it holds the owner token) to the requestor. Nodes that do not hold tokens will ignore these requests. Since these invalidation messages (which are actually persistent or transient requests) in the base protocol require acknowledgments (the caches holding tokens must respond to the requester), losing a message cannot lead to incoherence, as stated in section 3.2.

Also, losing transient requests is harmless. Even when we state that losing the message is harmless (we mean that no data loss, deadlock, or incorrect execution would be caused), some minor performance degradation may happen. In TokenCMP, if a transient request is not received by any of the nodes that would have had to respond with a token or data, the requester will issue a persistent request after a short timeout expires. Other token based protocols may reissue the transient request one or more times before issuing the persistent request.

However, losing any other type of message will lead to deadlock, and if the owner token was being carried in that message it will cause data loss too.

Particularly, losing coherence messages containing one or more tokens would lead to a deadlock because the total number of tokens in the whole system must remain constant to ensure correctness. Since a coherence node needs to hold all the tokens associated to a particular cache line to be able to write to that line, if a token disappears no node will be able to write anymore to the cache line associated with that token. This will cause a deadlock for this coherence transaction or the next time that any node tries to write to that memory line.

The same thing happens when a message carrying data and one or several tokens is lost, as long as it does not carry the owner token. However, no data loss can happen in this case because there is always a valid copy of the data at the coherence node which has the owner token.

Another different case occurs if the lost coherence message carries the owner token, since it must also carry the memory line and this may be the only currently valid copy of it. In the TokenCMP protocol, like in most cache coherence protocols, the data in memory is not updated on each write, but only when it is evicted from the owner node (that is, the node that currently holds the owner token). Also, the rules governing the owner token ensure that there is always at least a valid copy of the memory line which travels along with it every time that the owner token is transmitted.

If the owner token being carried in a lost message is dirty (that is, if some node has modified the data of the cache line since the last time that it was written back to memory) then that message may be carrying the only valid copy of the

data currently in the system. Hence, if the owner token is lost, no processor (or memory module) would send the data and a deadlock and possibly data loss would occur.

Finally, in TokenCMP, starvation avoidance is provided by means of distributed persistent requests (see section A). A persistent request is initiated by a *persistent request activation message* which is sent by the requester to every other coherence node (using broadcast). Only a single persistent request can be active for each node at the same time and the persistent request will be active until the requestor node deactivates it broadcasting a *persistent deactivation message*, which it will send once it receives the data and collects all the tokens that it requires.

The loss of either the activation or deactivation messages of a persistent request will lead to deadlock: if the activation message is not received by all the nodes which hold tokens, the requestor will not be able to collect them and will wait indefinitely; and if the deactivation message is not received by some node, that node will not be able to perform any request for that cache line, causing a deadlock if it later needs to read or write to that line.

4.2 Fault tolerance measures introduced by FtTokenCMP

Once we have seen the problems arising in TokenCMP due to the use of an unreliable interconnection network, we introduce a fault-tolerant coherence protocol intended to cope with these problems. The new coherence protocol, FtTokenCMP, detects and recovers from deadlock situations and avoids data loss.

Instead of ensuring reliable end-to-end message delivery, we have extended the *TokenCMP* protocol with fault tolerance measures.

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

Token counting rules for reliable data transference

The defining observation of the token framework is that simple token counting rules can ensure that the memory system behaves in a coherent manner. The token counting rules used by TokenCMP were created by Martin [67] and, along with the starvation avoidance mechanism provided by persistent requests, they define the correctness substrate of the coherence protocol. See appendix A for an explanation of the original rules.

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 fault-tolerant token counting rules are based of those introduced by Martin [67], and extend them to ensure reliable ownership transference (modifications 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 T tokens for that block, *none of them is 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 cannot also carry the backup token.*
- ***Backup Token Transfer Rule**: The backup token can only be sent to another component that already holds the owner token.*
- ***Blocked Ownership Rule**: The owner token cannot be sent to another 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 from 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 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 unreliable 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 acknowledgment* (see section 4.2). 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. Section 4.2 explains the actual mechanism for ownership transference used by FtTokenCMP, which complies with the above rules.

Fault detection

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

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.

Once a problematic situation has been detected, the main recovery mechanism used by our protocol is the *token recreation process* described later in section 4.2. That process resolves a deadlock ensuring both that there is the correct number of tokens and one and only one valid copy of the data.

We present a summary of all the problems that can arise due to loss of messages and their proposed solutions in table 4.3. In the rest of this section, we explain how our proposal prevents or solves each one of these situations in detail.

Avoiding data loss

In TokenCMP, the node that holds the owner token associated with a memory line is considered the owner node of that line and is responsible of sending the data to any other node that needs it, and writing it back to memory when necessary. Hence, transferring the owner token means transferring ownership, and this happens whenever a write request has to be answered since all the tokens need to be transferred to the requestor. It also happens at some other

Table 4.2: 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	A persistent request becomes active.	The starver cache.	The persistent request is satisfied or deactivated.	Request a token recreation.
Lost Data	A backup state is entered (the owner token is sent).	The cache that holds the backup.	The backup state is abandoned (the *ownership acknowledgment* arrives).	Request a token recreation.
Lost Backup Deletion Acknowledgment	A line enters the blocked state.	The cache that holds the owner token.	The blocked state is abandoned (the Backup Deletion Acknowledgment arrives).	Request a token recreation.
Lost Persistent Deactivation	A persistent request from another cache is activated.	Every cache (by the persistent request table).	The persistent request is deactivated.	Send a persistent request ping.

times, like when doing a write-back (the ownership is transferred from a L1 cache to L2 or from L2 to memory), when the first read request is answered by memory or when answering some read requests if a migratory-sharing optimization is used.

In both TokenCMP and FtTokenCMP, the owner token always travels along with data (see the token counting rules explained in appendix A and the fault-tolerant token counting rules in section 4.2) and many times the data included in such a message is the only up-to-date copy of the data available in the system. This does not happen when data travels in a message which does not contain the owner token, since in that case it is guaranteed that the owner node still has the owner token and a valid copy of the data.

To avoid losing data in our fault-tolerant coherence protocol, a component

Table 4.3: Summary of the problems caused by loss of messages in FtTokenCMP and the detection and recovery mechanisms employed

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

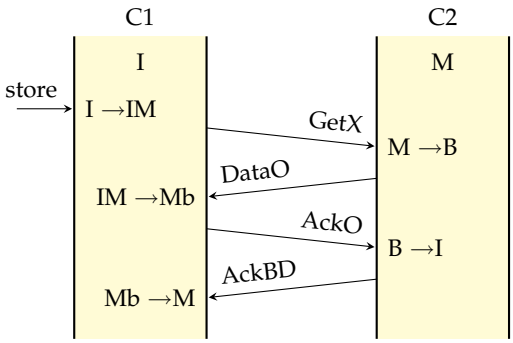
(cache or memory controller) that has to send the owner token will keep the data line in a *backup* state. This is required by the *owner token transfer rule* of section 4.2 which states that the sender must keep the backup token.

A line holding the backup token and no other token is considered to be in a backup state. A line in backup state will not be evicted from the component until an *ownership acknowledgment* is received, even if every other token is sent to other components. This acknowledgment is sent by every component in response to a message carrying the owner token. When the ownership acknowledgment is received, the backup token will be sent to the new owner and the backup data will be discarded. This is the only way that the backup token can be transferred due to the *backup token transfer rule*. The *blocked ownership rule* ensures that once an ownership acknowledgment has been sent, the owner token will stay at the same component until the backup token is received by that component. The message carrying the backup token is called a *backup deletion acknowledgment*

because it acknowledges to the new owner that the old backup data has been discarded.

While a line is in *backup* state its data is considered invalid and will be used only if required for recovery. Hence, the cache will not be able to read from that line.

Notice that it is possible for a cache to receive valid data and a non-backup token before abandoning a backup state, only if the data message was not actually lost. In that case, the cache will be able to read from that line, since it will be transitioned to an intermediate backup and valid state until the *ownership acknowledgment* is received.



Cache C1 receives a store request from its processor while having no tokens (I state), so it broadcasts a transient exclusive request (*GetX*). C2, which has all the tokens and hence is in *modified* state (M), answers to C1 with a message (*DataO*) carrying the data and all the tokens except the backup token and including the owner token. Other caches will receive the *GetX* message, but they will ignore it since they have no tokens. Since C2 needs to send the owner token, it goes to the *backup* state (B) and starts the *lost data timeout*. When C1 receives the DataO message, it satisfies the miss and enters the *modified and blocked* state (Mb), sending an ownership acknowledgment (*AckO*) to C2. When C2 receives it, it discards the backup, goes to *invalid* state (I), stops the *lost data timeout* and sends a *backup deletion acknowledgment* caryying the backup token (*AckBD*) to C1. Once C1 receives it, it transitions to the normal *modified* state (M).

Figure 4.1: Message exchange example of a cache-to-cache transfer using owned data loss avoidance in FtTokenCMP

A cache line in a backup state (that is, holding the backup token) will be used for recovery if no valid copy is available when a message carrying the owner token is lost. To be able to do this in an effective way, it is necessary to ensure that there is a valid copy of the data or one and only one backup copy at all times,

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

Hence, to avoid having several backup copies, a cache which has received the owner token recently cannot transmit it again until it is sure that the backup copy for that line has been deleted. In this situation, the line enters the *blocked ownership* state, following the *blocked ownership rule*. A line will leave this state when the cache receives a *backup deletion acknowledgment* which is sent by any cache when it deletes a backup copy after receiving an *ownership acknowledgment*. Figure 4.1 shows an example of how the owner token is transmitted with our protocol.

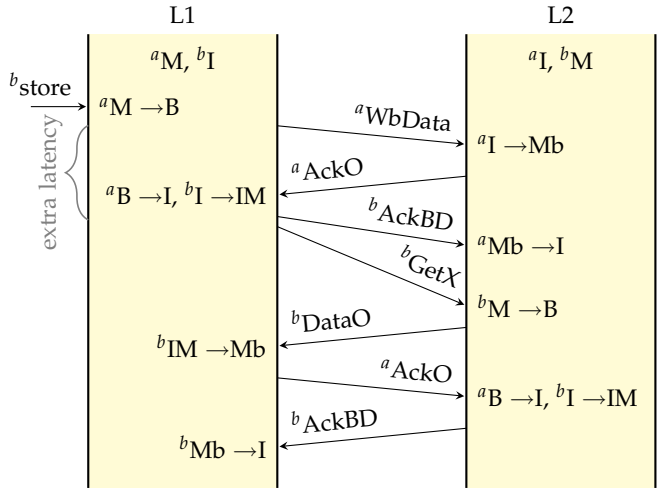
The two acknowledgments necessary to finalize the ownership transference transaction are out of the critical path of the miss, hence the miss latency is unaffected by this mechanism (ignoring the extra network traffic that could arguably increase the latency).

However, there is a period after receiving the owner token until the *backup deletion acknowledgment* with the backup token arrives during which a cache cannot answer to write requests because it would have to transmit the owner token, which is blocked. This blocking also affects persistent requests, which are serviced immediately after receiving the *backup deletion acknowledgment*. This blocked period could increase the latency of some cache-to-cache transfer misses, however we have found that it does not have any measurable impact on performance, as most writes to the same line by different processors are sufficiently separated in time. The blocked period lasts as long as the round-trip delay for control messages between the new owner and the old owner.

Even for highly contended cache lines, the blocking period is unlikely to affect performance. For example, for a highly contended lock, when the new owner receives the data message and enough tokens, it can already acquire the lock (writing to the cache line) and perform whatever work it has to do that required acquiring the lock. Except for very small critical sections, the backup deletion acknowledgment will likely arrive before the lock is released (writing again to the cache line, but without needing another ownership transference). This way, the lock can be immediately acquired by another processor.

The safe ownership transference mechanism also affects replacements (from L1 to L2 and from L2 to memory). We have found that the effect on replacements is much more harmful for performance than the effect on cache-to-cache transfer misses mentioned above. This is because the cache line cannot be actually evicted from the cache that needs to perform the replacement until the ownership

acknowledgment is received. Of course, this also affects the latency of the cache miss that initiated the replacement, as shown in figure 4.2.



Initially, we have memory line *a* with all the tokens in the L1 cache, and memory line *b* with all the tokens in the L2 cache. The L1 cache receives a store event for line *b*, but before it can issue a *GetX* request it needs to replace line *a* (to make room). For doing so, it sends the data and all the tokens, except the backup token, to L2 using a *WbData* message. It has to keep the data in the cache in B state until it receives an *AckO* message. When the *AckO* message is received by the L1 cache, it can send the backup token to L2 using an *AckBD* message, change the tag of the cache line to that of line *b*, and then issue the *GetX* message for line *b*. The rest of the transaction is handled as in figure 4.1. There is some extra latency with respect to TokenCMP because the replacement cannot be finished and (and the tag of the cache line updated) until the *ownership acknowledgment* is received. This extra latency can be avoided by moving the data of line *a* to a backup buffer as soon as the write-back starts.

Figure 4.2: Message exchange in FtTokenCMP without backup buffer for a miss including the required previous write-back

To alleviate the effect in the latency of replacements, we propose using a small *backup buffer* to store the backup copies. In particular, we add a backup buffer to each L1 cache. A line is moved to the backup buffer when it is in a backup state, it needs to be replaced and there is enough room in the backup buffer[1]. The backup buffer acts as a small victim cache, except that only lines in backup

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

states are moved to it. We have found that a small backup buffer with just 1 or 2 entries is enough to virtually remove the negative effect of backup states (see section 8.3). Alternatively, a write-back buffer could achieve the same effect.

Handling the loss of an owned data carrying message or an ownership acknowledgment

In TokenCMP, losing a message which carries the owner token means that possibly the only valid copy of the data is lost. However, with the mechanism described above there would still be an up to date backup copy at the cache which sent the data carrying message and still holds the backup token.

If the data carrying message does not arrive to its destination, no corresponding *ownership acknowledgment* will be received by the component holding the backup token (former owner), leading to a deadlock.

To detect this deadlock, we use the *lost data timeout*. It will start when the owner token is sent and stop once the ownership acknowledgment arrives. In other words, it is active while the line is in backup state.

The lost data timeout will also trigger if an *ownership acknowledgment* is lost. In that case, the backup copy will not be discarded and no *backup deletion acknowledgment* will be sent. Hence, the backup copy will remain in one of the caches and the data will remain blocked in the other. The *lost backup deletion acknowledgment timeout* (see section 4.2) may trigger too in that case.

When either timeout triggers, the cache requests a token recreation process to recover the fault (see section 4.2). The process can solve both situations: if the *ownership acknowledgment* was lost, the memory controller will send the data which had arrived to the other cache; if the data carrying message was lost, the cache will use the backup copy as valid data after the recreation process ensures that all other copies have been invalidated.

The loss of an owner token carrying message can be detected by the lost token timeout (see section 4.2) since the number of tokens decreases. However, that timeout is not enough to detect the loss of an ownership acknowledgment.

Handling the loss of the backup token

During the reliable ownership transference process described in section 4.2, the backup token is sent using a *backup deletion acknowledgment* in response to an *ownership acknowledgment* to signal that the backup has been discarded and to allow the new owner of the cache line to finish the blocking period and to be able to transfer ownership again if necessary.

When a *backup deletion acknowledgment* is lost, a line will stay indefinitely in a blocked ownership state. This will prevent it from being replaced or to answer any write request by another cache. Both things would eventually lead to a deadlock.

The first problem is detected by *lost backup deletion acknowledgment timeout*. To be able to replace a line in a blocked state when the *backup deletion acknowledgment* is lost, the *lost backup deletion acknowledgment timeout* is activated when the replacement is necessary, and deactivated when the *backup deletion acknowledgment* arrives. If it triggers, a *token recreation process* will be requested. Alternatively, the *lost backup deletion acknowledgment timeout* could be activated immediately after sending the *ownership acknowledgment*, but activating it only when a replacement is required allows us to deallocate the MSHR entry as soon as the miss has been satisfied.

For the second problem, if a miss cannot be resolved because the line is blocked in some other cache waiting for a backup token which has been lost, eventually a persistent request will be activated for it and after some time the *lost token timeout* will trigger too (see section 4.2). Hence, the *token recreation process* will be used to solve this case.

The token recreation process will solve the fault in both cases, since even lines in blocked states are invalidated and must transfer their data to the memory controller (see section 4.2).

Dealing with token loss

As explained in appendix A, in a token based cache coherence protocol every memory line has an associated number of tokens which should remain always constant. To be able to write to a memory line, a processor needs to collect all these tokens to ensure that no other processor can read from that line.

In TokenCMP all response messages carry at least one token. Hence, when a response message is lost, the total number of available tokens associated to that memory line will decrease. In that case, when a processor tries to write to that line either in the same coherence transaction or in a later one, it will eventually timeout after sending a transient request (or several of them depending on the particular implementation of token coherence) and issue a persistent request.

In the end, after the persistent request gets activated, all the available tokens in the whole system for the memory line will be received by the starving cache. Also, if the owner token was not lost and is not blocked (see section 4.2 for what happens otherwise), the cache will receive it too together with data. However,

since the cache will not receive all the tokens that it expects, it will not be able to complete the write miss, and finally the processor will be deadlocked.

We use the *"lost token timeout"* to detect this deadlock situation. It will start when a persistent request is activated and will stop once the miss is satisfied and the persistent request is deactivated. The value of the timeout should be long enough so that, in normal circumstances, every transaction will be finished before triggering this timeout. Using a value too short for any of the timeouts used to detect faults would lead to many false positives which would hurt performance, and would increase network traffic. In section 8.3 we explain how to determine optimal values for the timeouts.

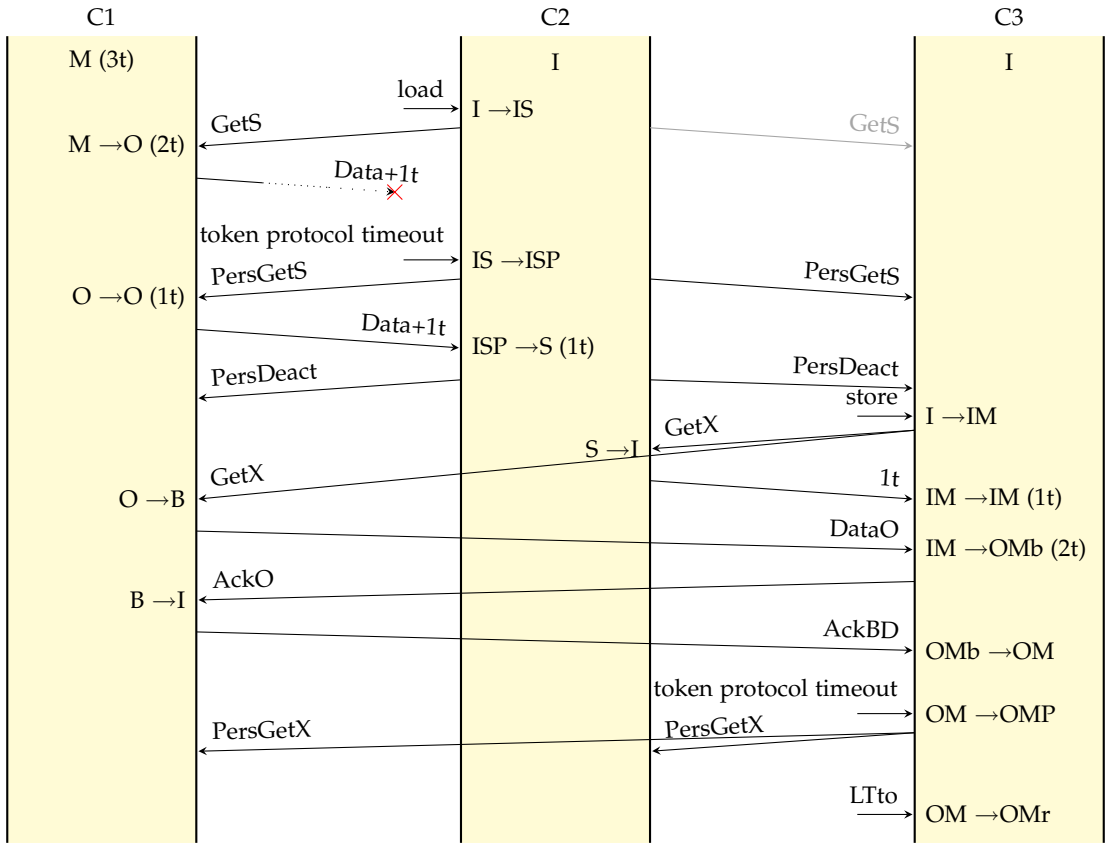
Hence, if the starving cache fails to acquire the necessary tokens within certain time after the persistent request has been activated, the *lost token timeout* will trigger. In that case, we will assume that some token carrying message has been lost and we will request a token recreation process for recovery to the memory module. By means of the *token serial number* (see section 4.2), this process will also take care of false positives of the *lost token timeout* which could lead to an increase in the total number of tokens and to coherence violations. Notice that the *lost token timeout* may be triggered for the same coherence transaction that loses the message or for a subsequent transaction for the same line. Once the token recreation has been done, the miss can be satisfied immediately.

Dealing with faults in persistent requests

Assuming a distributed arbitration policy as the one used by TokenCMP and FtTokenCMP, persistent request messages (both requests and deactivations) are always broadcast to keep the persistent request tables at each cache synchronized.

These persistent request tables have an entry for each processor that records whether that processor has an active persistent request and which memory line it is requesting. Each processor can have only one active persistent request at any given time and there is a priority mechanism in place to ensure fairness (see appendix A for details).

Losing either a persistent activation or deactivation message will lead to inconsistency among the tables in different processors. If the persistent request tables are inconsistent, some persistent requests may not be activated by some caches or some persistent requests may be kept activated indefinitely. In the first case, the cache would not forward any tokens that it had to the starver and in the second case the cache would not be able to make any request for that memory line itself. These situations could lead to starvation or deadlock.



In a system with three tokens per line plus the backup token, cache C1 currently holds all the tokens and hence is in M state. C2 receives a load event from its processor and then broadcasts a transient *GetS* request. C1 answers with a data message carrying also one token, but that message gets corrupted and never arrives to C2. After a short interval, the token protocol timeout triggers at C2 and C2 broadcasts a persistent request which is answered by C1 with another data message and another token. Afterwards, C3 receives an store event from its processor and broadcasts a *GetX* request which is answered by C2 with one token and by C1 with a *DataO* message that carries the data and the owner token, which is the only non-backup token left in C1. C3 sends an *AckO* message to C1 which answers with an *AckBD* carrying the backup token. Although C3 has now all the tokens in the system for the memory line, it cannot write to the line since it expects three non-backup tokens for doing so. Hence, C3 will issue a persistent request that will not be answered by anyone, and eventually the *lost token timeout* will trigger detecting the missing token.

Figure 4.3: Message exchange example of a token loss detected using the *lost token timeout*

Dealing with the loss of a persistent request activation

Firstly, it is important to note that the cache which issues the persistent request (also known as the *starver*) will always activate it as soon as any higher priority persistent request for the same line is deactivated, since it does not need to send any message through the interconnection network to update its own persistent request table.

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

Since the persistent request has been activated at the starver cache, the *lost token timeout* (see section 4.2) will trigger eventually and the token recreation process will solve this case too.

On the other hand, if the cache that does not receive the persistent request did not have tokens necessary to satisfy the miss, it will eventually receive an unexpected deactivation message which should ignore, but there would not be any negative effect. This means that in FtTokenCMP the persistent request table needs to be able to receive unexpected persistent request deactivation messages.

Dealing with the loss of a deactivation message

If a persistent request deactivation message is lost, the request will be permanently activated at some caches. This would make it impossible for those caches to issue any request for that memory line.

To avoid this problem, caches start the *lost persistent deactivation timeout* when a persistent request is activated and stop it when it is deactivated. In other words, this timeout is active while the persistent request is active. Note that the timeout is active at every cache except the one that issued the persistent request. Also, this timeout cannot use the same counter as the rest of timeouts (including the usual starvation timeout of TokenCMP), instead we need to add a counter for each entry in the persistent request table.

When this timeout triggers, the cache will send a *persistent request ping* to the starver. A cache receiving a *persistent request ping* will answer with a persistent request activation or deactivation message whether it has a pending persistent request for that line or not, respectively.

The *lost persistent deactivation timeout* is restarted after sending the *persistent request ping* to cope with the potential loss of this message.

Note that since the timeout is active at several caches at the same time, several *persistent request ping* messages may be sent almost simultaneously. There are two ways to deal with this without overloading the interconnection network with the responses:

- The response (which is either a persistent request activation or deactivation) can be sent only to the cache that sent the ping, instead of broadcast as these messages usually are. This is what our implementation does.

- Do not answer to persistent request pings received shortly after another persistent request ping for the same line has been received.

Additionally, if a cache receives a persistent request from a starver which already has an active persistent request in the persistent request table before the *lost persistent deactivation timeout* triggers, it should assume that the deactivation message has been lost and deactivate the old request, because each cache can have only one pending persistent request and a cache will not send a persistent request activation message before sending the persistent request deactivation message for the previous request.

Token recreation process

FtTokenCMP provides two fault recovery mechanisms: the *token recreation process* and the *persistent request ping* described in section 4.2. The *token recreation process* is the main fault recovery mechanism provided by our proposal and is requested by any coherence node when either the *lost token*, *lost data* or *lost backup deletion acknowledgment* timeouts triggers.

This process needs to be effective, but since it should happen very infrequently, it does not need to be particularly efficient. In order to avoid any race and keep the process simple, the memory controller will serialize the token recreation process, attending token recreation requests for the same line in FIFO order.

The process will work as long as there is at least a valid copy of the data in some cache or one and only one backup copy of the data or both things (the valid data or backup can be at the memory too).

