Technical Universisty of Valencia
Department of Systems Data Processing and Computers

Efficient Techniques to Provide Scalability for Token-based Cache Coherence Protocols

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

Doctor of Philosophy
(Computer Sciences)

Author

Blas Antonio Cuesta Sáez

Advisors

Antonio Robles Martínez
José Francisco Duato Marín

Valencia, 2009

Acknowledgments

"Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away"

Eric S. Raymond, The Cathedral and the Bazaar

I have received a lot of support, encouragement, and help throughout all these years in university. Many of the people I have met during these years have greatly influenced my research and changed me as a person.

First and foremost I thank my parents and my brother for their support, encouragement, constant interest in my life, and patience throughout all this time; they have always been there for me.

My advisers, Antonio Robles and José Duato, have had a great influence on me. The depth of knowledge that I have gained due to my numerous meetings with them is incredible. I have learned a lot about computer architecture and interconnection networks, but they have not been the only things, as they taught me about many aspects of performing research, communicating with others, delivering speeches, and many other related aspects. Most importantly, I have learned a great deal by simply observing their excellent example and behavior. I do not have any doubt that my research and non-research life has forever been changed by their advice. I specially thank Antonio Robles for his dedication and, above all, his patience, because like all true "*manchegos*", sometimes I may be too stubborn.

Many of the other faculty members in the Parallel Architecture Group have helped me too. I have benefited from interacting with them. Julio Sahuquillo, José Flich, Pedro López, and Federico Silla all provided useful insight into my dissertation research and provided valuable feedback during the process.

Likewise, I would like to thank María Engracia Gómez, Vicente Santonja, and Elvira Baydal because they all contributed in making me feel comfortable and integrated in the group.

I have met many interesting students while in university. Although I can not possibly mention everyone who has enriched my experience or provided moral support, I wish to specifically thank a few individuals. Gaspar Mora, Crispín Gómez, Francisco Gilabert, Andres Mejía, José Miguel Montañana, Rafa Ubal, Carles Hernández, Héctor Montaner, and David Yuste have unselfishly helped me in different aspects of my research and I would like to highlight their support. Furthermore, I would like to thank Ricardo Marín for his constant availability to help me to resolve each one of the numerous problems that I have faced

Gaspar and I created a fantastic hobby called "football Xtreme". I want to thank Gaspar for his help in creating and maintaining this hobby alive and for introducing me to such a great group of people. "Football Xtreme" has been very useful for me to relieve stress, keep fit, and have fun.

I could not finish without thanking all my English teachers at the American Institute, specially Meredith and Dana, who had to deal with strange questions.

Finally, I thank Virtutech AB for their support of Simics and for providing us with a license server. I also want to thank those individuals who paid taxes and, therefore, helped me during my time in university. This work was supported by the Spanish program CONSOLIDER-INGENIO 2010 under Grant CSD2006-00046, by the Spanish CICYT under Grant TIN2006-15516-C04-01, the JCC de Castilla-La Mancha under Grant PBC-05-007-2, and by the European Commission in the context of the SARC integrated project #27648 (FP6).

Contents

List of Figures

List of Tables

Abstract

Token-based cache coherence protocols simultaneously capture the best attributes of the two predominant approaches to coherence: low-latency cache misses (like in snooping-based protocols) and no reliance on totally-ordered interconnects (like in directory-based protocols). The key aspect to get those features is the use of requests that do not need to be put in order. Unfortunately, unordered requests present a serious problem since, in case of contention, they generate protocol races, which may prevent them from succeeding in solving cache misses. Therefore, to eliminate races, ensure the completion of all requests, and eventually resolve all cache misses, protocols based on tokens require the use of a starvation prevention mechanism.

The main drawbacks of token-based protocols are caused by the starvation prevention mechanism. Thus, although several alternative mechanisms have been proposed so far, all of them suffer from (1) being too inefficient, (2) requiring storage resources at each node that grow proportionally to the system size, and (3) requiring the use of broadcast messages. These problems compromise the scalability of token-based protocols because, as the system size increases, the protocol performs worse, the storage requirements at each node grows significantly, and the interconnect is more and more flooded with broadcast messages. However, these are not the only problems that compromise the protocol scalability, since token-based protocols require the use of non-silent invalidations, which threaten the scalability too.

As long as shared-memory multiprocessors include an increasingly number of nodes, the use of efficient and scalable cache coherence protocols is required. Therefore, the contributions of this dissertation aim to improve the performance and scalability of token-based protocols by tackling the weakest

aspects of the starvation prevention mechanisms and the non-silent invalidations. The most important contribution is the observation that simple routing algorithms can naturally avoid protocol races in cache coherence protocols that do not rely on either totally-ordered interconnects or directories to put requests in order. By using routing techniques based on ordered paths, we can easily create an efficient and scalable starvation prevention mechanism for token-based protocols. Additionally, those routing strategies can help us to develop techniques that improve the mechanism scalability by (1) minimizing its storage requirements and (2) considerably reducing the traffic overhead caused by broadcast messages. Besides, one of these techniques can be readily adapted to improve the problem of non-silent invalidations.

Resumen

Los protocolos de coherencia de caché basados en *tokens* son capaces de ofrecer simultáneamente las principales ventajas de los dos esquemas que tradicionalmente se han utilizado para implementar los protocolos de coherencia: baja latencia en los fallos de caché (como en los protocolos basados en *snooping*) y no dependencia de redes de interconexión totalmente ordenadas (como en los protocolos basados en directorio). Los protocolos basados en *tokens* pueden aunar esas características gracias en gran parte a la utilización de peticiones no ordenadas. Desafortunadamente, las peticiones no ordenadas introducen un problema nuevo, ya que, en caso de competir por un mismo bloque de memoria, pueden generar carreras de protocolo, lo que impide la resolución de los fallos de caché. Para eliminar las carreras de protocolo y poder asegurar que todas las peticiones son finalmente completadas (asegurando así la resolución de todos los fallos de caché), los protocolos basados en *tokens* requieren la utilización de un mecanismo de prevención de inanición.

Los problemas más significativos de los protocolos de coherencia basados en *tokens* son causados principalmente por el mecanismo de prevención de inanición. Aunque hasta ahora se han propuesto varias alternativas para implementar estos mecanismos, siguen presentando problemas importantes: primero, son demasiado ineficientes; segundo, consumen unos recursos de almacenamiento en cada nodo que crecen proporcionalmente con el tamaño del sistema; y tercero, requieren la utilización de mensajes *broadcast*. Todos estos problemas comprometen la escalabilidad de los protocolos basados en *tokens* porque, conforme crece el tamaño del sistema, el rendimiento del protocolo empeora, la cantidad de recursos de almacenamiento requeridos en cada nodo crece significativamente y el tráfico en la red de interconexión crece de forma

exponencial debido a