The protocol guarantees that the above conditions are true at every moment despite any message loss by means of the fault-tolerant token counting rules (see sections 4.2 and 4.2). In particular, these conditions are true if no message has been lost, hence the *token recreation process* is safe for false positives and can be

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

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

Token serial numbers

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

All the valid tokens for the same memory line should have the same serial number. The serial number will be transmitted within every coherence response. Every cache in the system must know the current serial number associated with each memory line and must discard every message received containing an incorrect serial number. The *token recreation process* modifies the current *token serial number* associated with a line to ensure that all the old tokens are discarded. Hence, if there was no real fault but a token carrying message was delayed on the network due to congestion (a false positive), it will be discarded when received by any cache because the *token serial number* in the message will be identified as obsolete.

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

Since the *token serial number* table is finite, serial numbers are reset using the

owner token recreation mechanism itself whenever the table is full and a new entry is needed, since setting the *token serial number* to 0 actually frees up its entry in the table. The entry to be evicted can be chosen at random.

Additionally, when a token serial number needs to be reset (either to replace it from the token serial number table or because it has reached the maximum value and needs to be incremented again) the interconnect could be drained to ensure that there is not any old token still in the network.

Token recreation protocol

The token recreation process is orchestrated by the memory controller. It updates the token serial number of the memory line which is stored in the table of each cache, gathers the valid data or backup data and copies it to memory, invalidates all tokens and evicts the line from all caches, and finally, the memory sends the data to the node that requested the token recreation process, sends it to the current higher priority starver or keeps it if no node needs it currently.

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

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

   The *token recreation request* message includes a bit indicating whether the requestor needs the data after the recreation or not. The first case is the most common, and the memory will send the data and all the tokens to the requestor with the last message of the recreation process. The second case happens when the recreation is detected by a cache that holds the backup token (because the *lost data timeout* has triggered), and the memory will either keep the data itself or send it to the current higher priority starver after the recreation process.

2. When servicing a *token recreation* request, the memory will increase the *token serial number* associated to the line and send a *set token serial number* message to every cache (and to itself since it is also a coherence node, although no real message travels through the interconnection network for this purpose).

3. When receiving the *set token serial number* message, each cache updates the *token serial number* in its table and sends an acknowledgment to the memory. The *set token serial number acknowledgment* will also include data if the cache had valid data (even if it was in a blocked owner state) or a backup. A bit indicates whether the data sent is valid or a backup state.

4. When the memory receives a *set token serial number acknowledgment* it will decrease a counter of pending acknowledgments and take one of the following actions:

   - If the message does not contain data, no further action is required.
   - If the message contains backup data, no message containing valid data has been already received and the memory did not have valid data either, then the backup data is stored in memory and assumed to be valid, at least until a message with actual valid data arrives. Only one message with backup data can arrive, because the protocol guarantees that there is at most one backup copy of the data, as said above.
   - If the message contains valid data (and no previous message with valid data has been received) the data will be stored in memory, possibly overwriting backup data received in a previous message. Further messages with either valid data or backup data will be treated like dataless messages.

5. Once the memory has received valid data or once it has received all the acknowledgments (and hence, also the backup copy of the data), it will send a *forced backup and data invalidation* message to all other nodes.

6. When any cache receives the *forced backup and data invalidation*, it discards any token and any data or backup data that it could have.

   Since all the tokens held by a cache are destroyed, the state of the line will become invalid (or the appropriate intermediate state if there is any pending request), even if the line was in a blocked owner state or backup.

7. When the memory receives all the acknowledgments for both the *set token serial number* and the *forced backup and data invalidation* message, it will send a *token recreation done* message to the node that requested the recreation.

   At this point, the memory must have received one or more copies of the valid data of the memory line or, at least, a backup copy of the data which

it can use for recovery. Also, due to the *forced backup and data invalidation* messages, no other node has any data or token.

If the requestor of the recreation process indicated that it wanted to receive data, the data (and all the tokens) will be sent to it after the *token recreation done* message using the safe ownership transference mechanism described in section 4.2.

Otherwise, either the data and tokens will be kept at the memory or they will be sent to the current higher priority starver, if it exists, also using the safe ownership transference process described in section 4.2.

At the end of the process, either the memory or one of the caches has the only valid copy of the data and all the tokens.

Handling faults in the token recreation process

As said above, the efficiency of the token recreation process is not of great concern, since it should happen only when one of the fault detection timeouts triggers (hopefully only after a fault actually occurs). Hence, we can use unsophisticated (brute force) methods to avoid problems due to losing the messages involved.

For this reason, we could use point to point acknowledgments for all the messages involved in the *token recreation process*. These acknowledgments should not be confused with the acknowledgments mentioned in section 4.2 (which actually would need these acknowledgments too). These messages are repeatedly sent every certain number of cycles (2000 in one of our implementations) until an acknowledgment is received. Serial numbers can be used to detect and ignore duplicates unnecessarily sent. The acknowledgments are sent as soon as the message is received, not when it is actually responded.

A simpler way to recover errors in the token recreation process is to restart the process when it takes too much time to complete. This requires the addition of only one timeout counter in each memory controller which is started when the recreation process begins and is stopped when the last acknowledgment arrives. To cover for the loss of the *token recreation request*, the cache that sends the request should restart the timeout that detected the (potential) fault instead of stopping it so that it will send a second *token recreation request* if the first one is lost. If the memory receives two token recreation requests for the same memory line and from the same node, it should ignore the second one.

4.3 Hardware overhead of FtTokenCMP

The fault tolerance measures that FtTokenCMP adds to TokenCMP incur a certain overhead, both in terms of performance and extra hardware required to keep coherence. Performance overheads are measured in chapter 8. In this section we make an approximation to the hardware overheads required to implement the proposed fault tolerance measures.

Firstly, FtTokenCMP uses a number of timeouts for fault detection. However, most of the timeouts employed to detect faults can be implemented using the same hardware already employed by TokenCMP to implement the starvation timeout required by token coherence protocols, although the counters may need more bits since the new timeouts are longer. Only for the *lost persistent deactivation timeout* it is necessary to add a new counter per processor at each cache and at the memory controller (one counter per persistent table entry).

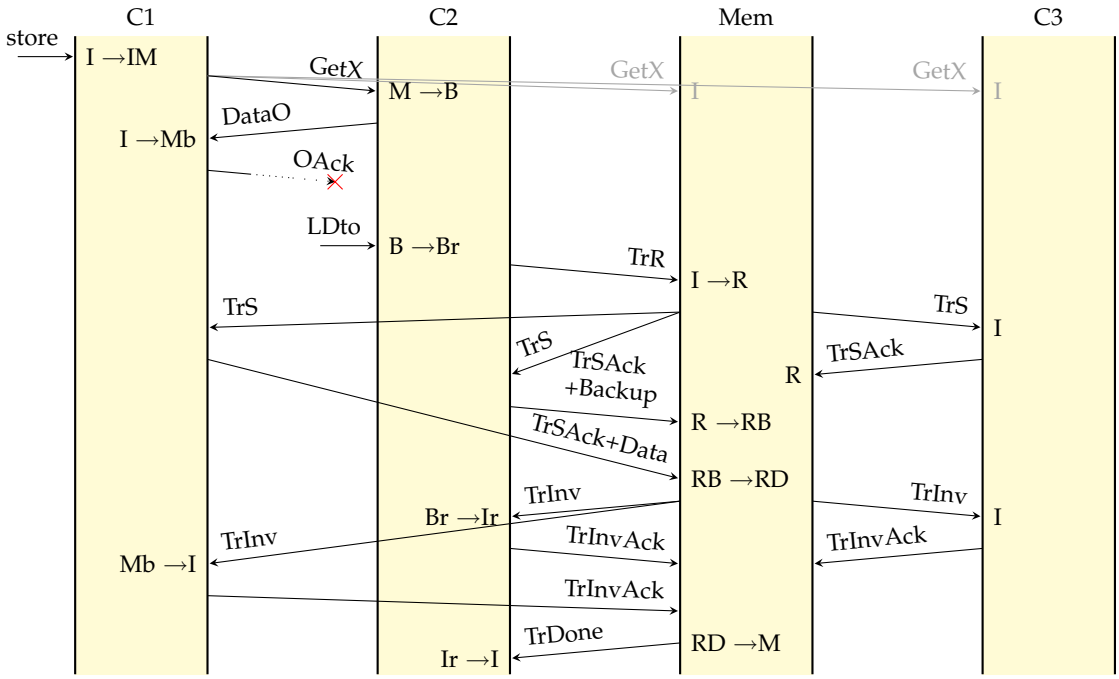
Probably the highest overhead in terms of additional hardware is caused by the token serial number tables. To implement them, we have added a small associative table at each cache (each L1 and L2 bank) and at the memory controller to store those serial numbers whose value is not zero. In this work, we have assumed that each serial number requires a small number of bits (two bits were used in our tests) and that the number of lines whose token serial number is different than 0 is usually small. Hence, to implement each token serial number table we have used a small (16 entries) associative table which has an entry for every line whose token serial number is different than zero. If the tokens of any line need to be recreated more than 4 times, its counter will wrap and the serial number of the line will become 0 again effectively freeing an entry from the table. If the token serial number table fills up because more than 16 lines need to have their tokens recreated, the least recently modified entry will be chosen for eviction. To evict the entry, it is enough to reset to 0 the token serial number associated with that line using the token recreation process.

Additionally, some hardware is needed to calculate and check the error detection code used to detect and discard corrupt messages. Both the check for the token serial number and the error detection code can and should be done in parallel with the access to the cache tags or to the MSHR to avoid increasing the response latency in the common case.

Due to the exchange of acknowledgments required to ensure reliable ownership transference, the worst case message dependence chain of FtTokenCMP is two messages longer that in the case of TokenCMP. Hence, FtTokenCMP may require up to two more virtual networks than TokenCMP to be implemented,

depending on the particular topology of the interconnection network and the deadlock avoidance mechanism employed by the network (see appendix C).

Finally, to avoid a performance penalty in replacements due to the need of keeping the backup data in the cache until the ownership acknowledgment has been received, we have proposed to add a small backup buffer at each L1 cache. The backup buffer can be effective having just one entry, as will be shown in section 8.3.



In a transaction like the one of figure 4.1 the *AckO* gets lost. Hence, C2 keeps the line in backup state (B) and, after some time, the *lost data timeout* triggers (*LDto*) and C2 sends a *token recreation request* message (*TrR*) to the memory controller and enters the *backup and recreating* state (Br). Since C2 does not have any pending miss for that line, it will indicate that it does not need the data after the recreation process is done (using a bit in the *TrR* message). When the memory controller receives that message, it goes to a *token recreation* state (R) and sends a *set token serial number* message (*TrS*) to each node. C3 receives it and answers with an acknowledgment (*TrSAck*). C2 is in *backup* state (B), and, when it receives the *TrS*, it answers with an acknowledgment message which carries the backup data too (*TrSAck+Backup*). C1 is in *modified and blocked* state, hence it returns an acknowledgment with data (*TrSAck+Data*). When the memory receives the *TrSAck+Backup* message, it writes the data to memory and transitions to a *token recreation with backup data* (RB) state. Later, when it receives the *TrSAck+Data*, it overwrites the backup data with the newly received data and transitions to a *token recreation with valid data* state (RD). If the *TrSAck+Backup* had arrived after the *TrSAck+Data*, it would have been ignored. When the memory receives the data, it sends a backup and data invalidation message (*TrInv*) to each cache. C3 receives it and answers with an acknowledgment (*TrInvAck*) without changing its state, since it has no tokens. C1 and C2 discard their tokens and valid and backup data respectively and transition to *invalid* (I) before sending the *TrInvAck*. Once the memory receives all the acknowledgments, it transitions to M state (that is, the memory will recreate a new set of tokens and hold all of them itself) and sends a *TrDone* message to C2.

Figure 4.4: Message exchange in a token recreation process (used in this case to recover from the loss of an *ownership acknowledgment*)

Chapter 5

A directory-based fault-tolerant cache coherence protocol

In the previous chapter we have described how a token-based cache coherence protocol can be extended with fault-tolerant measures so that it can be guaranteed correct executions even when the on-chip interconnection network loses some messages. Unfortunately, token coherence is not the cache coherence protocol of choice in current CMP proposals. However, we can adapt the techniques used in chapter 4 to most types of protocols. In this chapter, we will adapt them for directory-based cache coherence, which is a widely used and well understood class of protocols which is often proposed as the cache coherence mechanism of choice for future many-core CMPs [72].

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 [68] protocols which usually require frequent broadcasts, directory-based ones are more scalable and energy-efficient.

In this chapter we explain the design of a fault-tolerant directory-based cache coherence protocol which we call FtDirCMP and that is based on DirCMP, a directory-based cache coherence protocol that we describe in appendix B.

The design of FtDirCMP was heavily influenced by the lessons learned while designing FtTokenCMP. Both share several fault tolerance characteristics:

- Messages are assumed to either arrive correctly to their destination or not arrive at all. If a corrupted message arrives to any node, the corruption will be detected using error detection codes and the message will be discarded.

- The integrity of data when cache lines travel through the network is ensured by means of explicit acknowledgments out of the critical path of cache misses. These acknowledgments are used only for a few selected messages which cannot be discarded without losing data.
- A number of timeouts are used to detect deadlocks and trigger fault recovery mechanisms. However, the number of timeouts which are necessary and the function of each timeout depend on the particular coherence protocol.
- Ping messages are used in some cases for deadlock recovery. However, both protocols have another recovery mechanism which is different in each case.

However, the implementation of the above measures is different for FtDirCMP and FtTokenCMP. Moreover, some mechanisms are significantly different between both protocols:

- FtDirCMP does not have a centralized recovery mechanism like the *token recreation process* (see section 4.2). When a potential fault is detected, FtDirCMP either uses ping messages or reissues requests for recovery.
- FtDirCMP uses *request serial numbers* (see section 5.2) to avoid creating incoherences due to stale responses to reissued requests in case of false positives. Their function is analogous to *token serial numbers* (see section 4.2), but *request serial numbers* are more scalable and easier to implement.

5.1 Problems caused by an unreliable interconnection network in DirCMP

Table 5.1 shows a summary of the message types used by DirCMP, their function, and the effect that losing them has in the execution of a parallel program.

As can be seen in the table, 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. Hence, we will be able to use timeouts to detect all faults.

Additionally, losing some data carrying messages 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. *Data* messages can be lost without losing data because the node that sends them in response to a read request will always keep a copy of the data in *owned* state.

Table 5.1: Message types used by DirCMP and the effect of losing them

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

Differently than TokenCMP, there are no messages in DirCMP which could be lost without causing any problem.

Since write access to a line is only granted after all the necessary invalidation acknowledgments have been actually received, losing any message cannot lead to a cache incoherence, as stated in section 3.2.

5.2 Fault tolerance measures introduced by FtDirCMP

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. It will detect and recover from deadlocks and avoid data loss.

To avoid data loss, FtDirCMP uses extra messages to acknowledge the reception of a few critical data messages. When possible, those messages are kept out of the critical path of any cache miss so that the miss latency is unaffected. Also, they are piggybacked in other messages in the most frequent cases to reduce the network traffic overhead. This mechanism is explained in section 5.2.

To implement fault tolerance, FtDirCMP adds a few new message types to those shown in table 5.1. Table 5.2 shows the new message types and a summary of their function. The following sections explain their roles in more detail.

Table 5.2: Additional message types used by FtDirCMP

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

As previously said, thanks to the fact that every message lost in DirCMP leads to a deadlock, FtDirCMP can use timeouts to detect potentially lost messages. FtDirCMP uses a number of timeouts to detect faults and start corrective measures. Table 5.3 shows a summary of these timeouts, table 5.4 shows which timeout detects the loss of each message type, and sections 5.2, 5.2, 5.2 and 5.2 explain each timeout in more detail.

As can be seen in table 5.3, FtDirCMP provides two types of recovery mechanisms: issuing ping messages and reissuing some requests. The recovery mechanism used in each case depends on the timeout which detects the possible fault.

Table 5.3: 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	A request is issued.	At the requesting L1 cache.	The request is satisfied.	The request is reissued with a new serial number.
Lost Unblock	A request is answered (even write-back requests).	The responding L2 or memory.	The unblock (or write-back) message is received.	An *UnblockPing*/*WbPing* is sent to the cache that should have sent the *Unblock* or write-back message.
Lost backup deletion acknowledgment	The *AckO* message is sent.	The node that sends the *AckO*.	The *AckBD* message is received.	The *AckO* is reissued with a new serial number.
Lost data	Owned data is sent through the network.	The node that sends owned data.	The *AckO* message is received.	An *OwnershipPing* is sent.

Ping messages are used in situations where some component is waiting for some message from other component to be able to finalize a transaction. For example, this happens when an *Unblock* message is lost, since these messages are used to inform the directory (or memory controller) that a transaction has finished and the next pending transaction can be attended.

Most often, when a fault occurs and a timeout triggers, FtDirCMP reissues the request using a different serial number. The protocol assumes that some message involved in the transaction has been lost due to a transient fault and that trying the transaction a second time will work correctly. The need for request serial numbers is explained in section 5.2. 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 for the same line is satisfied, because that previous request may be precisely the older instance of the request that is being reissued. Hence, the L2

Table 5.4: Timeout which detects the loss of each type of message in FtDirCMP

Type	Timeout which detects its loss
GetX	Lost request timeout.
GetS	Lost request timeout.
Put	Lost request timeout.
WbAck	Lost request timeout.
WbAckData	Lost request timeout.
WbNack	Lost request timeout.
Inv	Lost request timeout.
Ack	Lost request timeout.
Data	Lost request timeout.
DataEx	Lost request timeout or lost data timeout.
Unblock	Lost unblock timeout.
UnblockEx	Lost unblock timeout.
WbData	Lost unblock timeout or lost data timeout.
WbNoData	Lost unblock timeout.
AckO	Lost backup deletion acknowledgment timeout.
AckBD	Lost backup deletion acknowledgment timeout.
UnblockPing	Lost unblock timeout.
WbPing	Lost unblock timeout.
WbCancel	Lost unblock timeout.
OwnershipPing	Lost data timeout.
NackO	Lost data timeout.

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.

Reliable data transmission

As mentioned in section 3.2, 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.

In a MOESI protocol, there is always one owner node for each line. This owner node is responsible of sending data to other nodes to satisfy read or write requests and of performing write-back when the data is modified. Hence, the owner will always have an up-to-date copy of the data of the memory line.

Due to the fact that the owner always has a copy of the data, data transmission is required to be reliable only when ownership is transferred. If a message transferring data but not ownership is lost while it travels through the network, it can always be resent by the owner node. But when ownership is being transferred, the message transferring the owned data cannot be resent if lost because it may be the only up-to-date copy of the data.

In DirCMP and FtDirCMP, ownership can be transferred either with an exclusive data response or a write-back response; and it is initiated either due to a request from a lower level cache, a forwarded request from a higher level cache or because a node needs to write the data back due to a replacement.

Exclusive data responses are sent in response to an exclusive access request (*GetX*), or in response to a shared access request (*GetS*) if there are no current sharers when the request is received by the directory, or due to the migratory sharing optimization.

In order to ensure reliable data transmission of owned data, FtDirCMP keeps a backup of the data when transferring ownership until it receives confirmation from the new owner that the data has been received correctly.

To this end, FtDirCMP adds some additional states to the usual set of MOESI states[1] used by the non fault-tolerant protocol:

- **Backup (B)**: This state is similar to the Invalid (I) state with respect to the permissions granted to the local node, but the data line is kept in the cache to be used for potential recovery if necessary.

  A line will enter a Backup state when the ownership needs to be transferred to a different cache (that is, when leaving the Modified, Owned or Exclusive states) and will abandon it once an *ownership acknowledgment* message is received, either as an *AckO* message by itself or piggybacked with an *UnblockEx* message.

- **Blocked ownership (Mb, Eb and Ob)**: These states are similar to the standard Modified, Exclusive and Owned (M, E and O), respectively. The difference is that a node cannot transfer the ownership of a line in one of these states to another node.

  The purpose of these states is to prevent having more than one backup for a line at any given point in time, which is important to be able to recover

[1] In the actual implementation there are also many intermediate states which are not considered in this explanation for simplicity, both in the non fault-tolerant and in the fault-tolerant protocols.

in case of a fault. A cache that acquires ownership (entering the blocked 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.

While a line is in one of these states, the cache will not attend external requests to that line which require ownership transference, but the cache can read and write to the line just like in the corresponding unblocked state.

The requests received while in these states will be queued to be attended once the *backup deletion acknowledgment* has been received. However, note that they could also be discarded since the protocol has provisions for reissuing lost requests. This property can be useful to avoid having to increase any buffer size, specially in the L1 cache controllers.

The states described above mimic the effect of the token counting rules for reliable data transference described in section 4.2. With them, the transmission of owned data between two nodes works as follows:

1. When a node sends owned data to another node (leaving the M, O or E state), it does not transition to an *Invalid* state like in a standard directory protocol. Instead, it enters a *Backup* (B) 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 or in a write-back buffer, as explained for the FtTokenCMP protocol in section 4.2. 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 directly to an M, O or E state like an usual directory protocol would do. 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 be allowed to 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* can be interpreted as a "request to delete the old backup". Hence, to be able to reissue it in case that it is lost (see sections

5.2 and 5.2), it will carry a request 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 5.1 shows an example of how a cache-to-cache transfer miss which requires ownership change is handled in FtDirCMP and compares it with DirCMP.

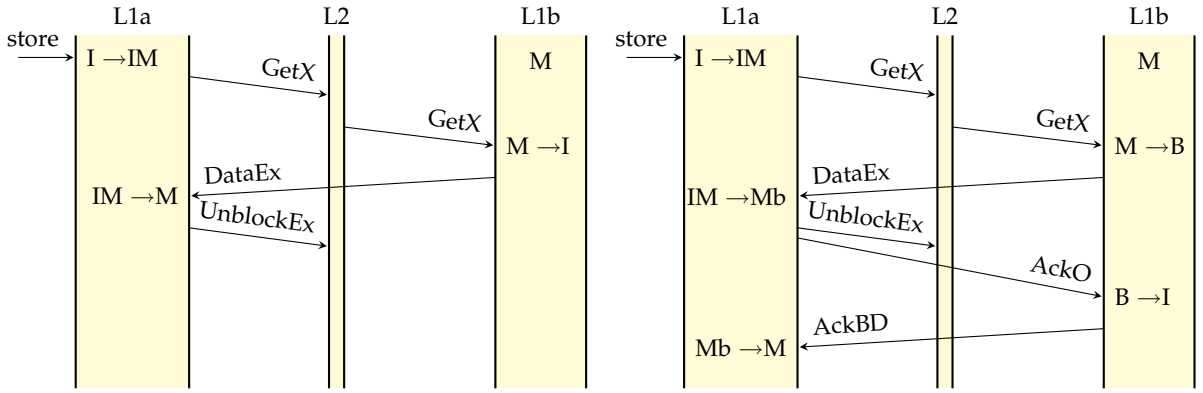
As will be shown in chapter 8, the main overhead introduced by FtDirCMP when compared to DirCMP is the extra network traffic incurred during the reliable ownership transference due to the pair of acknowledgments that it entails. To reduce this overhead, the ownership acknowledgment can be piggybacked 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 5.2.

The ownership transference procedure described in this section ensures that for every memory 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.

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, those rules still create potential performance problems due to 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.

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



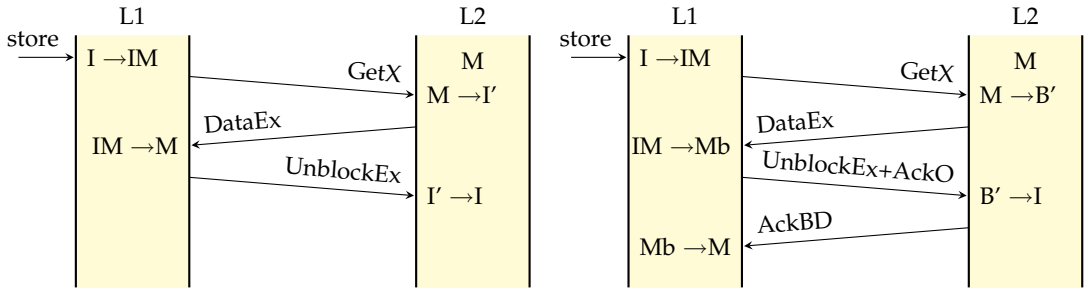
(a) DirCMP (b) FtDirCMP

Initially, for both protocols, L1b has the data in modifiable (M), exclusive (E) or owned[2] (O) state and L1a receives a store request from its processor and requests write access to L2 which forwards the request to L1b. In DirCMP, L1b sends the data to L1a 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. 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. Notice that L1a can use the data as soon as it arrives (at the same moment than in DirCMP and that L2 start attending the next pending miss for that memory line as soon as the *UnblockEx* message arrives (like in DirCMP too). When L1b receives the *AckO*, it discards the backup data, transitions to invalid (I) state and sends a *AckBD* message to L1a, which transitions to the usual modifiable (M) state when receives it.

Figure 5.1: Message exchange for a cache-to-cache write miss

As discussed for FtTokenCMP, this is not a problem when the data is received by an L1 cache since the node that requested the data can already use it while it waits for such acknowledgment. Only later requests to the same address from other nodes could be affected: in the worst case the second request would see its latency increased by as much time as the round-trip time of the acknowledgments between the new owner to the former owner. We have not been able to measure any slowdown in applications due to this problem.

In any case, that worst case is highly unlikely except for highly contended memory lines, like locks. In the particular case of locks, the negative effect is further reduced because when a node requests exclusive access to a memory line where a lock is stored, it does it usually in order to acquire the lock. Usually



(a) DirCMP (b) FtDirCMP

In both protocols, the L1 cache sends a *GetX* message 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 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 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, performs the store, and answers with a message which serves both as the *UnblockEx* and as the *AckO* messages used in figure 5.1. The backup copy in L2 will be kept until the *UnblockEx+AckO* message send by L1 is received and the *AckBD* message is sent to the L1, which transitions to the modifiable (M) state when receives it.

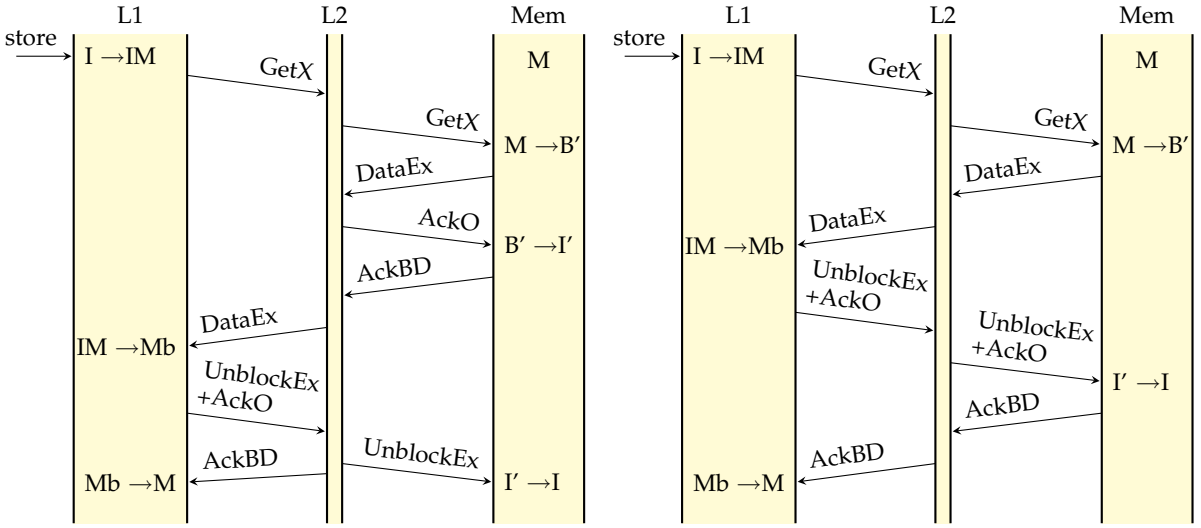
Figure 5.2: Message exchange for an L1 write miss that hits in L2

that node needs to do some work before it releases the lock (writing again to the memory line) and another node can acquire it. Hence, slightly increasing the latency of the coherence miss that happens when a second node tries to acquire the lock has no actual effect as long as that increase is less than the time required by the first node to perform work inside of the critical region protected by the lock.

However, in the case of L2 misses, the L2 cannot forward the data to the L1 cache to answer the 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*. Only then, it would send the data to L1 (keeping a backup), wait for another *ownership acknowledgment* from L1 and then send another *backup deletion acknowledgment*. Hence, in the case of L2 misses, the rules above would add two messages in the critical path of cache misses.

To avoid increasing the latency of L2 misses, we relax the rules in these cases while still ensuring that recovery is possible. As an optimization, 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 5.3 shows an example of how an L2 miss would be resolved without and with this optimization.



(a) Unoptimized (b) Optimized

In both cases, the L1 sends a *GetX* message to L2 which, since it does not have the data, forwards it to the memory controller. The memory controller fetches the data and sends it to L2 using a *DataEx* message. Now, in the unoptimized case, when the L2 receives the owned data, it sends an ownership acknowledgment to the memory controller and waits for the backup deletion acknowledgment before answering to L1 with the data. Once L1 receives the data, it sends a message to L2 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. On the other hand, in the optimized version, when the L2 receives the *DataEx* message from memory, it sends another *DataEx* message 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 with the unblock and the ownership acknowledgment to L2 which will then send the unblock to memory and the backup deletion acknowledgment to L1. Finally, when the memory controller receives the ownership acknowledgment, it will answer with a backup deletion acknowledgment message.

Figure 5.3: L2 miss optimization of ownership transference

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, but it can send it to an L1 cache keeping a backup until the respective ownership acknowledgment is received.

Additionally, 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, although it can move from one cache of the chip to another cache of the same chip.

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 correct recovery 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, but it could be transferred to memory for write-back purposes, although FtDirCMP does not currently do that in any case, making the internally blocked states equivalent to the full blocked states. The internally blocked states would be more useful in a multiple CMP setting, since in that scenario the L2 could transfer ownership to another chip before it receives the *backup deletion acknowledgment* from the L1 cache.

Faults detected by the *lost request timeout*

The purpose of the *lost request timeout* is to allow the requester to detect when a request is taking too long to be fully satisfied. As can be seen in table 5.4, it detects the loss of most messages, and hence is the fault detection timeout that most frequently triggers.

This timeout starts at the L1 cache when a request (*GetX*, *GetS* or *Put* message) is issued to the L2 and stops once it is satisfied, that is, when the L1 cache acquires the data and the requested access rights for it or when the L1 receives a write-back acknowledgment (*WbAck* or *WbAckData* messages) from L2. 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 lost, causing a deadlock.

This timeout is also used for write-back requests from the L2 to the memory

controller (*Put* messages). In this case, the timeout starts when the *Put* message is sent and stops once the write-back acknowledgment (*WbAckData* message) is received. When it triggers, the *Put* message will be reissued with a different serial number.

In the case of write-backs (both from L1 to L2 or from L2 to memory), 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* (see section 5.2).

It is maintained by the L1 for each pending miss and by L1 and L2 for each pending replacement. Hence, the extra hardware required to implement it is one extra counter for each MSHR entry.

When the *lost request timeout* triggers, FtDirCMP assumes that some message which was necessary to finish the transaction has been lost due to a transient fault and retries the request. The particular message that may have been lost is not very important: it can be the request itself (*GetX*, *GetS* or *Put*), an invalidation request sent by the L2 or the memory controller (*Inv*), a response to the request (*Data*, *DataEx*, *WbAck* or *WbAckData*) 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 arriving with the old serial number after the *lost request timeout* triggers in case of a false positive. See section 5.2 for more details.

Faults detected by the *lost unblock timeout*

FtDirCMP uses the on-chip directory to serialize requests to the same memory line. When the L2 receives a request (*GetS* or *GetX*) from the L1, it will answer it and block the affected memory line until it receives a notification from the L1 that the request has been satisfied. 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 memory line, if any.

Hence, when an unblock message is lost, the L2 will be blocked indefinitely and will not be able to attend further requests for the same memory 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. The timeout will be stopped when the unblock message arrives. When this timeout triggers, the L2 will send an *UnblockPing* message to the L1. See section 5.2 for more details about this recovery mechanism.

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

Write-Backs are serialized with the rest of requests by the L2 directory (using three-phase write-backs), but in this case the message which is expected by the L2 to proceed with the next requests is the write-back message itself (*WbData* or *WbNoData*). Hence, this timeout is also used to detect lost write-back messages 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 write-back message and assumes that the write-back is already done. Once the write-back message arrives to L2, the *lost unblock timeout* is deactivated.

If the write-back 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. See section 5.2 for details about how these messages are handled.

Faults detected by the *lost backup deletion acknowledgment timeout*

As explained in section 5.2, when ownership has to be transferred from one node to another, FtDirCMP uses a pair of acknowledgments (*AckO* and *AckBD*) to ensure the reliable transmission of the data. These acknowledgments are sent out of the critical path of a miss and are often 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* timeouts (unless the ownership acknowledgment was lost along with an unblock message) because these timeouts are deactivated once the miss has been satisfied. If the *backup deletion acknowledgment* (*AckBD*) is lost, the memory line will remain indefinitely in a blocked ownership state and it will not be possible to write the line back to memory nor answer any further write request for that line. On the other hand, if the

ownership acknowledgment (*AckO*) is lost the same will happen and, additionally, the former owner will not be able to discard its backup cache which will waste space in the cache or in the write-back buffer.

To detect the deadlock situations described above, we intoduce the *lost backup deletion acknowledgment* timeout whose purpose is to detect when the exchange of acknowledgments takes too much time. It 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 (see section 5.2). To ensure that ownership acknowledgments can be reissued even in case of false positives of this timeout, a node which receives one *AckO* must discard its backup copy if it had any and answer with a backup deletion acknowledgment whether it actually had a backup copy or not.

Reissued ownership acknowledgments will always be sent as independent *AckO* messages, never piggybacked with *UnblockEx* messages.

Faults detected with the *lost data timeout*

The rules described in section 5.2 guarantee that when an owned data carrying message is corrupted and discarded due to its CRC or discarded due to a wrong serial number, the data will always be in backup state in the former owner. Ownership transference starts when an owner node receives either a *GetX* message (which may be forwarded from L2 to an owner L1 cache) or a *WbAckData* message received from L2 by an L1 cache after the L1 cache requests a write-back (or received by L2 from the memory controller).

Hence, usually when a message carrying owned data is lost, 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 write-back, the *lost unblock timeout* will trigger in the L2 (or memory), and the data will be resent when the *WbPing* message is received.

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. This will be always the case in FtDirCMP as long as the network guarantees point-to-point ordering of messages. That is, as long

as two messages sent from one node to another are guaranteed to arrive in the same relative order as they were sent (if both of them arrive uncorrupted).

Interconnection networks with point-to-point ordered message delivery are very common and useful. However, not all interconnection networks have this property. In particular, any network that provides adaptive routing (which is itself useful for fault tolerance among other things) will not be able to provide point-to-point ordering guarantees.

In absence of faults, DirCMP will work correctly either over an ordered or an unordered point-to-point interconnection network. However, FtDirCMP may not work correctly over an unordered point-to-point network even in absence of faults due to the possibility of a reissued request arriving to its destination even after the original request and several other requests have been satisfied.

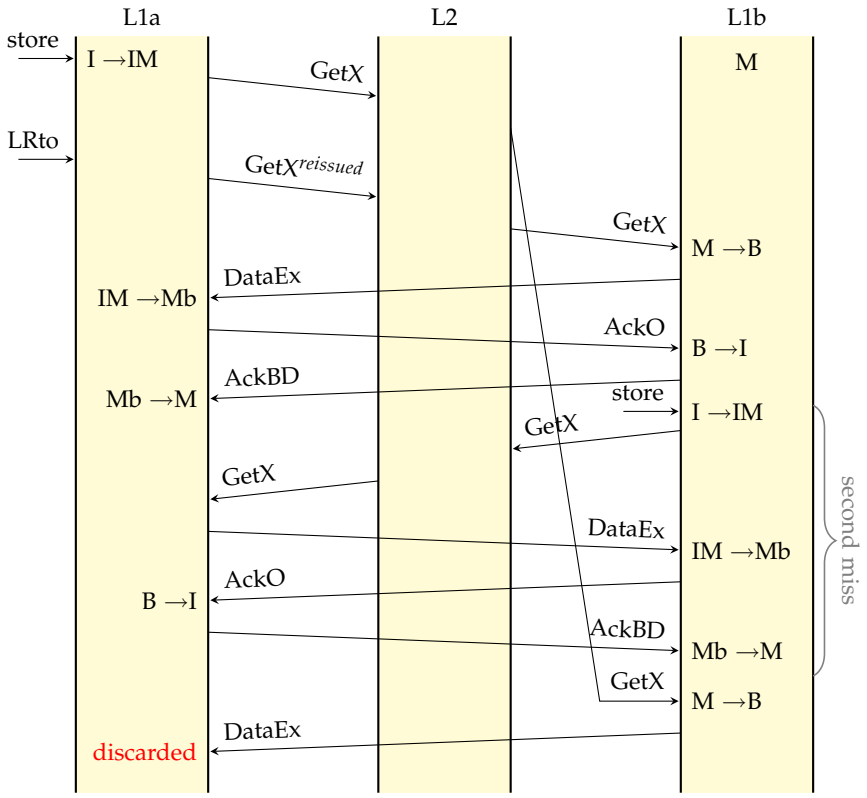
This is because if request messages can be reordered while traveling through the network, it can happen that the owned data is sent to some node which does not expect the data and hence will discard it. Since in that case the data will be kept only in backup state in the sender node, no node will be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

For this to happen, one *GetX* request has to arrive to an owner after it has been reissued and satisfied, and other request has returned the ownership to the same owner. Hence, it is a very unlikely event which requires a very congested network, a short *request timeout* and a particular sequence of requests.

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. Its purpose is to detect when the owned data has been unnecessarily discarded. 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 necessary only if the network guarantees point-to-point ordering of messages (see section 5.2).

Figure 5.4 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 *OwnershipPing* message with a new serial number is sent to the node that was sent the data before. See section 5.2 for more details.



L1a makes a request to L2 which forwards it to L1b. The forwarded message gets delayed in the network and hence the *lost request timeout* triggers in L1a. When this happens, L1a reissues the request which is forwarded again by L2. This time, the request arrives to L1b which sends the data to L1a. When L1a receives the data, it sends an unblock message to L2 and an ownership acknowledgment to L1b which responds with a backup deletion acknowledgment to L1a. Later, L1b makes a new request which is handled in a similar way (second miss). So at the end, L1b has the only copy of the data again with exclusive access. When the first forwarded request from the first miss 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 memory line would cause a deadlock (unless L1a requests the data again before anyone else).

Figure 5.4: Transaction where the *lost data timeout* detects data which has been wrongly discarded

Reissuing requests and request serial numbers

The most often used recovery mechanism in FTDIRCMP is to reissue a message which has been potentially lost. This is done with requests (*GetX*, *GetS* and *Put*)

when the *lost request timeout* triggers, ownership acknowledgments (*AckO*) when the *lost backup deletion timeout* triggers, and reissued *OwnershipPing* messages when the *lost data timeout* triggers more than once.

Message loss detection is done by means of a number of timeouts. This cannot be done with absolute exactness because, in general, the time to receive a response for any message in unbounded. Moreover, it is not desirable to choose a value too large for any of these timeouts because that would increase the latency for fault detection. In other words, it will be inevitable to have false positives. 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. Hence, the fault recovery measures taken after the timeouts trigger must be safe even if no message had been actually lost.

For example, 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, we cannot know for sure which of the messages involved in the transaction (request, response, invalidations, acknowledgments, etc) has been lost, if any.

Hence, 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 incoherence. For this reason, FtDirCMP uses *request serial numbers* to discard responses which arrive too late, when the request has already been reissued.

Figures 5.5 and 5.6 show cases where not using request serial numbers to discard a message that arrives too late would lead to incoherency or to using stale data.

Note that most times late responses are actually harmless because, even in the case of invalidation acknowledgments, which are the most obvious source of risk, when no faults occur it is very hard to arrive to an incoherency. However, an unlucky series of requests issued at the wrong time can cause problems even on a point-to-point ordered interconnection network, as shown in figure 5.5.

An unordered point-to-point interconnection network makes these problems more likely. For example, the situation depicted in figure 5.6 can only happen when an unordered point-to-point interconnection network is assumed.

For these reasons, FtDirCMP uses *request serial numbers*. Every request and every response message carries a serial number. These request serial numbers are chosen by the node which starts the transaction. That is, the L1 cache that issues the request, the L2 in case of write-backs from L2 to memory, or the current owner in an ownership transference (who has to choose a serial number for the ownership acknowledgment, see section 5.2).

Responses or forwarded requests will carry the serial number of the request that they are answering to. This way, when a request is reissued, it will be assigned a different serial number which will allow the requester 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 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. For this reason, it is preferable to discard any kind of message when it has an unexpected serial number even if it would be mostly harmless accepting them anyway. For example, unblock messages could be accepted even if they have a wrong serial number without compromising correctness.

This way, when the *lost request timeout* triggers, the requesting node will reissue its request with a new serial number and will discard any acknowledgment that it had already received (if any). Then the protocol will behave as follows:

- If the request message itself was lost, the new message will (most likely) arrive to its destination and the request will be eventually serviced as usual.
- If some response message was lost, the new request will arrive to the L2 directory, who will notice that this is a reissued request and will answer it again, without waiting for the current request to finish. Any third node involved in the transaction will receive a reissued invalidation or forwarded request again and react like it did when it received the message for the first time (if it was not lost). In particular, the owner node may receive a reissued forwarded request and, if it had received the first forwarded request, it must detect that it is a reissued request and send the data again using its backup copy. Depending on which messages were lost, one or several responses (*Data*, *DataEx*, *Ack*, etc) will arrive several times to the requestor who will discard those that do not carry the serial number of the last request.
- If no message had actually been lost, all the involved nodes will receive reissued requests, invalidation or forwarded requests and all responses will be reissued. Only the responses to the last request will be considered by the requestor.

As mentioned before, the L2 and the memory controller need to be able to detect reissued requests and replace the original request in the MSHR (assuming it was not lost) with the new one. The L2 or memory controller 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 memory 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 information can be stored in the MSHR or in a small associative structure. This way, if a *DataEx* response is lost, it will be detected using the *lost request timeout* and corrected by resending the request.

Serial numbers are also used to be able to discard late unblock messages, late write-back messages or late backup deletion acknowledgments. These duplicated messages can appear due to unnecessary *UnblockPing* or *WbPing* messages sent in the case of false positives of the *lost unblock timeout*.

As mentioned in section 5.2, the *ownership acknowledgment* (*AckO*) can be seen as a request sent by a node that has just received the data and acquired the ownership of a memory line to the former owner node requesting it to discard its backup copy of the data. This request may need to be reissued if the *lost backup deletion acknowledgment timeout* triggers (see section 5.2).

When an ownership acknowledgment is reissued after the *lost backup deletion acknowledgment timeout* triggers, the original ownership acknowledgment may or may not have been lost.

If the first 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 with the new serial number will be returned.

On the other hand, if the first ownership acknowledgment arrived 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.

Not discarding *backup deletion acknowledgments* with unexpected serial numbers would cause problems if an old *backup deletion acknowledgment* arrives to a new owner and the *ownership acknowledgment* which has just been sent by it is lost, since then the new owner would abandon the blocked ownership state while

the backup has not been discarded, making possible to have two backup copies of the same memory line. However, this situation is very unlikely.

OwnershipPing messages which are sent when the *lost data timeout* triggers (see section 5.2) can be seen as "request to confirm (or reject) ownership"[3] too. They also carry serial numbers to be able to discard late *NackO* messages.

Choosing serial numbers and serial number sizes

Serial numbers are used to discard duplicated responses to reissued requests[4]. They need to allow the requestor to differentiate among responses to the last attempt to reissue a request and responses to previous attempts to the same request.

This means that they need to be different for requests to the same address and by the same node but it does not matter if the same serial number is used for requests to different addresses or by different requestors.

Ideally, we should chose serial numbers in a way that minimizes the probability of picking the same serial number for two requests from the same node for the same address, since otherwise, if the first request was reissued at least once, it could be possible to accept a late response to the first request as a response to the second one. However, in practice two requests from the same node for the same address are usually spaced enough in time to make this possibility very remote, except in the case of a request for write access shortly after a request for read access (an upgrade).

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

On the other hand, the number of available serial numbers is finite. In our implementation, we use a small number of bits to encode the serial number in messages and MSHRs to minimize the overhead. This means that, if a request is reissued enough times, it will have to eventually reuse a serial number.

On the other hand, when reissuing a request, it is desirable to minimize the chances of using the serial number of any response to the former request still traveling through the interconnection network. For this, serial numbers for reissued requests are chosen sequentially increasing the serial number of

[3]More precisely, it is a request to confirm whether the node has ownership of the memory line, is currently requesting ownership, or neither.

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

the previous attempt (wrapping to 0 if necessary). This way, when using n bits to encode the serial number, we would have to reissue the same request 2^n times before having any possibility of receiving a response to an old request and accepting it, which could cause problems in some situations as shown in figure 5.6.

As the fault rate increases, requests will have to be reissued more frequently, so it will be necessary to use more bits to encode request serial numbers and ensure that late responses are always discarded. Hence, the number of bits used for serial numbers will limit the ability of the protocol to work correctly with higher fault rates.

Ping messages

Some faults cannot be solved by means of reissuing a request as described in section 5.2. This is the case when the fault is detected by a responding node and not by the requestor, as happens when either the *lost unblock timeout* or the *lost data timeout* trigger.

In the case of the *lost unblock timeout*, an *UnblockPing* or *WbPing* is sent when it triggers (see section 5.2). *UnblockPing* messages are sent for *GetS* and *GetX* requests and include a bit indicating whether the pending request is a read or a write access request.

When an L1 cache receives an *UnblockPing* message from L2 it will either answer with an unblock message or ignore the ping, depending on whether it has a pending request for that address of the corresponding type (either read or write access request).

If the cache has already satisfied that miss (hence it has already sent a corresponding unblock message which may have been lost or not), it will not find a corresponding request. It will assume that the unblock message got lost and will answer with a reissued *Unblock* or *UnblockEx* message with the same serial number as the received ping, depending on whether it has exclusive or shared access to the line[5].

If the L1 cache does have a pending miss of the correct type for the involved

[5] As a consequence of some very uncommon races, it may be the case that the cache does not only not have a matching pending miss, it may not even have the data (it would be in an stable I state). It is safe if it answers with an *Unblock* message even in those cases: this will make the directory information less exact because the directory will add the node as a sharer, but this is not a problem because the protocol will work correctly as long as the set of sharers recorded in the directory is a non-strict superset of the real set of sharers.

address, it will assume that the *UnblockPing* refers to that miss[6]. If the miss has not been resolved yet, no unblock message could have been lost because it was not sent in the first place and, hence, the *UnblockPing* message will be ignored. Sooner or later either the response will arrive to the L1 and then it will send the unblock message or, if the response was lost, the *lost request timeout* will trigger.

The *UnblockPing* message can carry the same request serial number as the expected unblock message. This way, if the message was not actually lost the line will be unblocked as soon as it arrives and only two extra control messages will travel through the network (the ping message and its answer, which will be discarded once it arrives to L2).

On the other hand, the *lost unblock timeout* can also trigger for a *Put* request. In that case, the L2 will assume that the write-back message has been lost and will send a *WbPing* message to the L1.

The L1 will answer with a new write-back message (in case it still has the data) or a *WbCancel* message which tells the L2 that the write-back has already been performed and the data (which was previously in shared state) has been discarded. Note that modified data cannot be lost thanks to the rules described in section 5.2, although clean data may be lost from the chip and would have to be requested to memory again.

Finally, pings are also used when the *lost data timeout* triggers. This timeout triggers during an ownership transference when the former owner detects that the owned data may have been discarded because the receiver did not expect it. When this timeout triggers, an *OwnershipPing* message is sent to the receiver, with a newly chosen serial number.

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 of the line in a blocked ownership state (i.e., it already sent an ownership acknowledgment and is waiting for the corresponding backup deletion acknowledgment) it will reissue the

[6]Although it may not necessarily be the case, there are race conditions which may have invalidated the data acquired with the request related with the *UnblockPing* message and then a new request may have been issued. But if this is the case, the new request will eventually arrive to L2, which will treat it as a reissue of the previous one.

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. This can happen if an *OwnershipPing* message gets delayed in the network.

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 actually 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.

Taking advantage of a point-to-point ordered network

Until now, we have assumed in this work that our protocols use an unordered point-to-point interconnection network, which is the most general kind of network from the point of view of the coherence protocol since it does not provide any guarantee regarding the order in which messages arrive to their destinations. In particular, it provides no guarantee about the relative order in which two messages sent from the same source node to the same destination node will actually arrive.

This freedom at the network level makes possible to implement adaptive routing which can be useful for fault tolerance and for other purposes.

However, this kind of networks, although useful in some situations, is not always used. Most times, the network does provide at least ordering guarantee between each pair of nodes. That is, two messages sent from a node to another will arrive in the same order that they were sent.

Assuming that messages between two nodes arrive in the same order that they were sent significantly reduces the number of race conditions that can happen and allows for some simplifications for both DirCMP and FtDirCMP. Most of these simplifications are common to both protocols and are not discussed here.

In FtDirCMP, one simplification in particular has significant impact in the hardware overhead of fault tolerance. When point-to-point order is guaranteed, the *lost data timeout*, the *OwnershipPing* and *NackO* message types become unnecessary, since owned data would never be sent to a node that does not expect it (hence, it will never be discarded unless the request had already been reissued). That is, the cases described in section 5.2 become impossible. This way, the *lost request timeout* would be able to detect all cases of lost owned data.

5.3 Hardware overhead of FtDirCMP

FtDirCMP fault tolerance measures add some hardware overhead with respect to DirCMP. Chapter 8 presents an evaluation of this overhead in terms of performance, while in this section we enumerate the main hardware overheads incurred by an implementation of FtDirCMP in comparison with an implementation of DirCMP.

The main hardware overheads in FtDirCMP are due to the timeouts used to detect faults, the need for the requestor to store the request serial numbers of the current request to be able to discard late responses, and the need of the responder to store the identity of the requestor to be able to detect reissued requests.

Differently to FtTokenCMP, FtDirCMP cannot take advantage of preexisting hardware to implement the timeouts used for fault detection. Hence, some counters have to be added to implement the different timeouts. These counters could be added to the MSHRs or using a separate pool of timeout counters for cases where no MSHR would need to be allocated otherwise. Notice that, although there are up to four different timeouts involved in a coherence transaction, no more than one counter is required at any time in the same node for a single coherence transaction.

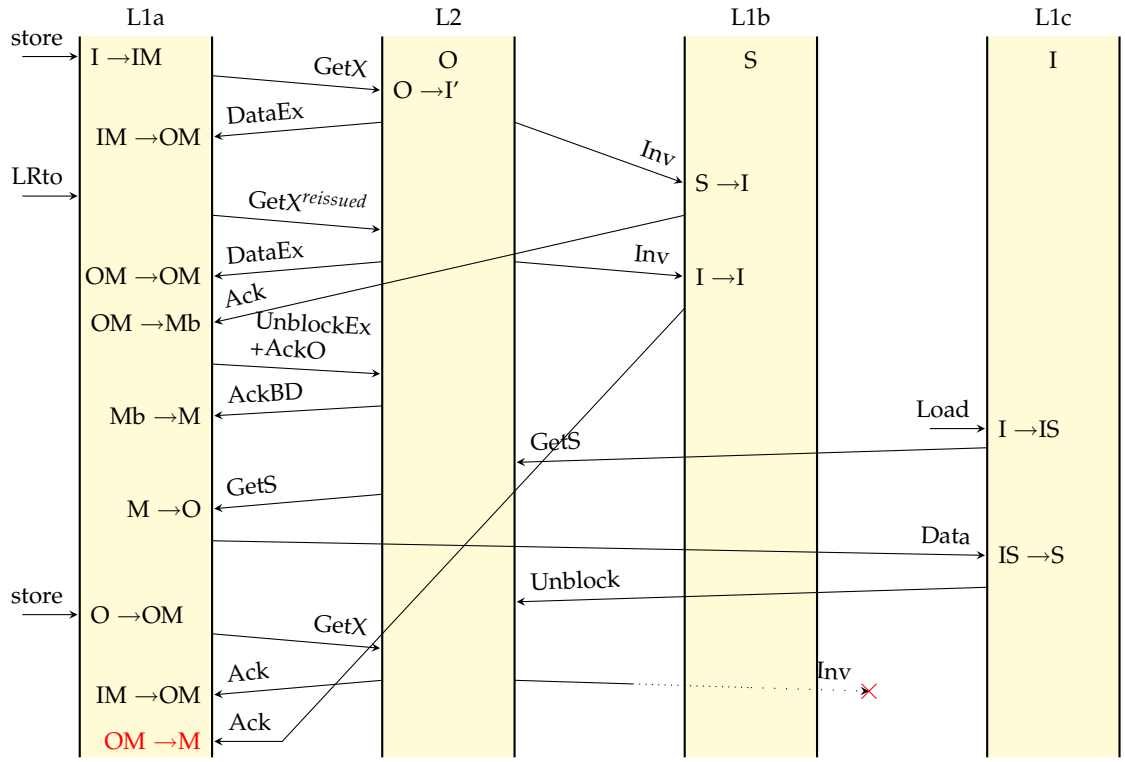
Request serial numbers need to be stored by the requestor to be able to discard late responses. This requires a few extra bits per entry in each MSHR. The particular number of bits that need to be added depends on the rate of faults that needs to be supported by the protocol, as it is evaluated in section 8.3. Request serial numbers need to be remembered in some situations where usually no MSHR would be allocated, like when an *OwnershipPing* is sent. For those cases, the protocol would need to allocate an MSHR or, alternatively, a smaller pool of registers could be provided for this purpose.

The identity of the requester currently being serviced needs to be stored by the L2 cache and the memory controller to be able to detect reissued requests of the request being serviced and to be able to attend them immediately. This information needs to be stored anyway in most cases in non fault-tolerant protocols like DirCMP, but FtDirCMP additionally requires that any L1 cache stores the identity of the receiver of owned data when transferring ownership to make possible to detect reissued forwarded requests and be able to resent the data which is being kept in backup state.

A less obvious hardware requirement added by the fault tolerance measures of FtDirCMP is that the number of virtual networks provided by the interconnection network to avoid deadlock may need to be increased, depending on the

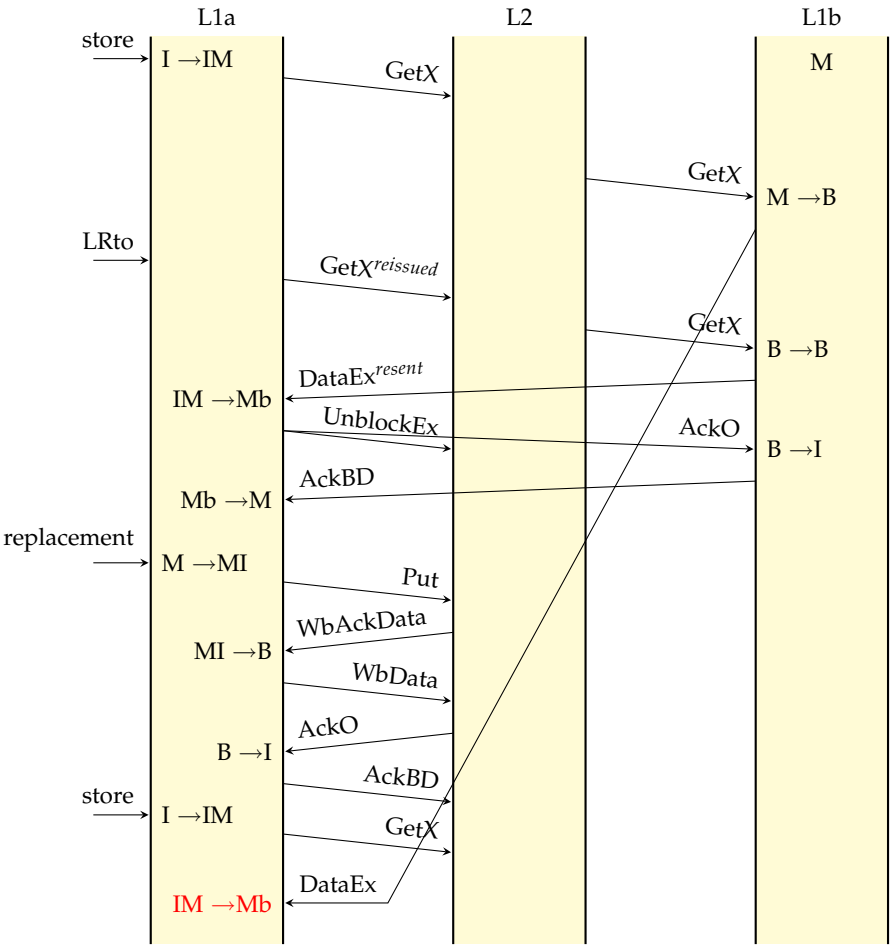
particular network topology and deadlock avoidance strategy which is used. The dependency chanins that we have considered for DirCMP and FtDirCMP are shown in appendix C.

A less important source of overhead is the increased pressure in caches and write-back buffers due to the blocked ownership and backup states. When a write-back buffer is used, we have not been able to detect any effect in the execution time due to these reasons. The size of the write-back buffer may need to be increased, but as mentioned in section 4.2 for the case of the FtTokenCMP, one extra entry would be enough to avoid any slowdown.



Initially, the data is at L2 in O state and at L1b in S state. L1a makes a write request to L2 which sends an invalidation to L1b and a *DataEx* message to L1a. The *DataEx* also tells L1a that it needs to wait for one invalidation acknowledgment before entering the M state. When L1b receives the invalidation, it sends an acknowledgment to L1a. Due to network congestion, this message takes a long time to arrive to L1a, so the *lost request timeout* triggers and L1a reissues the write request. The reissued request arrives to L2, which resends the data and the invalidation. When the invalidation arrives to L1b, it resends an acknowledgment to L1a. Let's assume that the first acknowledgment arrives to L1a before the second one, as in a point-to-point ordered network. If request serial numbers are not used, it will be accepted (otherwise it would be discarded and L1a would wait for the second one) and an *UnblockEx+AckO* message will be sent to L2 which will answer with an *AckBD*. Now, due to the stale acknowledgment traveling through the network, the system can arrive to an incoherent state: another cache L1c issues a read request which is forwarded by L2 to L1a. L1a answers it and transitions to O state. L1c receives it, sends an unblock and transitions to S state. Next, L1a issues a new write request to L2 which sends an invalidation to L1c and an acknowledgment to L1a which tells it that it needs to wait for one invalidation acknowledgment. The invalidation request gets lost due to corruption. When the stale acknowledgment from the first transaction arrives now to L1a, it will assume that it can transition to M state despite the fact that L1c is still in S state, thus violating coherency.

Figure 5.5: Transaction where request serial numbers are needed to avoid incoherency in a point-to-point ordered network



L1a makes a request to L2 which forwards it to L1b. L1b sends the data 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 which is forwarded again to L1b which has a backup copy of the data and resends it. This time, the message arrives to L1a which sends an unblock message to L2 and an ownership acknowledgment to L1b. L1b answers with a backup deletion acknowledgment to L1a. After modifying the data, L1a performs a write-back to L2 and after that, it issues another request. When the first data message arrives at this moment and is not discarded using its serial number, it would allow L1a to use the old data.

Figure 5.6: Transaction where request serial numbers avoid using stale data in a point-to-point unordered network

Chapter 6

A broadcast-based fault-tolerant cache coherence protocol

We have presented two fault-tolerant cache coherence protocols in the previous two chapters: one based in token coherence and another based in directory coherence. We think that these two approaches to the design of cache coherence protocols are the most reasonable and the most likely candidates to be used in future many-core CMPs with point-to-point unordered networks and generic topology. Other attractive approaches to coherence impose additional constraints on the interconnection networks, like ring based coherence [71].

However, to date no real system has been implemented using a cache coherence protocol based on the token coherence framework. Snoopy-based and directory-based protocols have been used in several systems, but many of the cache coherence protocols which are used in widespread systems cannot be precisely categorized as snoopy-based nor directory-based. There are several cache coherence protocols which have been designed ad-hoc considering a set of requirements and priorities that did not coincide with either snoopy or directory protocols[1]. The protocol used by systems built using AMD Opteron processors [5, 54, 93] is one of these protocols.

In this chapter, we will show how the techniques developed for FtTokenCMP and FtDirCMP can be applied to other more specific protocols like AMD's Hammer protocol.

[1]Often, the most important requirement was to keep backward compatibility with supporting hardware for a previous chip or to adhere to a particular specification.

6.1 A cache coherence protocol for CMPs based on AMD's Hammer protocol

In this section we describe the protocol used by AMD in their Opteron systems. It targets systems with a small number of processors using a tightly-coupled point-to-point interconnection network. We will call it Hammer which is the name of the design project that brought Opteron [54]. It was designed for systems which use HyperTransport for the interconnection network among chips. Besides being used in a recent and widespread system, the Hammer protocol is interesting because it presents a mixture of properties from both snoopy-based and directory-based protocols. Our understanding of Hammer is based on the information published by AMD [54, 93] and in the reverse engineering of the protocol performed by the Multifacet Project of the University of Winsconsin [67].

Since it targets a system with a point-to-point interconnection network, there is no shared bus that can be used by the protocol for serialization like in snoopy protocols. Hammer uses the home node of each memory line as the serialization point, similarly to directory-based protocols.

On the other hand, Hammer relies on frequent broadcasts, like snoopy-based protocols. This is acceptable because the number of total nodes of the target systems is small enough, and in this way the protocol can spare the memory overhead of directory information and the extra miss latency due to the access to that information.

Strictly speaking, this protocol could be classified as a directory-based protocol without directory information, also known as Dir_0B [2].

The advantage of Hammer with respect to a snoopy-based protocol is that it does not rely on a totally ordered interconnection network. The advantage with respect to a directory-based protocol is that it does not need to store any directory information, hence reducing the memory overhead and the latency due to accessing to this information.

Hammer avoids the overhead of directory information and the latency of accessing the directory structure at the cost of much more interconnection network traffic. While it avoids the latency of accessing to the directory information, it still needs to send all requests to the home node for serialization. Also, the cache controllers of all processors need to intervene in all misses, like in a snoopy protocol.

The original Hammer protocol was designed to be used in multiprocessors built with several chips, not in CMPs. We have implemented HammerCMP,

which is an adaptation of AMD's Hammer protocol to the tiled CMP environment described in section 3.1, and we have used it as a base for FtHammerCMP, which is a new fault-tolerant protocol that could be used for small scale CMPs. HammerCMP assumes that the network is not totally ordered but point-to-point ordered.

Differently from Hammer, HammerCMP has been optimized by including at each home tile a copy of the tag of the blocks that are held in the private L1 caches. In this way, the home knows whether the owner of the block is one of the caches on-chip (the L2 cache or one of the L1 caches) or is off-chip. These tags avoid both off-chip accesses when the owner block is on-chip (thus reducing both latency and memory bandwidth consumption) and broadcasting requests for blocks that are not stored on-chip (thus saving network traffic).

Like DirCMP, HammerCMP sends requests to a home L2 bank which acts as the serialization point for requests to its cache lines. There is no directory information, and all requests are forwarded using broadcast to all other caches. The L2 only has enough information to know which memory lines are on-chip (present either in at least one L1 cache or one L2 cache bank).

In a typical miss, a requester sends a message to the home L2 bank. If the memory line is present on-chip, the L2 always forwards the request to all other nodes and sends the data if it has it. Otherwise, the data is requested to the memory controller.

All processors answer to the forwarded requests sending either an acknowledgment or a data message to the requestor. The requestor needs to wait until it receives an acknowledgment from each other node. When the requestor receives all the acknowledgments, it informs to the home L2 controller that the miss has been satisfied with an unblock message.

For replacements, HammerCMP uses three-phase write-backs like DirCMP.

Problems caused by an unreliable interconnection network in HammerCMP

HammerCMP behaves very similarly to DirCMP in presence of faults. Like in DirCMP, every lost message will lead to a deadlock and data can only be lost if a data carrying message is corrupted while ownership of the memory line is being transferred from one node to a different one.

The only difference is due to the different properties of the assumed interconnection network. While DirCMP does not assume any ordering guarantees, HammerCMP assumes at least point-to-point ordering. Hence, a (very infre-

quent) class of errors than may happen in DirCMP (see section 5.2) become impossible in HammerCMP. As a result, FtHammerCMP requires one timeout less than FtDirCMP for fault detection.

6.2 Fault tolerance measures introduced by FtHammerCMP

Using the principles described in section 3.2, FtHammerCMP adds fault tolerance measures to HammerCMP as the ones described for FtDirCMP.

FtHammerCMP shares many characteristics with FtDirCMP and FtTokenCMP. Like the two previous cache coherence protocols, it assumes that every message either arrives correctly to its intended destination or is discarded due to corruption detected using error detection codes. It also protects critical data with a pair of acknowledgments out of the critical path and uses a number of timeouts for fault detection.

Since, as mentioned before, HammerCMP can be described as a directory-based protocol without directory information (Dir_0B), FtHammerCMP shares most features with FtDirCMP. However, HammerCMP and FtHammerCMP are somehow simpler than DirCMP and FtDirCMP respectively.

FtHammerCMP uses the same pairs of acknowledgments as FtDirCMP for reliable transference of owned data, just as described in section 5.2. FtHammerCMP piggybacks ownership acknowledgments in unblock messages too.

Every message lost in HammerCMP leads to a deadlock, so fault detection can be accomplished using timeouts. The set of timeouts used by FtHammerCMP is a subset of those used by FtDirCMP. Table 6.1 shows a summary of each of the timeouts used. FtHammerCMP does not require the *lost data timeout* (see section 5.2) because it assumes a point-to-point ordered interconnection network, hence owned data messages cannot be received by a node that does not expect them.

The recovery mechanisms are like in FtDirCMP: faults detected by the requestor are recovered by means of reissuing the request and faults detected by the responder (the L2 bank or memory controller) are recovered using ping messages.

FtHammerCMP reissues requests when the *lost request timeout* triggers and reissues *AckO* messages when the *lost backup deletion timeout* triggers. It also uses *request serial numbers* for discarding old responses to reissued requests and to avoid creating incoherences (see section 5.2).

Table 6.1: Summary of timeouts used in FtHammerCMP

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

Summary of differences of the fault tolerance measures of FtHammerCMP and FtDirCMP

As seen above, the fault tolerance measures of FtHammerCMP are very similar to those of FtDirCMP. The main differences are:

- Due to assuming an ordered point-to-point network no owned data message can be sent to a node that does not expect it, hence no *lost data timeout* is required (see section 5.2).
- FtHammerCMP does not have *OwnershipPing* messages nor *NackO* messages.

It is also worth mentioning that since HammerCMP is simpler and uses less hardware to keep coherence than DirCMP, the overhead of the fault tolerance measures of FtHammerCMP is higher than in the case of FtDirCMP in relative terms. This is true for both the overhead in terms of protocol complexity and the overhead in terms of hardware.

Chapter 7

Evaluation methodology

In this chapter we explain the experimental methodology that we have used to evaluate the fault-tolerant protocols proposed in this thesis. The main aims of our evaluation are:

- To ensure that the fault tolerance measures of our protocols actually work. That is, programs running in a tiled CMP system using our protocols terminate and produce correct results even if not all messages are correctly delivered by the on-chip interconnection network.
- To measure the overhead introduced by the fault tolerance features of the cache coherence protocols when no faults occur. We focus on the execution time overhead and the network traffic overhead. We expect that since the amount of extra hardware required to implement the proposed fault tolerance measures is small, so will be the power overhead due to that hardware. Hence, we expect the power overhead of the fault tolerance measures to be caused mainly by the extra network traffic.
- To measure the performance degradation of programs running in a system using our fault-tolerant protocols when the interconnection network is subject to several fault rates.
- To determine appropriate values for the configuration parameters provided by our protocols, like the value of the fault-detection timeouts.

The methodology chosen for this evaluation is based on full-system simulation. We have prototyped all the evaluated cache coherence protocols in a modified

version of Multifacet GEMS [69] from the University of Winsconsin and we have used the resulting simulation environment to execute a number of parallel applications.

GEMS is a simulation environment which is based on Virtutech Simics [65]. Simics is a functional full-system simulator capable of simulating several types of hardware including multiprocessor systems. At the implementation level, GEMS is a set of modules that plug into Simics and add timing abilities to the simulator.

GEMS provides several modules for modelling different aspects of the architecture. For example, *Opal* models an out-of-order processor core, *Ruby* models a detailed memory hierarchy and *Tourmaline* is a functional transactional memory simulator. For the evaluation presented in this thesis, we only use Ruby.

Ruby provides an event-driven framework to simulate a memory hierarchy with sufficient detail to be able to measure the effects of changes to the coherence protocols. One of the key pieces of Ruby is SLICC (Specification Language for Implementing Cache Coherence): a domain-specific language to specify cache coherence protocols which has been used to implement the cache controllers for the protocols discussed in this thesis.

The memory model provided by Ruby is made of a number of components that model the L1 and L2 caches, memory controllers and directory controllers. These components model the timing calculating the delay since a request is received until a response is generated and injected into the network. All the components are connected using a simple network model that calculates the delay required to deliver a message from one component to another.

The models of the L1 caches interact with models of processors. For this thesis, we use a simple in-order processor model provided by Simics that assumes that all instructions execute in a single cycle except memory accesses (which are simulated by Ruby). Since we are simulating a 16-way CMP, we think that assuming in-order cores is the most realistic choice. We have performed some simulations using out-of-order cores in a previous work [37] and we have found that the performance overhead of the fault-tolerant protocol is similar with both in-order and out-of-order cores. In any case, the correctness of the fault tolerance measures of our protocols is unaffected if out-of-order processors are used.

The interconnection network model provided by GEMS is very idealized. The network is specified as a graph connecting the cache controllers and internal switches. Each link has a fixed latency and bandwidth parameters. These parameters are used to calculate the latency incurred by each message taking into acount any queueing delay due to insufficient bandwidth. With this interconnection

network model, to inject faults we just randomly discard as many messages as necessary depending on the desired fault rate (see section 7.1).

Since we are simulating multithreaded workloads, the results of our simulations are not deterministic. The order in which threads arrive to synchronization points and their interactions in general can cause some variability not only in the final execution time but also in the execution path due to, for example, different operating system scheduler decisions. To account for this variability we add non-determinism to our simulator by slightly modifying the memory access time at random[1]. Then, we execute the same simulation several times using different random seeds and calculate the 95% confidence interval for our results, which is shown in our plots with error bars. Each data point is the result of at least 6 simulations, or more in the case of applications that show more variability.

When simulating, we only measure the parallel parts of the applications. Our benchmark checkpoints have been prepared so that the application is already running and loaded in memory to avoid page faults during the measurement. That is, we perform memory warm up before saving the checkpoint. Also, to avoid cold cache effects, we run our simulations with full detail for a significant time before starting the actual measurements. That is, we warm up the cache before each simulation.

We have implemented the proposed fault-tolerant coherence protocols using the detailed memory model provided by Ruby. The base non fault-tolerant protocols used were provided with the GEMS distribution and had already been used in published works [70], except the HammerCMP protocol which has been developed from scratch.

We have also performed an extensive functional validation of the fault-tolerant protocols using randomized testing. This randomized testing stresses protocol corner cases by issuing requests that simulate very contended accesses to a few memory lines and using random latencies for message delivery. The tester also issues many concurrent requests, like a very aggressive out-of-order processor would do. To ensure that the fault-tolerant measures actually work, the tester also performs fault injection with very high fault rates.

7.1 Failure model

As mentioned in section 1.3, a transient fault in the interconnection network of a CMP can have a number of causes. For example, any event that changes

[1]The latency varies up to two cycles plus or minus with uniform probability.

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.

However, 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 failure model we assume that the interconnection network will either deliver a message correctly or not at all. This can be achieved by means of using an error detection code (EDC) in each message and discarding corrupted messages upon arrival. We have not specified the particular error detection code employed for this purpose. Nodes should also discard misrouted messages.

We also assume that caches and memories are protected by means of ECC, so that data corruption can happen only when data is traveling through the interconnection network.

Transient faults are usually modeled with a bit-error rate (BER, ratio of bits incorrectly received to total bits transmitted). It is quite difficult to get the precise frequency of soft errors as it depends on many factors, including the location in which the system is placed. However, a study of literature about fault tolerance reveals that it usually remains in the range of 10^{-9} and 10^{-20} [23].

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. The first case can be considered a particular case of the second one with bursts of one message in length.

When performing fault injection of bursts of length L, we randomly determine for each message whether it has been corrupted or not on arrival 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 or even if there are no bursts. See section 8.2 for an evaluation of the effect of bursts of transient faults.

7.2 Configuration of the simulated systems

We have simulated a tiled CMP as described in section 3.1. Table 7.1 shows the most relevant configuration parameters of the modeled system.

In particular, the values chosen for the fault-detection timeouts have been fixed experimentally to minimize the performance degradation in presence of faults while avoiding false positives which would reduce performance in the fault-free case (see section 8.3). Using even shorter timeout values would reduce the performance degradation in presence of faults only moderately, but would significantly increase the risk of false positives.

Using out-of-order execution does not affect the correctness of the protocol at all and does not have an important effect in the overhead introduced by the fault tolerance measures compared to the non fault-tolerant protocol, as can be seen in our previous work [37]. For brevity, we do not include results for out-of-order processors in this evaluation.

7.3 Applications

Our results have been obtained by simulating a mix of scientific, multimedia and server applications. *Apache* and *SpecJbb* are well known server applications. *Barnes*, *FFT*, *LU*, *Ocean*, *Radix*, *Raytrace* and *WaterNSQ* are from the SPLASH-2 benchmark suite [136]. *Em3d* models the propagation of electromagnetic waves through objects in three dimensions and *Unstructured* is a computational fluid dynamics application. And *FaceRec*, *MPGdec*, *MPGenc* and *SpeechRec* are multimedia applications from the ALPBench benchmark suite [63].

Table 7.2 shows the input sizes used in the simulations. Due to the long simulation times required for full-system simulation, we are constrained to scaled down problem sizes for many applications. For others, the execution has been divided in arbitrary units of work that we call *transactions*, and we measure the execution of a fixed number of such transactions. More details about each application are given in the next section.

Server applications

Apache

This is a static web serving benchmark using Apache version 2.2.4. Requests are made by a Surge client running on the same machine, simulating 500 clients

Table 7.1: Characteristics of simulated architectures

16-Way Tiled CMP System	
Processor parameters	
Processor frequency	2 GHz
Cache parameters	
Cache line size	64 bytes
L1 cache:	
Size	32 KB
Associativity	4 ways
Hit time	3 cycles
Shared L2 cache:	
Size	1024 KB
Associativity	4 ways
Hit time (same tile)	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) Parameters common to all configurations

FtTokenCMP specific parameters	
Lost token timeout	2000 cycles
Lost data timeout	2000 cycles
Lost backup deletion acknowledgment timeout	2000 cycles
Lost persistent deactivation timeout	2000 cycles
Token serial number size	2 bits
Token serial number table size	16 entries
Backup buffer size	2 entries
FtDirCMP specific parameters	
Lost request timeout	1500 cycles
Lost unblock timeout	1500 cycles
Lost backup deletion acknowledgment timeout	1500 cycles
Lost data timeout	1500 cycles
Request serial number bits per message	8 bits
FtHammerCMP specific parameters	
Lost request timeout	1500 cycles
Lost unblock timeout	1500 cycles
Lost backup deletion acknowledgment timeout	1500 cycles
Request serial number bits per message	8 bits

(b) Parameters specific for each fault-tolerant cache coherence protocol

Table 7.2: Benchmarks and input sizes used in the simulations

Benchmark	Input Size
	Server benchmarks
Apache	10 000 HTTP transactions
SpecJbb	8 000 transactions
	SPLASH-2 benchmarks
Barnes	8 192 bodies, 4 time steps
FFT	64K complex doubles
LU	512×512 matrix
Ocean	258×258 ocean
Radix	1M keys, 1024 radix
Raytrace	10MB, teapot.env scene
WaterNSQ	512 molecules, 4 time steps
	Other scientific benchmarks
Em3d	38 400 nodes, degree 2, 15% remote and 25 time steps
Unstructured	Mesh.2K, 5 time steps
	ALPBench benchmarks
FaceRec	ALPBench training input
MPGdec	525_tens_040.mv2
MPGenc	MPGdec output
SpeechRec	ALPBench default input

making random requests to 2 000 different HTML files. Each client waits 10ms between requests (think time). The server uses 16 processes and 25 threads per process. It also has all logging functionality disabled.

This benchmark is divided in transactions, where each transaction is an HTTP request. We simulate 100 000 transactions without the detailed timing model to warm up main memory, 500 transactions with the timing model enabled to warm up the caches and 10 000 transactions to obtain results.

SpecJbb

SpecJbb is based on SPEC JBB2000. It is a Java based server workload that emulates a 3-tier system. In this benchmark, the work of the middle tier predominates, which is the business logic and object manipulation. We use Sun's HotSpot Java virtual machine version 1.5.0 for Solaris. Our benchmark uses 24 warehouses.

This benchmark is divided in transactions. We simulate 1 000 000 transactions without the detailed timing model to warm up main memory, 300 transactions with the timing model enabled to warm up the caches and 8 000 transactions to obtain the results.

Applications from SPLASH-2

Barnes

The *Barnes* application simulates the interaction of a system of bodies (galaxies or particles, for example) in three dimensions over a number of time steps, using the Barnes-Hut hierarchical N-body method. Each body is modelled as a point mass and exerts forces on all other bodies in the system. To speed up the interbody force calculations, groups of bodies that are sufficiently far away are abstracted as point masses. In order to facilitate this clustering, physical space is divided recursively, forming an octree. Bodies are assigned to processors at the beginning of each time step in a partitioning phase. Each processor calculates the forces exerted on its own subset of bodies. The tree representation of space has to be traversed once for each body and rebuilt after each time step to account for the movement of bodies.

There are several barriers for separating different phases of the computation and successive time steps. Some phases require exclusive access to tree cells and a set of distributed locks is used for this purpose. The communication patterns are dependent on the particle distribution and are very irregular.

FFT

The *FFT* kernel implements a radix-$\sqrt{n}$ six-step Fast Fourier Transform algorithm as described in [9], which is optimized to minimize interprocessor communication. The data set consists of n complex data points to be transformed, and another n complex data points referred to as the *roots of unity*. Both sets of data are organized as $\sqrt{n} \times \sqrt{n}$ matrices partitioned so that every processor is assigned a contiguous set of rows. All-to-all interprocessor communication occurs in three matrix transpose steps. Synchronization in this application is accomplished by using barriers.

LU

The *LU* kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The algorithm uses blocking techniques and blocks are assigned to processors using a 2D scatter decomposition and each block is updated by the processor that owns it. Synchronization is done using barriers.

Ocean

The *Ocean* application studies large-scale ocean movements based on eddy and boundary currents. The algorithm simulates a cuboidal basin using discretized circulation model that takes into account wind stress from atmospheric effects and the friction with ocean floor and walls. The algorithm performs the simulation for many time steps until the eddies and mean ocean flow attain a mutual balance. The work performed every time step involves setting up and solving a set of spatial partial differential equations. For this purpose, the algorithm discretizes the continuous functions by second-order finite-differencing, sets up the resulting difference equations on two-dimensional fixed-size grids representing horizontal cross-sections of the ocean basin, and solves these equations using a red-back Gauss-Seidel multigrid equation solver. Each task performs the computational steps on the section of the grids that it owns, regularly communicating with other processes. Synchronization is done using barriers and locks.

Radix

The *Radix* kernel sorts a series of integers using the radix sorting method. Each processor is assigned a subset of the numbers. In each iteration, a processor passes over its assigned keys and generates a local histogram. These local histograms

are then accumulated into a global histogram which is used by each processor to permute its keys into a new global array for the next iteration. This permutation step requires all-to-all communication. Synchronization in this application is accomplished by using barriers.

Raytrace

This application renders a three-dimensional scene using ray tracing. A ray is traced through each pixel in the image plane and it produces other rays as it strikes the objects of the scene, resulting in a tree of rays per pixel. The image is partitioned among processors in contiguous blocks of pixel groups, and distributed task queues are used with task stealing. The data accesses are highly unpredictable in this application.

Synchronization in *Raytrace* is done using locks. This benchmark is characterised by having very short critical sections and very high contention.

WaterNSQ

WaterNSQ performs an N-body molecular dynamics simulation of the forces and potentials over time in a system of water molecules. It is used to predict some of the physical properties of water in liquid state.

Molecules are statically split among the processors. At each time step, the processors calculate the interaction of the atoms within each molecule and the interaction of the molecules with one another. Most synchronization is done using barriers, although there are also several variables holding global properties that are updated continuously and are protected using locks.

Applications from ALTPBench

FaceRec

FaceRec is a benchmark based on the Colorado State University face recognizer. Face recognition can be used for authentication, security and screening. similar algorithms can be used in other image recognition applications. The ALPBench version has been modified to compare a separate input image with all the images contained in a database. The application has an offline training phase, but only the recognition phase is considered in our evaluation.

MPGdec

The *MPGdec* benchmark is based on the MSSG MPEG decoder. It decompresses a compressed MPEG-2 bit-stream. Many recent video decoders use similar algorithms. The execution comprises four phases: variable length decoding, inverse quantization, inverse discrete cosine transform (IDCT) and motion compensation.

In this application threads are created and finished in a staggered fashion as contiguous rows of blocks are identified by the main thread. This affects scalability.

We have divided this benchmark in transactions, where each transaction is the decoding of one video frame.

MPGenc

This benchmark is based on the MSSG MPEG-2 encoder. It converts video frames into a compressed bit-stream. The ALPBench version has been modified to use an intelligent three-step motion search algorithm instead of the original exhaustive search algorithm and to use a fast integer discrete cosine transform (DCT) butterfly algorithm instead of the original floating point matrix based DCT. Also, the rate control logic has been removed to avoid a serial bottleneck. This application is divided in the same phases as *MPGdec*, but they are performed in the reverse order.

We have divided this benchmark in transactions, where each transaction is the encoding of one video frame.

MPGdec and *MPGenc* perform complementary functions. In fact, we use the output of *MPGdec* as the input of *MPGenc*.

SpeechRec

The *SpeechRec* benchmark uses the CMU SPHINX speech recognizer version 3.3. Speech recognizers are used with communication, authentication and word processing software and are expected to become a primary component of the human-computer interface.

The application has three phases: feature extraction, Gaussian scoring and searching in the language dictionary. The feature extraction phase is not parallelized. Thread barriers are used for synchronization between phases and fine-grain locking is used during the search phase.

Other scientific applications

Em3d

Em3d models the propagation of electromagnetic waves through objects in three dimensions. The problem is framed as a computation on a bipartite graph with directed edges from nodes, representing electric fields to nodes representing magnetic fields and conversely. The sharing patterns found in this application are static and repetitive.

Unstructured

Unstructured is a computational fluid dynamics application that uses an unstructured mesh to model a physical structure, such as an airplane wing or body. The mesh is represented by nodes, edges that connect two nodes, and faces that connect three or four nodes. The mesh is static, so its connectivity does not change. The mesh is partitioned spatially among different processors using a recursive coordinate bisection partitioner. The computation contains a series of loops that iterate over nodes, edges and faces. Most communication occurs along the edges and faces of the mesh.

Chapter 8

Evaluation results

In this chapter we show the results of the evaluation of the fault-tolerant cache coherence protocols that we have performed using the methodology described in the previous chapter. We have checked that the fault tolerance measures of the protocols perform correctly by means of randomized testing as mentioned in chapter 7, so the results presented in this chapter concern the overheads introduced by these measures in the fault-free case and the performance degradation due to faults when they occur.

The behavior of our protocols depends on a few parameters which determine the latency of fault-detection and the maximum fault rate that can be effectively supported by the protocols. We explain these parameters and the values chosen for them in section 8.3.

8.1 Overhead without faults

The fault tolerance measures introduced in our protocols try to minimize the overhead in terms of execution time or network traffic. Specially, we would like to have as little overhead as possible when no transient faults actually occur since that should be, by far, the most frequent case.

The protocols can have two main types of overhead: increased execution time and increased network traffic.

Execution time overhead

Adding fault tolerance measures to the cache coherence protocols may increase the execution time, as it is the case with most fault tolerance designs. In our case, the increased execution time, if any, would be due to several reasons, including:

Increased number of messages in the critical path: Since increasing the number of messages in the critical path of any type of miss would increase the latency of such misses, we have avoided adding messages in the critical path of most memory transactions in all our protocols. Unfortunately, there are situations where this has not been possible.

In particular, when writing data back from the L1 cache to the L2 cache or from the L2 cache to memory, an ownership acknowledgment has to be received before the data can be evicted from the cache. This message will be in the critical path of the cache miss that caused the replacement if no write-back buffer or backup buffer is used (see section 4.2 and figure 4.2).

FtDirCMP and FtHammerCMP use a write-back buffer (like DirCMP and HammerCMP), and FtTokenCMP has a two-entry backup buffer.

Increased cache pressure due to the storage of backups: Our protocols keep a backup copy of owned data while transferring ownership until an *ownership acknowledgment* is received. This backup copy is kept in the cache that loses the ownership, hence it increases cache pressure with respect to the non fault-tolerant protocols. Fortunately, this backup copy is kept during a very short period (the round-trip time between the cache that loses ownership and the cache that acquires it).

Increased latency due to the blocked ownership states: Our fault-tolerant protocols create a backup copy every time that ownership needs to be transferred. So, to avoid having more than one backup copy at a given time, the node that receives ownership of a block cannot transfer it again until it receives a backup deletion acknowledgment. During this *blocked ownership* period, the new owner cannot respond to requests which would require ownership transference. This may increase the latency of those requests.

Calculation of the error detection code: Our protocols rely on having error detection codes in the messages so that they can be discarded at reception when corrupted. We have not specified the error detection scheme employed for this purpose, hence we cannot measure its impact on execution

time either. In any case, we think that this should not have a very noticeable impact. The calculation of the error detection code can be pipelined with the reception or emission of the message, and the processing of the message can be started upon reception even before the error detection code has been checked because the message can be speculatively assumed to be correct (which, as mentioned, is expected to be the common case).

We have measured the execution time of our set of benchmarks with the three fault-tolerant protocols and the three non fault-tolerant protocols. The results can be seen in figure 8.1. Figure 8.1a shows the relative execution time of each protocol with respect to its non fault-tolerant counterpart, while figure 8.1b shows the relative execution times with respect to DirCMP to show the differences in execution time among the different families of protocols.

As can be seen in figure 8.1a, there is no statistically significant difference between each fault-tolerant protocol and its standard counterpart for most benchmarks. Some benchmarks with high variability in their results show small differences (less than 2%): *Raytrace* executes faster with the fault-tolerant protocols, while *Em3d* and *MPGenc* execute slower in some cases.

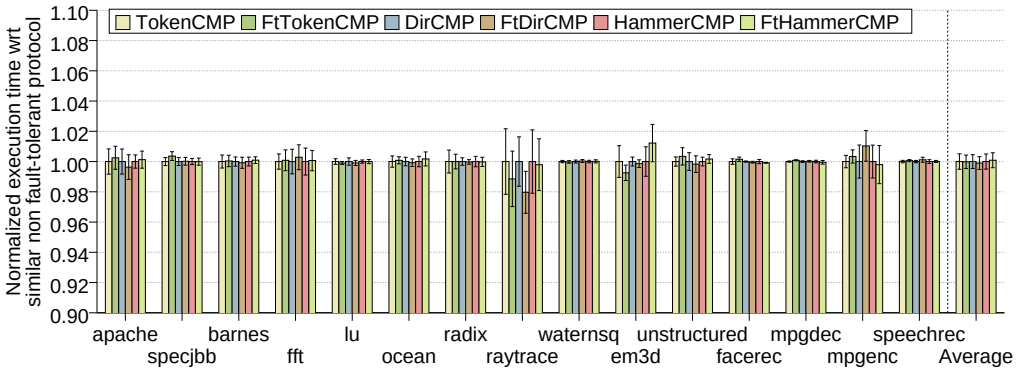
This means that we have not been able to measure any impact on average from any of the potential sources of execution time overhead that we mentioned above. We think that this result can be explained due to the following reasons:

- Although there are a few extra messages in the critical path of dirty write-backs, they are not harmful for performance because write-backs are out of the critical path of misses in our fault-tolerant protocols. This is the case also for DirCMP and HammerCMP since they use a write-back buffer[1]. It is not the case for TokenCMP, hence FtTokenCMP uses a two entry backup buffer (see section 4.2).

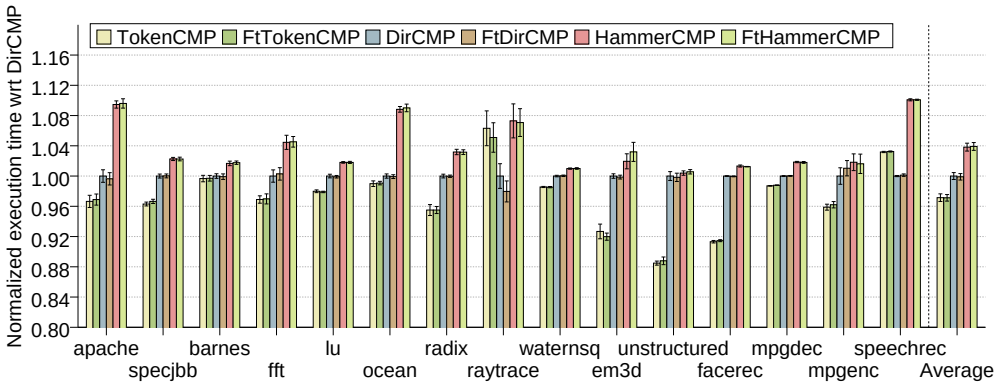
- The increased pressure due to the storage of backups in the cache is minimal because backups are kept only for very short periods of time. Moreover, in the case of FtTokenCMP the backup is moved to the backup buffer if the space is needed.

- The periods of blocked ownership are very short too (the round-trip time between the new owner node and the former owner node). Only those

[1]In the actual implementation of the simulator, the data is moved to the MSHR and the block is deallocated from the cache when a write-back is needed. Hence, the write-back buffer has as many entries as available MSHRs.



(a) Execution time normalized with respect to the most similar non fault-tolerant protocol for each case.



(b) Execution time normalized with respect to DirCMP.

Figure 8.1: Execution time overhead of the fault-tolerant cache coherence protocols in absence of faults

requests that arrive to the new owner node during this short period will have their latency increased by as much as the remaining blocked ownership period, or until a persistent request is issued in the case of FtTokenCMP.

Network traffic overhead

As we have shown in the previous section, in absence of faults the execution time of the applications is unaffected by the fault tolerance measures. However, these measures do have a cost even when no faults occur: the acknowledgments used to ensure reliable transference increase the number of messages that travel through the network[2] (see sections 4.2 and 5.2). This exchange of acknowledgments is the main difference in the behavior of our protocols with respect to their non fault-tolerant counterparts.

We have measured the network traffic of our set of benchmarks classifying each message by its type. We have grouped these types in the following categories:

Control: All control messages including: transient and persistent requests in the token based protocols, invalidation requests, dataless responses and every other dataless message which is not included in the *ownership* category.

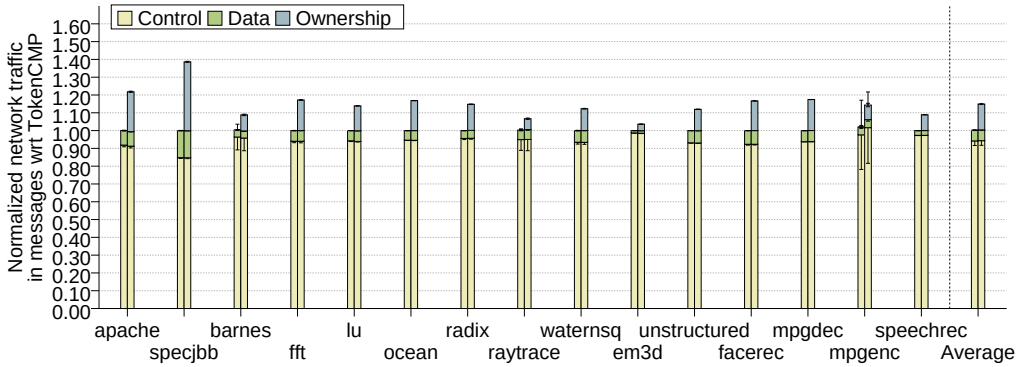
Data: All data carrying messages (responses with data and write-back messages).

Ownership: Ownership acknowledgment and backup deletion acknowledgment messages.

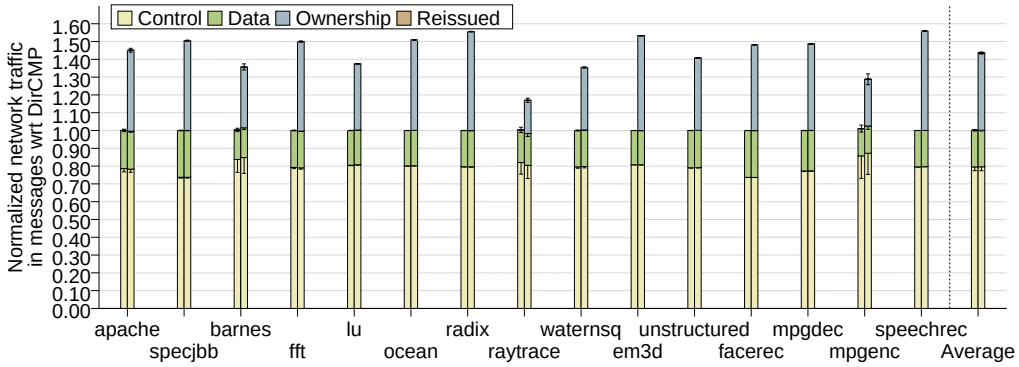
Figure 8.2 shows the network overhead measured as relative increase in the number messages of each category transmitted through the network for our set of benchmarks. As can be seen, the overhead varies for each protocol. On average, it is less than 10% in the case of FtHammerCMP, 15% in the case of FtTokenCMP and 40% in the case of FtDirCMP. That is, the most network efficient protocols have a higher overhead than other protocols that usually require more bandwidth. This is because the overhead per ownership transference is similar in all the protocols (2 messages or 1 message in some cases), hence it will be relatively smaller when the total traffic is higher. Figure 8.3 compares the network traffic of all the protocols in terms of number of messages.

Figure 8.2 also shows that the overhead comes entirely from the extra acknowledgment messages used to ensure reliable transference of data (*"ownership"* part of each bar).

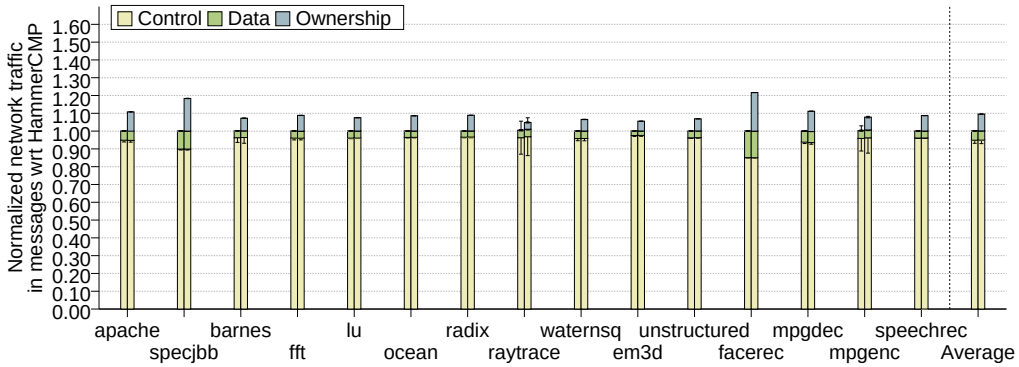
[2] Arguably, this extra network traffic could increase the execution time too if the interconnection network does not provide enough excess bandwidth.



(a) TokenCMP and FtTokenCMP



(b) DirCMP and FtDirCMP



(c) HammerCMP and FtHammerCMP

Figure 8.2: Network overhead for each category of message in terms of number of messages. Results are normalized with respect to the total number of messages of the most similar non fault-tolerant protocol

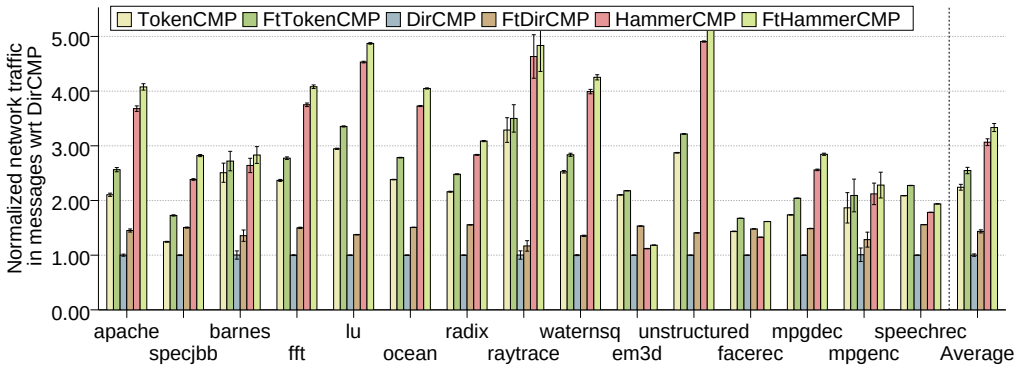
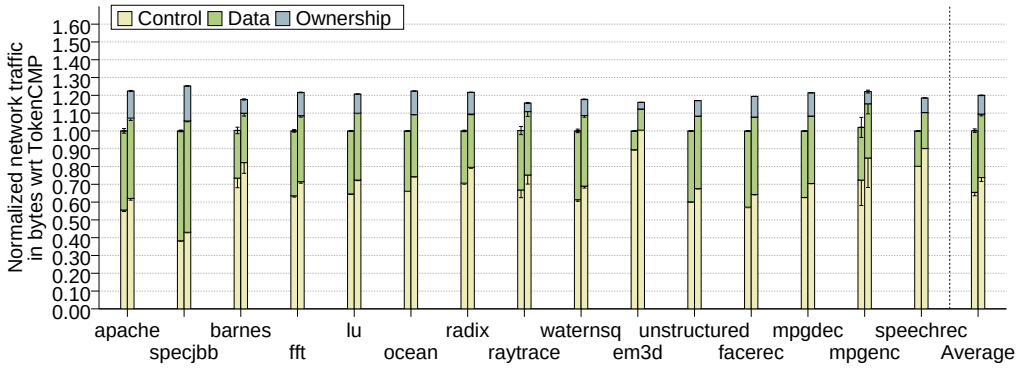
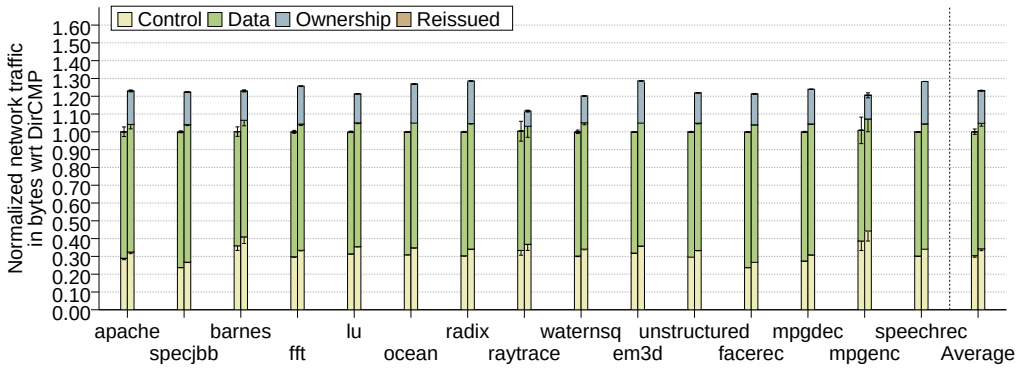


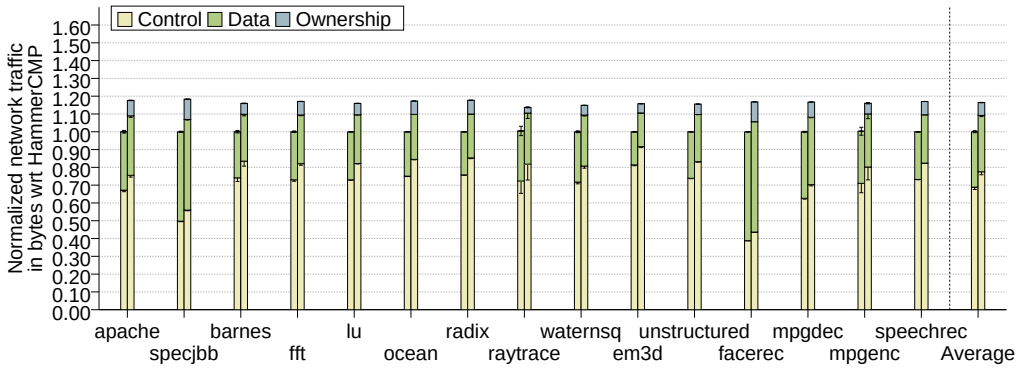
Figure 8.3: Relative network traffic in terms of messages for each protocol normalized with respect to DirCMP



(a) TokenCMP and FtTokenCMP



(b) DirCMP and FtDirCMP



(c) HammerCMP and FtHammerCMP

Figure 8.4: Network overhead for each type of message in terms of number of bytes. Results are normalized with respect to the total number of bytes of the most similar non fault-tolerant protocol

Moreover, since these messages are small (like the rest of control messages), the overhead drops considerably in the case of FtDirCMP if we measure it in terms of bytes that travel through the network (that is, in terms of bandwidth) even considering that every message is one byte longer in the fault-tolerant protocols. These results can be seen in figure 8.4. We have increased in one byte the message sizes of the fault-tolerant protocols to accommodate the *requests serial numbers* and *token serial numbers*. This means 1.14% increase in size for data messages and 12.5% increase for control messages. In terms of bandwidth, the overhead is less than 25% for FtDirCMP, less than 20% for FtTokenCMP and approximately 17% for FtHammerCMP.

8.2 Performance degradation under faults

We have already shown the overhead introduced by the fault tolerance measures of our cache coherence protocols. However, the purpose of our protocols is to allow the correct execution of parallel programs in presence of transient faults in the interconnection network. To be useful, the protocols need to provide good performance even when faults occur.

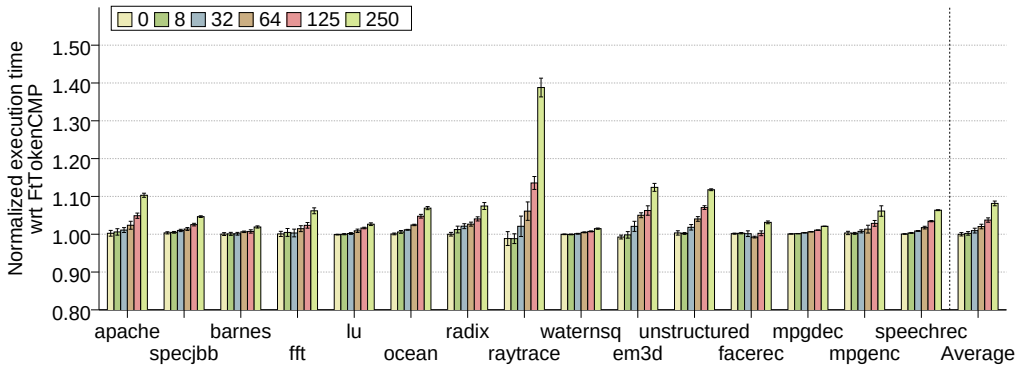
Faults will degrade performance in two ways: the execution time will increase (because the fault needs to be detected using a timeout and dealt with with messages which are in the critical path of some misses) and the network traffic will increase (because of the extra messages used to deal with the fault).

Execution slowdown

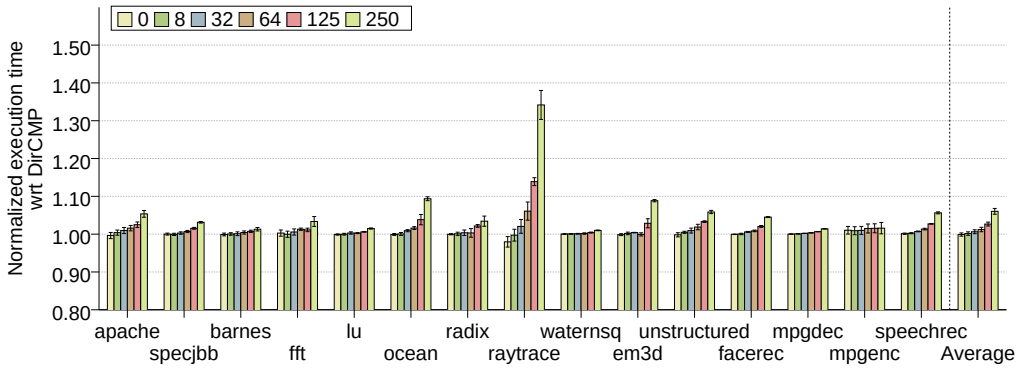
When a fault happens, it is detected in our three protocols after a certain amount of time by means of one of the fault-detection timeouts. We will show how we adjusted these timeouts in section 8.3. Figure 8.11 shows that, for a fixed fault rate, the performance degradation depends on the values chosen for the timeouts.

On the other hand, figure 8.5 shows how the execution time of each fault-tolerant protocol increases as the fault rate increases. For the three protocols, the execution time increases almost linearly with the fault rate, as can be seen more clearly in figure 8.6.

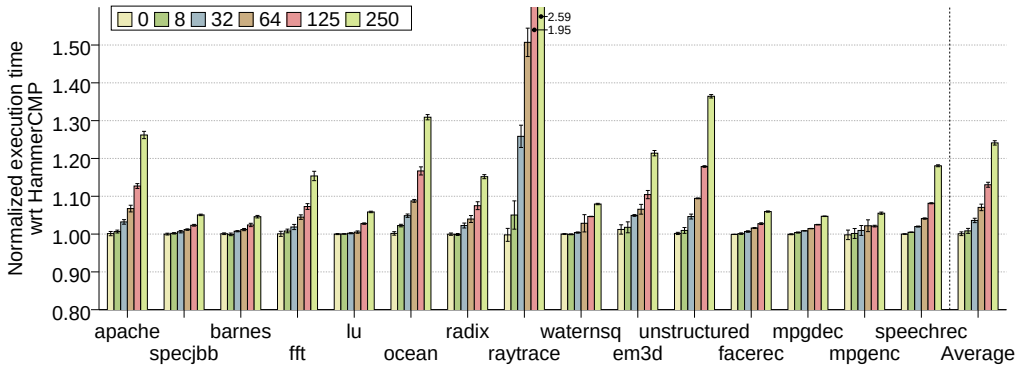
In all cases, the protocols can support very high rates with only a small performance degradation. On average, a performance degradation of 10% requires a fault rate of more than 250 corrupted messages per million in the cases of FtDirCMP and FtTokenCMP and more than 100 corrupted messages per million



(a) TokenCMP and FtTokenCMP



(b) DirCMP and FtDirCMP



(c) HammerCMP and FtHammerCMP

Figure 8.5: Performance degradation with different fault rates (in messages corrupted per million of messages that travel through the network) for each fault-tolerant protocol with respect to its non fault-tolerant counterpart

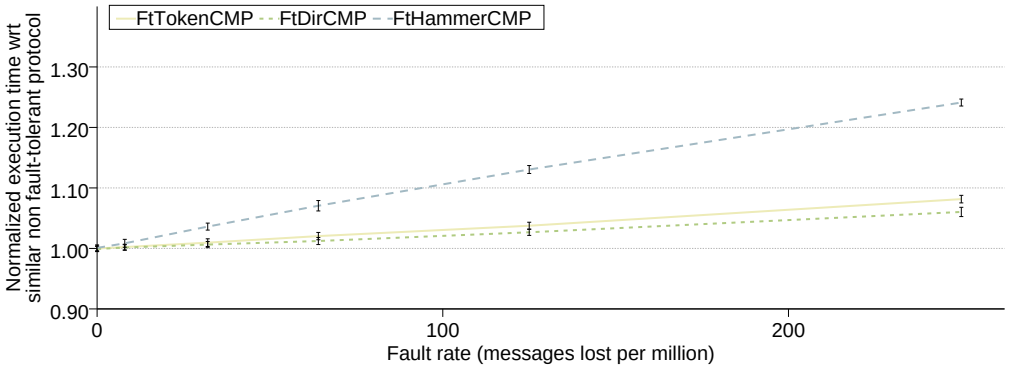


Figure 8.6: Average performance degradation with different fault rates (in messages corrupted per million of messages that travel through the network) for each fault-tolerant protocol with respect to its non fault-tolerant counterpart

in the case of FtHammerCMP. We think that these fault rates are much higher than what should be expected in a realistic scenario.

With the fault rates tested, only one benchmark suffers a slowdown of more than 10% in FtDirCMP, 20% in FtTokenCMP or 40% in FtHammerCMP. The benchmark with the highest performance degradation under faults with our three protocols is *Raytrace*. A possible explanation of this fact is the high contention observer in this benchmark. The second program with more contention in our benchmark set is *Unstructured*, which is also the second one most affected by faults in FtHammerCMP.

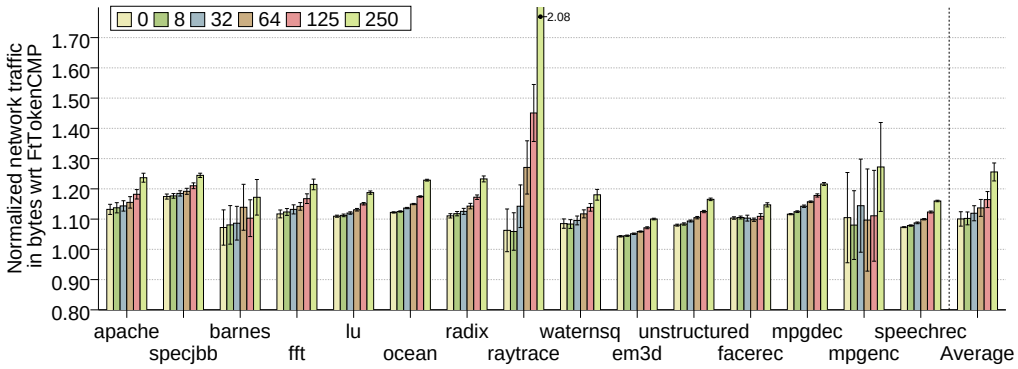
As said above, the performance degradation depends mainly on the latency of the error detection mechanism. Hence, shortening the fault detection timeouts can reduce performance degradation when faults happen but at the cost of requiring more bits to encode the request serial numbers of FtDirCMP and FtHammerCMP (due to the more frequent request reissues) and at the risk of increasing the number of false positives which could lead to performance degradation in the fault-free case.

Network traffic increase

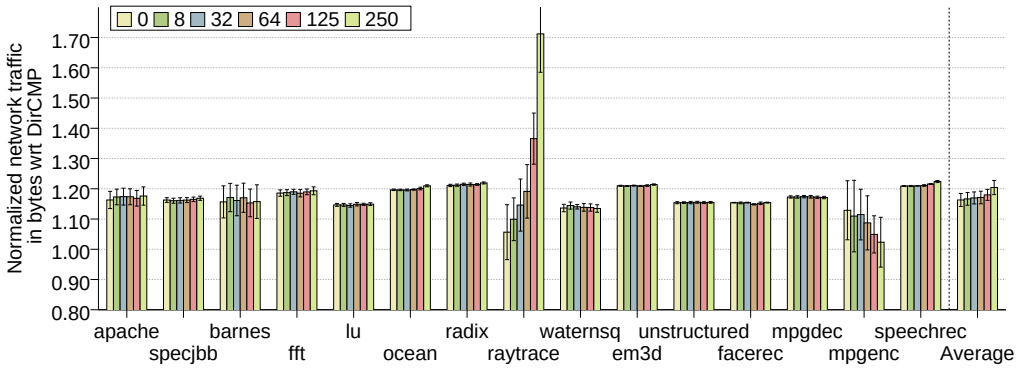
When a fault happens, extra messages are required to deal with it (either the request has to be reissued or a token recreation process has to be requested). These extra messages consume extra network bandwidth and this contributes to the execution time increase described in the previous section.

Figure 8.7 shows how the interconnection network traffic increases as the fault rate increases. The extra traffic is (mostly) due to the recovery process required when a fault is detected. For this reason, the network overhead of FtTokenCMP increases more rapidly than that of FtHammerCMP and the overhead of FtHammerCMP increases more rapidly than that of FtDirCMP, as can be seen more clearly in figure 8.8.

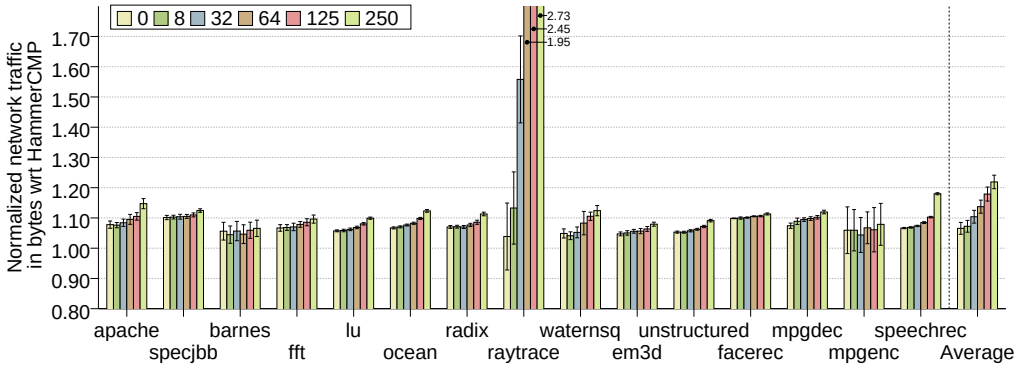
Each time a fault is detected in FtTokenCMP a new token recreation process needs to be initiated (see section 4.2). This process requires four messages per node plus the message for requesting it and the message to indicate that it has finished. In contrast, fault recovery in FtHammerCMP requires only two messages per node (an invalidation and its acknowledgment or a forwarded request and its response) plus the reissued request and unblock messages. Furthermore, FtDirCMP requires at most as many messages as FtHammerCMP but usually much fewer messages because only the current sharers of the memory line need



(a) FtTokenCMP



(b) FtDirCMP



(c) FtHammerCMP

Figure 8.7: Network usage increase with different fault rates (in messages corrupted per million of messages that travel through the network) for each fault-tolerant protocol with respect to its non fault-tolerant counterpart

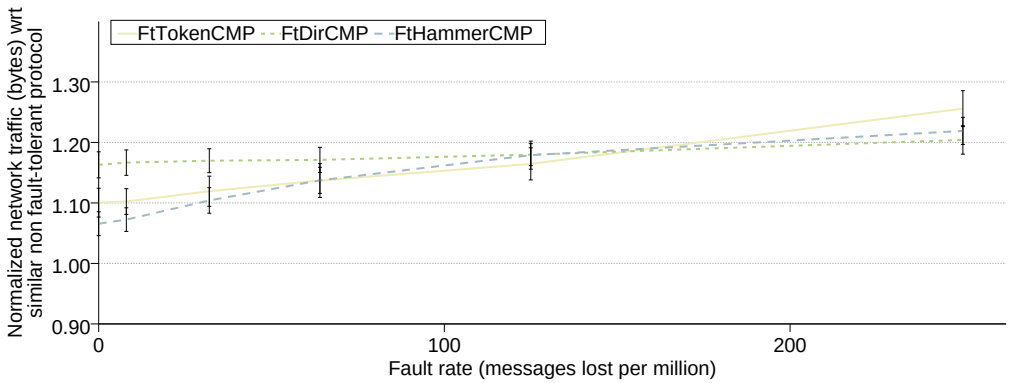


Figure 8.8: Average network usage increase with different fault rates (in messages corrupted per million of messages that travel through the network) for each fault-tolerant protocol with respect to its non fault-tolerant counterpart

to be involved, so although its overhead is higher for small fault rates, it is lower for higher fault rates.

As shown in this section, the total amount of network traffic increases with faults. However, since the execution time also increases and does it more rapidly, the amount of traffic per cycle actually decreases sightly in most cases. This is because the processors are stalled while faults are being recovered waiting for the corresponding cache misses to be resolved and hence cannot issue new requests.

Effect of bursts of faults

Until now, we have assumed that all faults are distributed evenly 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. That is, several messages may be discarded in a burst. Note that, most likely, each of those messages will be part of different coherence transactions involving different addresses.

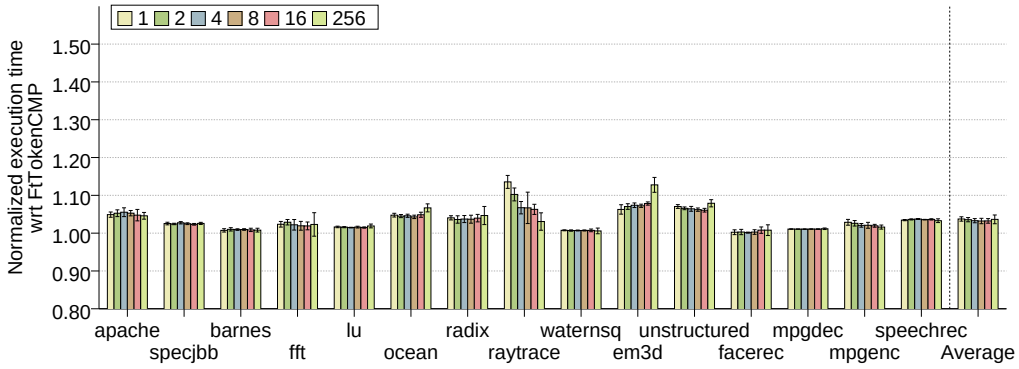
In this section, we assume that each fault affects several messages that pass consecutively through some point in the network, depending on the burst size. For consistency with the rest of the evaluation and as explained in section 7.1, we will express the fault rates in total number of corrupted messages per million of messages that travel through the network. In other words: for the same fault rate, a higher burst size means that less fault events have occurred, but each event has affected more messages. In our protocols, each affected memory transaction needs to be recovered independently, which means approximately one recovery process per discarded message.

Figure 8.9 shows how the execution time and network overhead vary as the length of bursts increases with a fixed fault rate of 125 messages per million of messages that travel through the network.

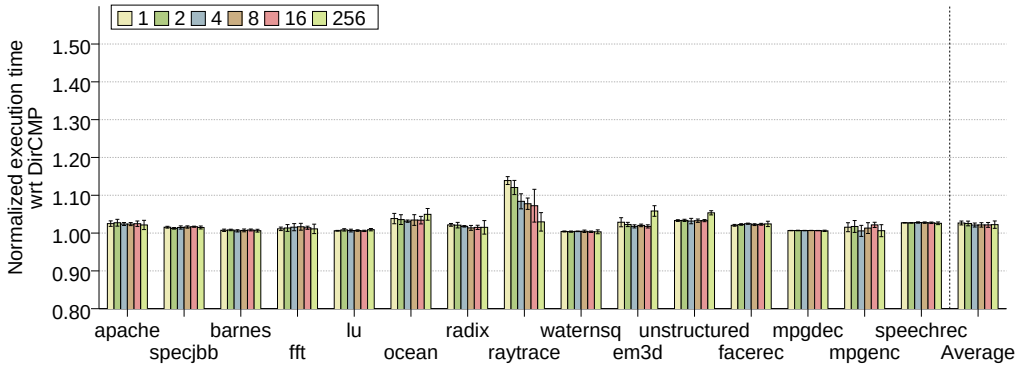
As can be seen in figure 8.9, on average and for most applications, increasing the length of the burst of dropped messages has no negative effect on the execution time of applications. Only some applications like *Em3d* and *Ocean* have a longer execution time under longer bursts of faults in some cases.

Actually, most applications seem to benefit from longer bursts. This can be explained due to the fact that although the total number of messages that get corrupted is approximately the same, and hence the protocols need to perform approximately the same number of recovery processes, the total overhead of all these recovery processes is lower.

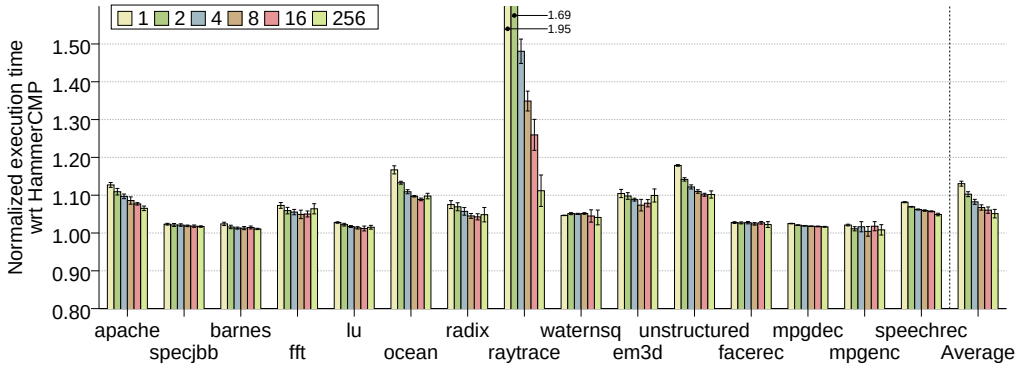
Since the recovery of each message usually happens in parallel to the recovery



(a) FtTokenCMP



(b) FtDirCMP



(c) FtHammerCMP

Figure 8.9: Performance degradation with burst faults of several lengths for each fault-tolerant protocol with respect to its non fault-tolerant counterpart. The total fault rate is fixed to 125 corrupted messages per million of messages that travel through the network

of other messages, the overhead of the recovery process may actually be reduced over the whole execution of the program. Raytrace is the application that benefits the most from this effect, and is also the application which is most affected by single message faults, as can be seen in figure 8.5.

8.3 Adjusting fault tolerance parameters

Our fault-tolerant cache coherence protocols introduce a number of parameters whose values affect the overhead introduced by the fault tolerance measures in terms of hardware required to implement them, the performance overhead without faults, the performance degradation when faults occur and the maximum fault rate that can be supported by the protocols while still guaranteeing correct execution of the parallel programs. In this section we will show how to adjust each of these parameters.

The optimal values for the fault tolerance parameters depend on the values of the rest of the configuration parameters of the system. The results presented in these section assume the configuration values shown in table 7.2a and we will derive the values shown in table 7.2b.

Adjusting the backup buffer size in FtTokenCMP

In section 8.1 we have shown that the execution time of our fault-tolerant cache coherence protocols is virtually the same than the execution time of the base protocols. These results assumed that FtDirCMP and FtHammerCMP used the write-back buffer to store backups when replacing owner data, and that FtTokenCMP moved backups to a backup buffer of 2 entries when required as explained in section 4.2. In this section we show the execution time overhead of implementing FtTokenCMP with a variety of sizes for its backup buffer, including not having a backup buffer at all, and justify our decision of using only two entries in our experiments.

Without a backup buffer or a write-back buffer, the latency of those L1 misses which need a previous replacement of an owned memory line to make room for the data increases, since the replacement cannot be immediate because it has to wait until an *ownership acknowledgment* is received from the L2 cache.

Figure 8.10 shows the execution time overhead of FtTokenCMP with several backup buffer sizes.

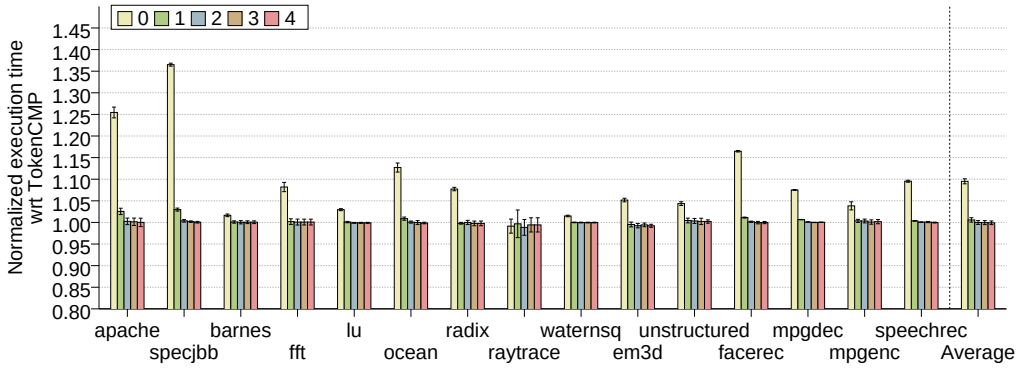


Figure 8.10: Execution time of FtTokenCMP normalized with respect to TokenCMP when no faults occur and using different backup buffer sizes

As derived from figure 8.10, without a backup buffer the overhead in terms of execution time is almost 10% on average and more than 35% in the worst case (*SpecJbb*). Benchmarks with fewer replacements of owned data from L1 to L2 suffer lower slowdowns. Fortunately, when we add a backup buffer with just one entry the overhead is reduced dramatically to almost 0 on average and less than 3% in the worst case. Larger numbers of entries for the backup buffer reduce the overhead to unmeasurable levels for all benchmarks.

Hence, we have decided to use two entries for the backup buffer of FtTokenCMP. In the case of FtDirCMP and FtHammerCMP, the size of the write-back buffer does not need to be increased with respect to the base protocol (the only difference is that some entries are occupied for a few more cycles, until the *ownership acknowledgment* arrives). Since DirCMP and HammerCMP use three-phase write-backs, they already require a write-back buffer for good performance.

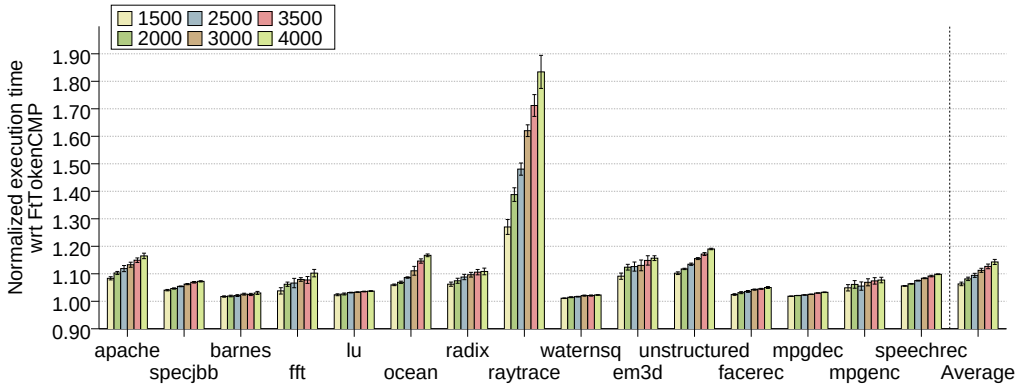
Adjusting the fault detection timeouts

The second set of parameters is the value of each of the timeouts used by the fault-tolerant protocols to detect potential deadlocks and start the recovery process.

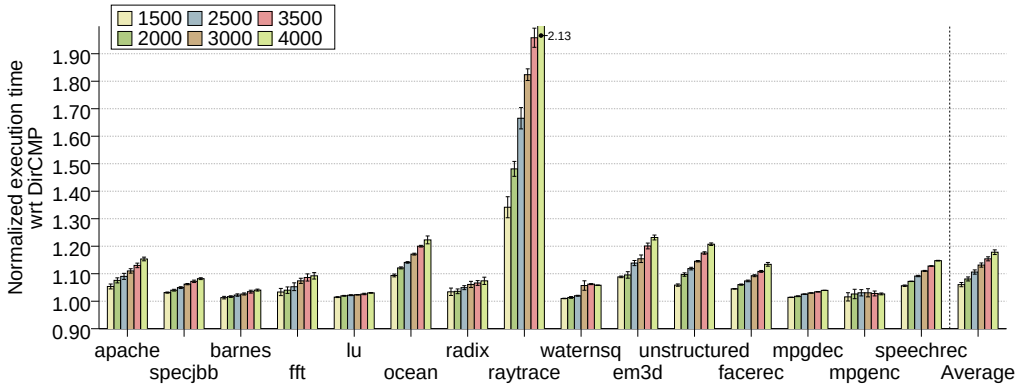
As explained before, all our fault-tolerant protocols rely on the fact that every message loss which is not harmless will eventually lead to a deadlock, so timeouts can be used to detect all faults. Each protocol requires up to four timeouts which are active at different places and times during a memory transaction or cache replacement. A brief description of each timeout can be seen in table 4.2 for FtTokenCMP, table 5.3 for FtDirCMP and table 6.1 for FtHammerCMP.

This way, the values of these timeouts determine the latency of fault detection and, hence, choosing their values appropriately helps to achieve lesser performance degradation since in the case of a transient fault, recovery would start earlier and no false positives would arise. For example, figure 8.11 shows for each of fault-tolerant protocol how the performance degradation with respect to the similar base non fault-tolerant protocol increases as the value of the timeouts increases while keeping the fault rate constant at 250 lost messages per million of messages that travel through the interconnection network. This fault rate is much higher than what we would expect in a real scenario.

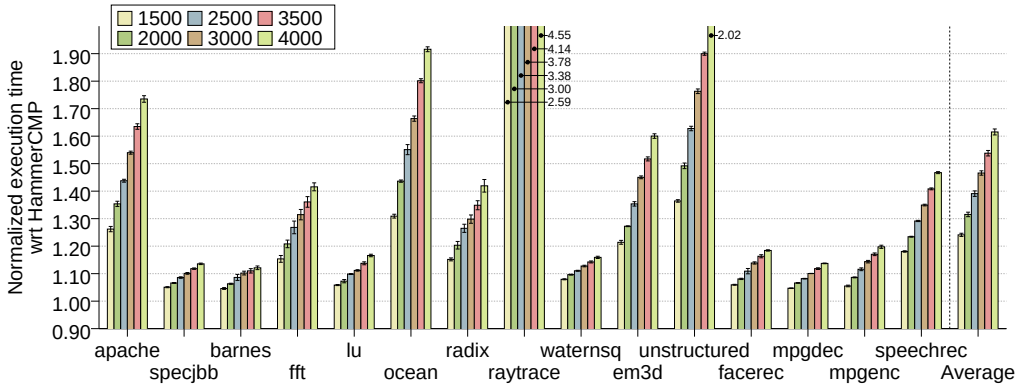
In figure 8.11, we have used the same values for all the timeouts of each fault-tolerant protocol. In principle, each timeout could be adjusted independently of the others, but we have decided to use the same value for all of them for simplicity.



(a) FtTokenCMP



(b) FtDirCMP



(c) FtHammerCMP

Figure 8.11: Performance degradation of the fault-tolerant protocols with a constant fault rate of 250 lost messages per million of messages travelling through the network and with different values for the fault detection timeouts, using the same value for all the timeouts of each protocol

We considered using different values for each timeout, but our experiments did not show any significant advantage on doing so.

We would like to make the values as small as possible to minimize performance degradation when faults occur. However, using excessively small values can be detrimental for performance in the fault-free case and may even prevent forward progress. This is because there is always risk of false positives with this deadlock detection mechanism since, occasionally, a timeout may trigger before a miss can be resolved. If the false positive rate is too high, the fault recovery mechanism will be invoked too often. A good value should avoid false positives while being as short as possible so that actual faults are detected as soon as possible to avoid excessive performance degradation.

In order to determine a minimum value for the fault detection timeouts, we have performed simulations of all the protocols considered in this thesis with virtually infinite timeouts (in the case of the fault-tolerant protocols) and no faults. Since false positives occur when a timeout triggers before a miss has had enough time to be satisfied, we have measured the maximum miss latency for each protocol to use it as a lower bound for the timeout values. We show the results in figure 8.12.

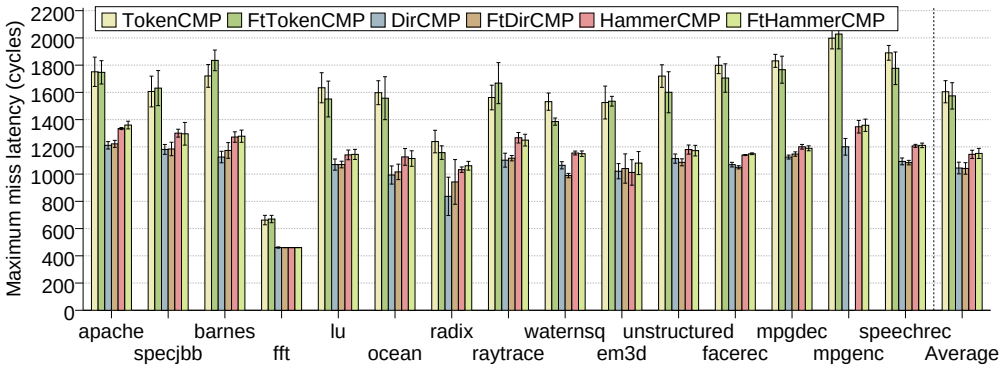


Figure 8.12: Maximum miss latency (in cycles) of each cache coherence protocol without faults

Looking at figure 8.12, we can see that, for each pair of base and fault-tolerant protocol, both have the same maximum miss latency on average. Only a few cases show a statistically significant difference, like *WaterNSQ* in FtTokenCMP. This is expected, since in absence of faults our fault-tolerant protocols should behave very similarly to their non fault-tolerant counterparts.

We can see than the maximum latency in case of FtDirCMP is just over 1000 cycles, and it is less than 1400 for all benchmarks. In the case of FtTokenCMP, the average maximum latency is almost 1600 cycles and it is never reaches 1900 cycles except in the case of *MPGenc*. And in the case of FtHammerCMP, the average maximum miss latency is just over 1150 cycles and never over 1400 cycles. It is important to remember that the values shown in figure 8.12 are the worst case latencies of misses for each protocol, which means that they do not have a huge impact in the execution time of applications with each protocol (see figure 8.1b). Since the exchange of messages of DirCMP and HammerCMP are identical in the worst case except that HammerCMP sends more invalidation messages, both protocols have similar maximum latencies (the same can be said with respect to FtDirCMP and FtHammerCMP respectively). TokenCMP and FtTokenCMP have higher maximum latencies because their worst case happens when a persistent request has to be issued because the token protocol timeout triggers after an unsuccessful transient request.

Considering this data, we have chosen a value of 1500 cycles for all the fault detection timeouts in FtDirCMP and FtHammerCMP, and a value of 2000 cycles in the case of FtTokenCMP. These values are large enough to avoid false positives in almost every case and, as shown in figure 8.11, keep the performance degradation low in most cases when faults actually occur. Making this value smaller achieves very little benefit while significantly increasing the risk of false positives.

Adjusting request serial number sizes for FtDirCMP and FtHammerCMP

FtDirCMP and FtHammerCMP rely on the ability to detect and discard stale responses to reissued requests (for example, to be able to discard old acknowledgments to reissued invalidation requests which could lead to incoherence in some cases). This ability ensures that the fault recovery mechanism works even in case of false positives. As explained in section 5.2, these two protocols use *request serial numbers* to do this.

Most times, when a node reissues a request due to a false positive of one of the timeouts, it will receive the stale responses shortly after and will discard them because the serial number will be different. More precisely, due to the way that serial numbers are chosen for the reissued requests (that is, increasing the serial number of the original request and wrapping to zero if necessary), the lower order bit will be different if the request has been reissued only once.

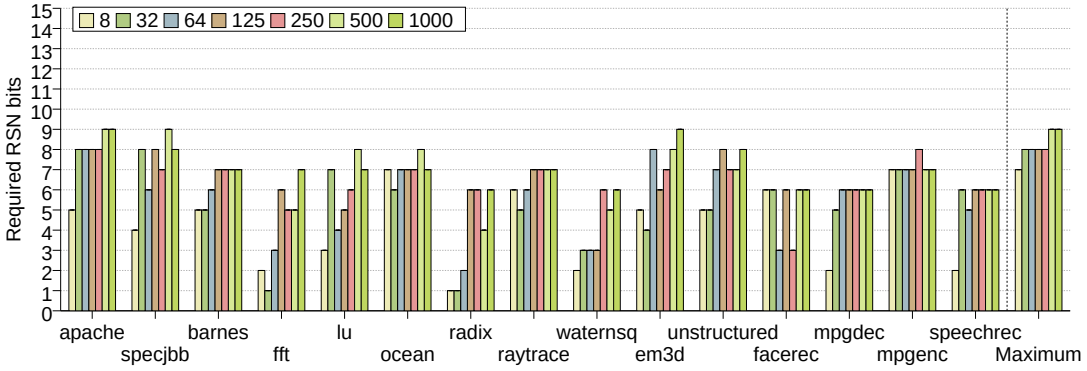
However, in situations of unusually high network traffic or high fault rates, a request may be reissued several times before the stale responses are received. Since the number of different serial numbers is finite, if the serial numbers are encoded with a small number of bits and the request is reissued many times in a row, the expected serial number for a reissued request may match the serial number of a stale response. This would prevent discarding the stale response and could cause an error.

Since, assuming fixed timeouts, the number of reissued messages increases as the fault rate increases, the number of bits used to encode the request serial numbers determines the maximum fault rate supported by each protocol. 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 still being sufficient to ensure that the protocol will be able to work correctly under the target fault rate and with the chosen timeout values.

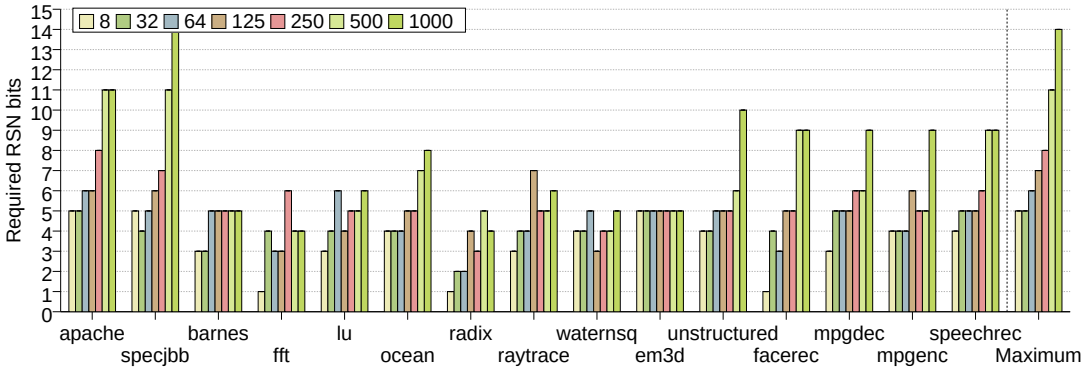
To decide a good value for the size of request serial numbers, we have measured how many bits would be necessary to support each of a set of fault rates. To do this, we have performed simulations of FtDirCMP and FtHammerCMP using the timeouts decided in the previous section. In these simulations, we have used 32 bit request serial numbers but we have recorded how many bits were required to distinguish all the request serial numbers that needed to be compared. For doing this, 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 the number of bits required to ensure correctness. These results are shown in figure 8.13, and they include several (at least 8) simulations for each case.

As seen in figure 8.13a, in the case of FtDirCMP 9 bits are enough for all the tested fault rates, and 8 bits suffice for fault rates up to 250 corrupted messages per million. In the case of FtHammerCMP, up to 14 bits are required for the highest fault rate tested, but 8 bits are enough for fault rates up to 250 corrupted messages per million too.

Hence, we have used 8 bits to encode the request serial numbers for our experiments. With this configuration, our protocols should be able to support fault rates up to 250 corrupted messages per million (and we will show results only up to such fault rate from now on). We think that this fault rate is already much higher than what should be expected in any case, but adjusting the number of bits used to encode request serial numbers or increasing the timeouts would allow the protocols to support even higher fault rates.



(a) FtDirCMP



(b) FtHammerCMP

Figure 8.13: Required request serial number bits to be able to discard every old response to a reissued message when faults occur. In the fault-free case, no bits are required at all in any of our tests

8.4 Hardware implementation overheads

In this section, we make a brief list of the main sources of overhead in terms of extra hardware required to implement the fault tolerance measures of our protocols.

- The *token serial number table* used in TokenCMP (see section 4.2) can be implemented with a small associative table at each cache and at the memory controller to store those serial numbers whose value is not zero. All these tables always hold the same information (they are updated using the token recreation process).

  Since the token serial number of a memory line is initially zero and only changes when a token recreation process is invoked on that line, only a few entries are needed and each entry only requires a few bits.

  When using n bits to encode the token serial numbers, if the tokens of any memory line need to be recreated more than $2^n - 1$ times the counter wraps to zero (effectively freeing an entry in the table).

  The number of bits per entry should be enough so that all messages carrying an old token serial number can be discarded before the counter wraps around, even when several token recreation processes are requested consecutively for the same memory line. Due to the long latency of this process, the number of required bits is very low because usually all stale tokens have arrived to some node even before the token recreation process starts.

  If the token recreation process is performed in a number of different memory lines larger than the number of entries in the token serial number table, the least recently modified entry (or any entry at random) is evicted from the table by means of using the token recreation process to set its serial number to zero.

  The results presented in the previous sections have been obtained using two bits to encode the token serial numbers[3] and having 16 entries at each token serial number table.

- The *request serial numbers* used by FtDirCMP and FtHammerCMP (see section 5.2) do not need to be kept once the memory transaction has completed, hence they do not need any additional structure or any extra

[3]Of all the simulations performed, only two of them actually required more than one bit for ensuring successful recovery from all faults.

bits in the caches. They need to be kept until the memory transaction has been resolved.

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 8.3, using 8 bits to encode request serial numbers is enough to achieve tolerance to very high fault rates, and even less bits are required to support more realistic but still very high fault rates.

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

- The timeouts used for fault detection in the three fault-tolerant protocols 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 of each protocol, no more than one counter is required at any time in the same node for a single coherence transaction.

  Moreover, 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 (see appendix C for additional details). Hence, a correct implementation requires an additional virtual network to ensure deadlock free operation. See appendix C for more details.

- To deal with the increased pressure in caches due to the blocked ownership and backup states and the effect of the reliable ownership transference mechanism in replacements, either a write-back buffer or a backup buffer has to be added (see section 4.2). All the results for the FtTokenCMP protocols use a 2-entry backup buffer, while FtDirCMP and FtHammerCMP use a write-back buffer like DirCMP and HammerCMP.

We think that the hardware overheads for the fault tolerance measures of our cache coherence protocols are assumable given the level of fault tolerance that they provide for faults in the interconnection network. Achieving this fault tolerance at the interconnection network level instead of at the protocol network level would have also its own hardware overhead.

From a hardware design point of view, the greatest cost of our proposals is the increased complexity. Cache coherence protocols are already complex and our fault tolerance measures significantly increase this complexity. However, these fault tolerance measures also add some resiliency to design errors in the protocols and may help to simplify other parts of the system, notably the interconnection network.

Chapter 9

Conclusions and future ways

9.1 Conclusions

Reliability of electronic components is already a major concern in computer architecture, and its importance is expected to grow as the scale of integration increases. Transient faults in particular are becoming a common event that needs to be considered when designing any chip using deep-submicron technologies. We expect that future systems will employ many techniques at several levels to deal with them, from mitigation techniques like using new materials (for example, silicon on sapphire or silicon on insulator) and careful layouts in critical parts of circuits to system-level fault tolerance like checkpointing support in operating systems or hypervisors. Architecture-level techniques will become increasingly common, and they will be included in commodity designs to make them cost effective.

Speed, power-efficiency and reliability are three variables that every computer architect wants to maximize, but unfortunately none of them can be improved without negatively affecting the other two. In addition, transient faults will affect most of the components of a CMP. To achieve good reliability for the whole chip, each component will have to be as resilient to transient faults as possible while still keeping good performance, and as much power-efficiency as possible too.

In this thesis, we have proposed a new approach to deal with transient failures in the interconnection network of a CMP. The interconnection network of future many-core CMPs is expected to be one of the components most vulnerable to transient faults due to the significant on-chip area that it occupies, which makes

it prone to particle impacts, and the relatively long wires that are needed to build it, which make it prone to crosstalk noise. Previous proposals to deal with these problems involved redesigning the interconnection network to make it fault-tolerant. Instead, we have shown how to modify the coherence protocol, which sits in the layer immediately over the interconnection network, so that it does not assume that all messages will be correctly delivered by the interconnection network. Our protocols only assume that most messages are indeed delivered, and that those messages which are delivered are correct, which can be easily ensured using error detection codes.

We have identified the problems that an unreliable interconnection network poses to a cache coherence protocol. We have found that, for those protocols where every request requires a response and every invalidation requires an acknowledgment message, a lost message cannot lead to a coherence violation and all faults eventually lead to a deadlock. Hence, we can detect faults using timeouts at the protocol level.

We have shown that fault tolerance measures can be added to cache coherence protocols without increasing the execution time of parallel programs running on the CMP when no faults occur and that the main cost of this approach is a moderate increase in network traffic. For this, we have developed three fault-tolerant cache coherence protocols (FtTokenCMP, FtDirCMP and FtHammer-CMP) based on different base cache coherence protocols (TokenCMP, DirCMP and HammerCMP, respectively) and have evaluated them using full-system simulation of several server, scientific and multimedia parallel applications.

The relative increase in interconnection network traffic due to our fault tolerance measures depends on the cache coherence protocol. The highest overhead in network traffic that we have measured is approximately 30% (in terms of bytes) for FtDirCMP with respect to DirCMP, which is the most efficient base protocol in terms of interconnection network traffic that we have used. Fault-tolerant cache coherence protocols based on less traffic efficient base protocols have lower interconnection traffic overhead, since the amount of extra traffic generated by the fault tolerance measures is proportional to the number of transfers of ownership of memory lines between coherence nodes. This quantity depends on the particular application as is roughly the same for different coherence protocols.

Since transient faults should be infrequent, we can use simple recovery mechanisms to deal with them when they happen. These mechanisms need to be effective, but their efficiency is not a major concern since the performance degradation when faults happen is dominated by the latency of the fault detection mechanism (that is, the value of the fault detection timeouts). We have proposed

two recovery mechanisms: the *token recreation process* and reissuing requests. The first one is centralized and requires a serial number for each memory line[1], while the second one is distributed and requires a serial number for each request.

We have measured the performance degradation for our protocols in presence of faults happen using the mechanisms mentioned above, and we have found that, on average, it is moderate even for the highest fault rates that we have tested (less than 25% of slowdown with 250 messages lost per million of messages that travel through the network for FtHammerCMP, which is the protocol whose performance degrades most rapidly). We have also shown how to adjust the maximum fault rate supported and the performance degradation of the fault-tolerant cache coherence protocols. The message loss rates used for our tests are much higher than the rates expected in the real world, hence under real world circumstances no important slowdown should be observed even in the presence of transient faults in the interconnection network.

Dealing with transient faults in the interconnection network at the cache coherence protocol level has two key advantages with respect to building a fault-tolerant interconnection network:

- The cache coherence protocol has more information about the meaning (and hence, criticality) of each message that travels through the interconnection network. This way, it can spare efforts for those messages whose loss is not very important. For example, our protocols only add acknowledgments for the few messages that may carry the only valid copy of a memory line (messages carrying owned data). Hence, the total traffic overhead can be reduced.

- By freeing the interconnection network of the responsibility of ensuring reliability, the interconnection network designer has much more flexibility to design it and can spend more effort in improving its performance. Since the performance of the interconnection network is critical to the performance of the whole system, we expect that a system with an unreliable and fast interconnection network will perform better than a system with a reliable network, even if the cache coherence protocol of the first one needs to recover from lost messages occasionally.

Our fault-tolerant cache coherence protocols can also benefit if the reliability of the interconnection network increases, since this way the fault recovery

[1] Although only a small number of these serial numbers need to be actually stored (see section 4.2).

mechanisms need to be invoked less often. This means that reliability in the interconnection network can be traded for performance, but still ensuring correct operation when faults happen. Our evaluation results show that, if the fault rate is kept low enough (up to 32 lost messages per million of messages that travel through the network for FtTokenCMP and FtDirCMP or 8 lost messages per million of messages that travel through the network for FtHammerCMP), the average performance degradation is almost unmeasurable.

In this thesis we have designed three fault tolerant cache coherence protocols based on three coherence protocols which were already proposed. We expect that the same principles can be applied to many other protocols with only minor adjustments. In particular, we have extended the generic token counting rules to ensure that, in addition to guaranteeing memory coherence, they also guarantee no data loss even when not all messages are delivered by the interconnection network.

In this way, fault-tolerant cache coherence protocols provide a solution to transient faults in the interconnection network of CMPs with very low overhead which can be combined with other fault tolerance measures to build reliable CMPs.

9.2 Future ways

The results presented in this thesis open a number of interesting new research paths. Amongst them, we have identified the following:

- We want to explore the application of these fault tolerant measures in new promising cache coherence protocols [71,107]. We expect that, in most cases, the same techniques will be easily applicable to a wide range of protocols with only small modifications.

- Similarly, we can build a fault-tolerant cache coherence protocol with support for transactional memory [49,83] based on one of the several recent proposals. By assuming a transactional memory programming model, we think that fault recovery may be done in alternative ways to those proposed here. We also think that it is worth investigating how the mechanisms that support transactional memory can be leveraged for fault tolerance.

- The fault-tolerant cache coherence protocols presented in this thesis are based on previously existing non fault-tolerant protocols. We think that

building a cache coherence protocol with support for fault tolerance from the ground up can be simpler and may even simplify the design of the protocol overall. For example, race conditions that would lead to deadlocks in other protocols if they are not carefully considered could be dealt with using the same recovery mechanism than transient faults.

- We think that the coherence protocol can also help to deal with permanent and intermittent faults in the interconnection network, in addition to transient faults. The same fault detection mechanisms presented in this thesis can be adapted to detect intermittent and hard faults and trigger a reconfiguration of the interconnection network to avoid faulty links and routers.

- Finally, since the main feature of our protocol is 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 design of interconnection networks that are simpler or that have better performance. These new interconnection networks may not deliver all messages correctly, but may still have good performance as long as most messages actually arrive to their destination when used in conjunction with a fault-tolerant protocol. By not having to guarantee the delivery of all messages, the interconnection network may use fewer resources, improve its performance or both.

  For example, deadlock situations in one of these interconnection networks could be recovered by discarding some of the messages involved in the cycles that cause the deadlock. Since our protocols can tolerate the loss of any message, the interconnection network can decide to drop any message when a potential deadlock is detected as long as this is done infrequently enough so that all messages are eventually delivered without livelock.

  Also, our fault-tolerant cache coherence protocols enable the use of bufferless interconnection networks (also known as "hot-potato" routing or deflection routing) [47,48,86] which have to resort to packet dropping when a packet arrives to a switch that has all its output ports busy. Bufferless interconnection networks provide significant benefits in area and power consumption.

Appendix A

Base token-based protocol

Our fist proposal for a fault-tolerant cache coherence protocol is based on the token coherence framework [68, 67]. In particular, we used TokenCMP [70], a token-based cache coherence protocol suitable for CMPs and multiple CMPs (M-CMPs, a system built using several chips, each one with several cores).

Token coherence [67] is a framework for designing cache coherence protocols whose main asset is that it decouples the correctness substrate from the performance policy. The correctness substrate ensures that coherence cannot be violated and ensures forward progress, while the performance substrate dictates how requests are handled in the common cases.

This allows great flexibility, making it possible to adapt the protocol for different purposes easily because the performance policy can be modified without worrying about correctness (specially for infrequent corner cases). Correctness is guaranteed by the correctness substrate which uses *token counting rules* to ensure coherence and *persistent requests* to avoid starvation.

The main observation of the token framework is that simple token counting rules can ensure that the memory system behaves in a coherent manner. The following token counting rules are introduced in [67]:

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

- **Write Rule**: A component can write a block only if it holds all T tokens for

that block and has valid data. After writing the block, the owner token is set to dirty.

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

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

Considering these rules, we can relate token protocols with traditional MOESI protocols and define each of the states depending on the number of tokens that a processor has, as shown in table A.1.

The rules above ensure that cache coherence is maintained, but do not ensure forward progress. The performance policy of each particular token-based protocol defines how *transient requests* are used to exchange tokens. Transient requests are used to resolve most cache misses, but they are not guaranteed to succeed (for example, if transient requests are received in a different order by different coherence nodes).

Table A.1: Correspondence of token counting states with MOESI states

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

Token coherence avoids starvation by issuing a persistent request whenever a processor detects potential starvation, using a timeout. Persistent requests, unlike transient requests which are issued most times, are guaranteed to eventually succeed. To ensure this, each token protocol must define how it deals with several pending persistent requests. The two best known approaches for this are using a centralized persistent arbiter [68] or using distributed persistent requests tables [70].

Token coherence can avoid both the need of a totally ordered network and the introduction of additional indirection caused by the directory in the common case of cache-to-cache transfers.

Token coherence provides the framework for designing several particular coherence protocols. The performance policy of a token based protocol is used to instruct the correctness substrate how to move tokens and data through the system. A few performance policies have been designed, amongst them *Token-using-broadcast* (TokenB) is a performance policy to achieve low-latency cache-to-cache transfer misses, although it requires more bandwidth than traditional protocols [68]. TokenD [67] uses the token coherence framework to emulate a directory protocol, and TokenM [67] uses destination-set prediction to multicast requests to a subset of coherence nodes that likely need to observe the request. In TokenM, the correctness substrate ensures forward progress when the prediction fails. Other examples of token-based protocols can be found in [71] and [72].

TokenCMP [70] is the token-based cache coherence protocol that we have used as a basis for FtTokenCMP (see chapter 4). TokenCMP has a performance policy similar to TokenB but targets hierarchical multiple CMP systems (M-CMPs). It uses a distributed arbitration scheme for persistent requests, which are issued after a single try using transient requests, to optimize the access to contended lines.

The distributed persistent request scheme employed by TokenCMP uses a

persistent request table at each coherence node. Each processor will be able to activate at most one persistent request at a time by broadcasting a persistent read request activation or a persistent write request activation. Once the request has been satisfied, the processor will broadcast a persistent request deactivation. To avoid livelock, a processor will not be able to issue a persistent request again until all the persistent requests issued by other processors before its last persistent request was deactivated have been deactivated too.

In TokenCMP, write-backs are done by simply sending the data and tokens to the L2 cache. If the L2 cache does not have an entry available for the data, it will forward the message to memory.

Appendix B

Base directory-based protocol

This chapter describes DirCMP, the directory-based cache coherence protocol that we have used as the basis for developing FtDirCMP (see chapter 5). DirCMP is one of the protocols included in the default distribution of Multifacet GEMS (see section 7).

In this thesis, we assume a single-chip CMP system built using a number of tiles (see section 3.1). Each tile contains a processor, private L1 data and instruction caches, a bank of L2 cache, and a network interface. The L2 cache is logically shared by all cores but it is physically distributed among all tiles. DirCMP also supports a multiple CMP (M-CMP) by means of hierarchical directories, although we have not considered such configuration because it is orthogonal to the fault tolerance measures that we propose.

DirCMP is a MOESI-based cache coherence protocol which uses an on-chip directory to maintain coherence between several private L1 caches and a shared non-inclusive L2 cache. Although the L2 cache is non-inclusive, the directory information kept at the L2 level is strictly inclusive. It also uses a directory cache in L2 for off-chip accesses.

Table B.1 shows a simplified list of the main types of messages used by DirCMP and a short explanation of their main function.

This cache coherence protocol uses per line busy states to defer requests to memory lines with outstanding requests. The L2 directory will attend only one request for each memory line at the same time for on-chip accesses, although several off-chip non-exclusive (*GetS*) requests can be processed in parallel.

A typical request in DirCMP works as follows: the L1 cache sends the request

Table B.1: Message types used by DirCMP

Type	Description
GetX	Request data and permission to write.
GetS	Request data and permission to read.
Put	Sent by the L1 to initiate a write-back.
WbAck	Sent by the L2 to let the L1 actually perform the write-back. 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 (although invalidation acknowledgments may still be pending).
Unblock	Informs the L2 or directory that the data has been received and the sender is now a sharer.
UnblockEx	Informs the L2 or directory 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.

to the tile which has the L2 bank that corresponds to the address of the memory line. When the L2 controller handles the request, it either responds with the data, forwards it to the current owner L1 cache or forwards it to the inter-chip directory. In the case of exclusive requests (*GetX*) it also sends invalidation messages (*Inv*) to the current sharers (if any). Invalidation acknowledgments (*Ack* messages) are received by the requestor, and the L2 informs it about the number of sharers in the response message (or in an additional *Ack* message in case of forwarded responses). Once the requestor receives the data and all the expected acknowledgments, it sends an *Unblock* or *UnblockEx* message to the L2 which then proceeds to handle the next pending request for that memory line (if any).

Also, it uses three-phase write-backs to coordinate write-backs and prevent races with the rest of requests. In this way, the L1 cache issues a *Put* message to the L2 directory which also indicates the current state of the block at L1 (shared, owned or exclusive). When the L2 directory handles the requests, it sends a

WbAck or *WbAckData* (depending on whether it already has valid data or not, respectively) which is then answered by the L1 with a *WbData* or *WbNoData*. If the memory line to be replaced was in owned or exclusive state but it is invalidated after the *Put* message is sent due to a request from another node that is handled before by the directory, it will send a *WbNack* message and the L1 directory will have to reissue the write-back request. Write-backs between L2 and memory are handled in the same way.

DirCMP implements 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.

DirCMP uses a full-map sharing code both at the level of the L2 directory and the main memory directory. Other (inexact) sharing codes could be employed with little changes to the protocol, but we have not considered them since it does not affect the impact of our fault tolerance measures.

DirCMP, as most directory-based protocols, assumes that the interconnection 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.

Appendix C

Message dependency chains of our directory-based protocols

One of the potentially most onerous additional hardware requirements imposed by our protocols is that they may need additional virtual networks for their correct implementation, as noted in sections 4.3 and 5.3.

The additional virtual networks (which are implemented by means of virtual channels) are used to prevent deadlocks in the interconnection network [32]. The particular details of how many virtual networks are required and how they are used depend on the particular topology, switching technique and routing technique employed and are outside of the scope of this work.

Deadlocks in the interconnection network appear when a message cannot advance because it requires a resource (buffer) which is currently held by another message which itself requires (directly or indirectly) a resource held by another stopped message. That is, deadlocks require cyclic dependencies between resources.

These cyclic dependencies can be avoided by means of ensuring that there are not cyclic dependencies between the messages that travel through the interconnection network and that a message that depends on another message[1] will never require a resource required by the first message. This can be achieved using different virtual networks to send each of the dependent messages. This dependency relation among messages is transitive, hence we need as many vir-

[1] That is, a message that cannot be sent until the first message has been received.

tual networks as the length of the longest message dependency chain that the coherence protocol can introduce.

Having as many virtual networks as stated in the above paragraph is a sufficient condition for preventing deadlock, but it may not be necessary if other conditions are met [32].

As an example, we have analyzed DirCMP and FtDirCMP from the point of view of their implementation using deterministic routing on a 2D-mesh. Figures C.1 and C.2 show the dependencies between messages in each protocol (some trivial or redundant dependency chains have been omitted for brevity). Message types also include the sender (as $MessageType_{Sender}$) since the exact role of most message types depends on the kind of node that sends it.

First thing that can be seen in both figures is that there are no cyclic dependencies between message types. Second, we can see that the longest message dependency chain for each protocol is the following:

DirCMP: $Gets_{L1} \rightarrow Gets_{L2} \rightarrow DataEx_{Mem} \rightarrow DataEx_{L2} \rightarrow UnblockEx_{L1} \rightarrow UnblockEx_{L2}$.

Length: 6 messages.

FtDirCMP: $Gets_{L1} \rightarrow Gets_{L2} \rightarrow DataEx_{Mem} \rightarrow DataEx_{L2} \rightarrow UnblockEx{+}AckO_{L1} \rightarrow UnblockEx{+}AckO_{L2} \rightarrow AckBD_{Mem}$.

Length: 7 messages.

Hence, FtDirCMP may require one additional virtual network than DirCMP to be implemented correctly.

On the other hand, while our protocols may make deadlock prevention at the interconnection network harder to achieve, they can also be used to simplify deadlock recovery, hence making deadlock prevention less critical. This is because deadlock situations can be recovered by discarding some of the messages involved in the cycles that cause the deadlock. Since our protocols can tolerate the loss of any message, the interconnection network can decide to drop any message when a potential deadlock is detected as long as this is done infrequently enough so that all messages are eventually delivered without livelock.

Figure C.1: Message dependency chains for the DirCMP protocol

Figure C.2: Message dependency chains for the FtDirCMP protocol

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture (ISCA-15)*, pages 280–298, June 1988.

[3] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P. Jouppi, and James E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA-34)*, June 2007.

[4] Nidhi Aggarwal, Kewal Saluja, James E. Smith, Partha Ranganathan, Norman P. Jouppi, and George Krejci. Motivating commodity multi-core processor design for System-Level error protection. In *Proceedings of the 3rd Workshop on Silicon Errors in Logic System Effects (SELSE-3)*, April 2007.

[5] A. Ahmed, P. Conway, B. Hughes, and F. Weber. AMD Opteron™ shared-memory MP systems. In *Proceedings of the 14th HotChips Symposium (HotChips-14)*, August 2002.

[6] Rana E. Ahmed, Robert C. Frazier, and Peter N. Marinos. Cache-aided rollback error recovery (CARER) algorithm for shared-memory multiprocessor systems. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing Systems (FTCS-20)*, pages 82–88, June 1990.

[7] Muhammad Ali, Michael Welzl, and Sven Hessler. A fault tolerant mechanism for handling permanent and transient failures in a network on chip.

In *Proceedings of the 4th International Conference on Information Technology (ITNG 2007)*, pages 1027–1032, 2007.

[8] Todd M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-32)*, pages 196–207, 1999.

[9] David H. Bailey. FFTs in external or hierarchical memory. *The Journal of Supercomputing*, 4(1):23–35, March 1990.

[10] 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*, 45(10):1101–1115, October 1996.

[11] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, pages 282–293, June 2000.

[12] W. Bartlett and L. Spainhower. Commercial fault tolerance: a tale of two systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, 2004.

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

[14] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 processor: a 64-core SoC with mesh interconnect. In *Proceedings of the Solid-State Circuits Conference (Digest of Technical Papers) (ISSCC 2008)*, pages 88–98, 2008.

[15] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. Nonstop advanced architecture. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN 2005)*, pages 12–21, 2005.

[16] Philip A. Bernstein. Sequoia: a fault-tolerant tightly coupled multiprocessor for transaction processing. *Computer*, 21(2):37–45, 1988.

[17] D. Bertozzi, L. Benini, and G. de Micheli. Low power error resilient encoding for on-chip data buses. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2002)*, page 102, 2002.

[18] Jason A. Blome, Shantanu Gupta, Shuguang Feng, and Scott Mahlke. Cost-efficient soft error protection for embedded microprocessors. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2006)*, pages 421–431, 2006.

[19] Shekhar Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[20] Fred A. Bower, Paul G. Shealy, Sule Ozev, and Daniel J. Sorin. Tolerating hard faults in microprocessor array structures. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN 2004)*, page 51, 2004.

[21] M. Bozyigit and M. Wasiq. User-level process checkpoint and restore for migration. *SIGOPS Operating Systems Review*, 35(2):86–96, 2001.

[22] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *SIGOPS Operating Systems Review*, 29(5):1–11, 1995.

[23] Michael L. Bushnell and Vishwani D. Agrawal. *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*. Springer, November 2000.

[24] Harold W. Cain, Mikko H. Lipasti, and Ravi Nair. Constraint graph analysis of multithreaded programs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT-12)*, pages 4–14, 2003.

[25] Kaiyu Chen, Sharad Malik, and Priyadarsan Patra. Runtime validation of memory ordering using constraint graph checking. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA-14)*, pages 415–426, 2008.

[26] 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 *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, pages 3–14, February 2006.

[27] D. J. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.

[28] Crescenzo D'Alessandro, Delong Shang, Alex Bystrov, Alex Yakovlev, and Oleg Maevsky. Multiple-Rail Phase-Encoding for NoC. In *Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems*, page 107, 2006.

[29] Matteo Dall'Osso, Gianluca Biccari, Luca Giovannini, Davide Bertozzi, and Luca Benini. ×pipes: a latency insensitive parameterized network-on-chip architecture for multi-processor SoCs. In *Proceedings of the 21st International Conference on Computer Design*, page 536, 2003.

[30] William J. Dally, Larry R. Dennison, David Harris, Kinhong Kan, and Thucydides Xanthopoulos. Architecture and implementation of the reliable router. In *Proceedings of the Hot Interconnects Symposium II*, pages 122—133, 1994.

[31] Chunjie Duan and Anup Tirumala. Analysis and avoidance of cross-talk in on-chip buses. In *Proceedings of the the 9th Symposium on High Performance Interconnects*, page 133, 2001.

[32] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers, Inc., 2002.

[33] Tudor Dumitraș, Sam Kerner, and Radu Mărculescu. Towards on-chip fault-tolerant communication. In *Proceedings of the 2003 Conference on Asia South Pacific Design Automation (ASP-DAC 2003)*, pages 225–232, January 2003.

[34] Eric Dupont, Michael Nicolaidis, and Peter Rohr. Embedded robustness IPs for Transient-Error-Free ICs. *IEEE Design & Test of Computers*, 19(3):56–70, 2002.

[35] A. Ejlali and B.M. Al-Hashimi. SEU-hardened energy recovery pipelined interconnects for on-chip networks. In *Proceedings of the 2nd International Symposium on Networks-on-Chip (NoCS-2)*, pages 67–76, April 2008.

[36] Ricardo Fernández-Pascual, José M. García, Manuel E. Acacio, and José Duato. A low overhead fault tolerant coherence protocol for CMP architectures. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA-13)*, pages 157–168, February 2007.

[37] Ricardo Fernández-Pascual, José M. García, Manuel E. Acacio, and José 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.

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

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

[40] Arthur Pereira Frantz, Fernanda Lima Kastensmidt, Luigi Carro, and Erika Cota. Dependable network-on-chip router able to simultaneously tolerate soft errors and crosstalk. In *Proceedings of the 2006 IEEE International Test Conference (ITC 2006)*, pages 1–9, 2006.

[41] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA-17)*, pages 15–26, 1990.

[42] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Van Doren. Architecture and design of AlphaServer GS320. *ACM SIGPLAN Notices*, 35(11):13–24, 2000.

[43] Brian T. Gold, Jangwoo Kim, Jared C. Smolens, Eric S. Chung, Vasileios Liaskovitis, Eriko Nurvitadhi, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Truss: a reliable, scalable server architecture. *IEEE Micro*, 25(6):51–59, 2005.

[44] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-30)*, pages 98–109, 2003.

[45] Shantanu Gupta, Shuguang Feng, Amin Ansari, Jason Blome, and Scott Mahlke. The StageNet fabric for constructing resilient multicore systems. In *Proceedings of the 41nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-41)*, pages 141–151, 2008.

[46] Shantanu Gupta, Shuguang Feng, Amin Ansari, Jason Blome, and Scott Mahlke. StageNetSlice: a reconfigurable microarchitecture building block for resilient CMP systems. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2008)*, pages 1–10, 2008.

[47] Crispín Gómez, María E. Gómez, Pedro López, and José Duato. An efficient switching technique for NoCs with reduced buffer requirements. In *Proceedings of the 14th IEEE International Conference on Parallel and Distributed Systems (ICPADS-14)*, pages 713–720, 2008.

[48] Crispín Gómez, María E. Gómez, Pedro López, and José Duato. Reducing packet dropping in a bufferless NoC. In *Proceedings of the 14th international Euro-Par conference on Parallel and Distributed Processing (Euro-Par 2008)*, pages 899–909, 2008.

[49] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, Honggo Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA-31)*, pages 102–113, June 2004.

[50] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE MICRO Magazine*, 20(2):71–84, March-April 2000.

[51] Rajamohana Hegde and Naresh R. Shanbhag. Toward achieving energy efficiency in presence of deep submicron noise. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(4):379–391, 2000.

[52] D.B. Hunt and P.N. Marinos. A general purpose cache-aided rollback error recovery (CARER) technique. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing Systems (FTCS-17)*, pages 170–175, 1987.

[53] Natalie Enright Jerger, Li-Shiuan Peh, and Mikko Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence. In *International Symposium on Microarchitecture*, October 2008.

[54] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, 2003.

[55] Jongman Kim, Dongkook Park, Chrysostomos Nicopoulos, N. Vijaykrishnan, and Chita R. Das. Design and analysis of an NoC architecture from performance, reliability and energy perspective. In *Proceedings of the 2005 ACM Symposium on Architecture for Networking and Communications Systems (ANCS 2005)*, pages 173–182, 2005.

[56] Ki Wook Kim, Kwang Hyun Baek, Naresh Shanbhag, C. L. Liu, and Sung Mo Kang. Coupling-driven signal encoding scheme for low-power interface design. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-aided Design (ICCAD 2000)*, pages 318–321, November 2000.

[57] Brent A. Kingsbury and John T. Kline. Job and process recovery in a UNIX-based operating system. In *USENIX Conference Proceedings*, pages 355–364, 1989.

[58] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann, 2007.

[59] Christopher LaFrieda, Engin Ipek, Jose F. Martinez, and Rajit Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proceedings of the 2007 International Conference on Dependable Systems and Networks (DSN 2007)*, pages 317–326, 2007.

[60] C.R. Landau. The checkpoint mechanism in KeyKOS. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 86–91, 1992.

[61] Teijo Lehtonen, Pasi Liljeberg, and Juha Plosila. Online reconfigurable self-timed links for fault tolerant NoC. *VLSI Design*, 2007, May 2007.

[62] Lin Li, N. Vijaykrishnan, Mahmut Kandemir, and Mary Jane Irwin. Adaptive error protection for energy efficiency. In *Proceedings of the 2003*

IEEE/ACM International Conference on Computer-aided Design (ICCAD 2003), pages 2–8, 2003.

[63] Man-Lap Li, R. Sasanka, S.V. Adve, Yen-Kuang Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC 2005)*, pages 34–45, October 2005.

[64] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical report, University of Winsconsin, 1997.

[65] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: a full system simulation platform. *Computer*, 35(2):50–58, 2002.

[66] 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*, 12(3):299–311, March 2004.

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

[68] Milo M.K. Martin, Mark D. Hill, and David A. Wood. Token coherence: Decoupling performance and correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-30)*, pages 182–193, June 2003.

[69] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33(4):92–99, September 2005.

[70] 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 *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, pages 328–339, February 2005.

[71] Michael R. Marty and Mark D. Hill. Coherence ordering for ring-based chip multiprocessors. In *Proceedings of the 39nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-39)*, pages 309–320, 2006.

[72] Michael R. Marty and Mark D. Hill. Virtual hierarchies to support server consolidation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA-34)*, pages 46–56, June 2007.

[73] Yoshio Masubuchi, Satoshi Hoshina, Tomofumi Shimada, Hideaki Hirayama, and Nobuhiro Kato. Fault recovery mechanism for multiprocessor servers. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing Systems (FTCS-27)*, pages 184–193, June 1997.

[74] Cameron McNairy and Rohit Bhatia. Montecito: a dual-core, dual-thread Itanium processor. *IEEE Micro*, 25(02):10–20, 2005.

[75] Mojtaba Mehrara, Mona Attariyan, Smitha Shyam, Kypros Constantinides, Valeria Bertacco, and Todd Austin. Low-cost protection for SER upsets and silicon defects. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2007)*, pages 1146–1151, 2007.

[76] Albert Meixner, Michael E. Bauer, and Daniel J. Sorin. Argus: low-cost, comprehensive error detection in simple cores. *IEEE Micro*, 28(1):52–59, 2008.

[77] Albert Meixner and Daniel J. Sorin. Dynamic verification of sequential consistency. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32)*, pages 482–493, 2005.

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

[79] Albert Meixner and Daniel J. Sorin. Error detection via online checking of cache coherence with token coherence signatures. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA-13)*, pages 145–156, February 2007.

[80] Avi Mendelson and Neeraj Suri. Designing high-performance & reliable superscalar architectures: the out of order reliable superscalar (O3RS)

approach. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (DSN 2000)*, pages 473–481, 2000.

[81] Subhasish Mitra, Ming Zhang, N. Seifert, T.M. Mak, and Kee Sup Kim. Soft error resilient system design through error correction. In *Proceedings of the 2006 IFIP International Conference on Very Large Scale Integration*, pages 332–337, 2006.

[82] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38, April 1965.

[83] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. LogTM: log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, pages 254–265, 2006.

[84] Christine Morin, Alain Gefflaut, Michel Banâtre, and Anne-Marie Kermarrec. COMA: an opportunity for building fault-tolerant scalable shared memory multiprocessors. *SIGARCH Computer Architecture News*, 24(2):56–65, 1996.

[85] Christine Morin, Anne-Marie Kermarrec, Michel Banâtre, and Alain Gefflaut. An efficient and scalable approach for implementing fault-tolerant DSM architectures. *IEEE Transactions on Computers*, 49(5):414–430, 2000.

[86] Thomas Moscibroda and Onur Mutlu. Transactional memory coherence and consistency. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA-36)*, June 2009.

[87] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The soft error problem: an architectural perspective. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, February 2005.

[88] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. *SIGARCH Computer Architecture News*, 30(2):99–110, 2002.

[89] Shubu Mukherjee. *Architecture design for soft errors*. Morgan Kaufmann, February 2008.

[90] Srinivasan Murali, Theocharis Theocharides, N. Vijaykrishnan, Mary Jane Irwin, Luca Benini, and Giovanni De Micheli. Analysis of error recovery schemes for networks on chips. *IEEE Design and Test of Computers*, 22(5):434–442, 2005.

[91] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReViveI/O: efficient handling of I/O in highly-available rollback-recovery servers. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, pages 200–211, 2006.

[92] Simon Ogg, Bashir Al-Hashimi, and Alex Yakovlev. Asynchronous transient resilient links for NoC. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 209–214, 2008.

[93] Jonathan M. Owen, Mark D. Hummel, Derrick R. Meyer, and James B. Keller. United states patent: 7069361 - System and method of maintaining coherency in a distributed communication system, June 2006.

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

[95] Tudor Dumitras Paul Bogdan and Radu Marculescu. Stochastic communication: a new paradigm for fault-tolerant networks-on-chip. *VLSI Design*, page 17, 2007.

[96] 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 *Proceedings of the IEEE Computer society Annual Symposium on VLSI*, pages 46–51, February 2004.

[97] James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[98] J.B. Postel. RFC 793: Transmission control protocol, September 1981.

[99] Dhiraj K. Pradhan, editor. *Fault-tolerant computer system design*. Prentice-Hall, Inc., 1996.

[100] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. ReVive: cost-effective architectural support for rollback. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA-29)*, pages 111–122, May 2002.

[101] Moinuddin K. Qureshi, Onur Mutlu, and Yale N. Patt. Microarchitecture-based introspection: a technique for transient-fault tolerance in microprocessors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN 2005)*, pages 434–443, 2005.

[102] Sumant Ramprasad, Naresh R. Shanbhag, and Ibrahim N. Hajj. A coding framework for low-power address and data busses. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(2):212–221, 1999.

[103] M.W. Rashid and M.C. Huang. Supporting highly-decoupled thread-level redundancy for parallel programs. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA-14)*, pages 393–404, February 2008.

[104] Joydeep Ray, James C. Hoe, and Babak Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-34)*, pages 214–224, 2001.

[105] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. *SIGARCH Computer Architecture News*, 28(2):25–36, 2000.

[106] Bogdan F. Romanescu and Daniel J. Sorin. Core cannibalization architecture: improving lifetime chip performance for multicore processors in the presence of hard faults. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT-17)*, pages 43–51, 2008.

[107] Alberto Ros, Manuel E. Acacio, and José M. García. DiCo-CMP: Efficient cache coherency in tiled CMP architectures. In *Proceedings of the 22nd International Parallel & Distributed Processing Symposium (IPDPS 2008)*, pages 1–11, April 2008.

[108] Eric Rotenberg. AR-SMT: a microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing Systems (FTCS-29)*, pages 84–91, 1999.

[109] M. Russinovich and Z. Segall. Fault tolerance for off-the-shelf applications and hardware. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing Systems (FTCS-25)*, pages 67–71, 1995.

[110] Kenneth L. Shepard and Vinod Narayanan. Noise in deep submicron digital design. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design (ICCAD 1996)*, pages 524–531, 1996.

[111] Premkishore Shivakumar, Stephen W. Keckler, Charles R. Moore, and Doug Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Proceedings of the 21st International Conference on Computer Design (ICCD 2003)*, pages 481–488, 2003.

[112] Timothy J. Slegel, Robert M. Averill III, Mark A. Check, Bruce C. Giamei, Barry W. Krumm, Christopher A. Krygowski, Wen H. Li, John S. Liptay, John D. MacDougall, Thomas J. McPherson, Jennifer A. Navarro, Eric M. Schwarz, Kevin Shum, and Charles F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[113] Jared C. Smolens, Brian T. Gold, Babak Falsafi, and James C. Hoe. Reunion: complexity-effective multicore redundancy. In *Proceedings of the 39nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-39)*, pages 223–234, 2006.

[114] Jared C. Smolens, Brian T. Gold, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Fingerprinting: bounding soft-error detection latency and bandwidth. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-11)*, pages 224–234, 2004.

[115] Jared C. Smolens, Jangwoo Kim, James C. Hoe, and Babak Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proceedings of the 37nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-37)*, pages 257–268, 2004.

[116] Daniel J. Sorin, Mark D. Hill, and David A. Wood. Dynamic verification of end-to-end multiprocessor invariants. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 281–290, 2003.

[117] 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 *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA-29)*, pages 123–134, May 2002.

[118] P.P. Sotiriadis and A. Chandrakasan. Low power bus coding techniques considering inter-wire capacitances. In *Proceedings of the IEEE 2000 Custom Integrated Circuits Conference (CICC 2000)*, pages 507–510, 2000.

[119] L. Spainhower and T. A. Gregg. IBM S/390 parallel enterprise server G5 fault tolerance: a historical perspective. *IBM Journal of Research and Development*, 43(5/6):863–873, September 1999.

[120] Srinivasa R. Sridhara and Naresh R. Shanbhag. Coding for system-on-chip networks: a unified framework. In *Proceedings of the 41st Design Automation Conference (DAC-41)*, pages 103–106, 2004.

[121] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32)*, pages 520–531, 2005.

[122] Mircea R. Stan and Waine P. Burlesson. Bus-invert coding for low power I/O. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(1):49—58, 1995.

[123] Dwight Sunada, Michael Flynn, and David Glasco. Multiprocessor architecture using an audit trail for fault tolerance. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing Systems (FTCS-29)*, pages 40–47, June 1999.

[124] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-9)*, pages 257–268, 2000.

[125] C. Svensson. Optimum voltage swing on on-chip and off-chip interconnect. *Solid-State Circuits, IEEE Journal of*, 36(7):1108–1112, 2001.

[126] Dennis Sylvester, David Blaauw, and Eric Karl. ElastIC: an adaptive self-healing architecture for unpredictable silicon. *IEEE Design and Test of Computers*, 23(6):484–490, 2006.

[127] Michael B. Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Benjamin Greenwald, Henry Hoffman, Jae-Wook Lee, Paul Johnson, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matthew Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: a computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, May 2002.

[128] 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 *Proceedings of the Solid-State Circuits Conference (Digest of Technical Papers) (ISSCC 2008)*, 2007.

[129] Praveen Vellanki, Nilanjan Banerjee, and Karam S. Chatha. Quality-of-service and error control techniques for network-on-chip architectures. In *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pages 45–50, 2004.

[130] Bret Victor and Kurt Keutzer. Bus encoding to prevent crosstalk delay. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design (ICCAD 2001)*, pages 57–63, 2001.

[131] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA-29)*, pages 87–98, 2002.

[132] Christopher Weaver, Joel Emer, Shubhendu S. Mukherjee, and Steven K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31nd Annual International Symposium on Computer Architecture (ISCA-31)*, pages 264–, 2004.

[133] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Adapting to intermittent faults in multicore systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-13)*, pages 255–264, 2008.

[134] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Mixed-mode multicore reliability. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-14)*, pages 169–180, 2009.

[135] D. Wilson. *The STRATUS computer system*, pages 208–231. John Wiley & Sons, Inc., 1986.

[136] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA-22)*, pages 24–36, June 1995.

[137] Frédéric Worm, Paolo Ienne, Patrick Thiran, and Giovanni De Micheli. An adaptive low-power transmission scheme for on-chip networks. In *Proceedings of the 15th International Symposium On System Synthesis (ISSS-15)*, pages 92–100, 2002.

[138] Kung-Lung Wu, W. Kent Fuchs, and Janak H. Patel. Error recovery in shared memory multiprocessors using private caches. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):231–240, April 1990.

[139] H. Zhang, V. George, and J.M. Rabaey. Low-swing on-chip signaling techniques: effectiveness and robustness. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):264–272, 2000.

[140] Yan Zhang, John Lach, Kevin Skadron, and Mircea R. Stan. Odd/even bus invert with two-phase transfer for buses with coupling. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design (ISLPED 2002)*, pages 80–83, 2002.

[141] Heiko Zimmer and Axel Jantsch. A fault model notation and error-control scheme for switch-to-switch buses in a network-on-chip. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 188–193, 2003.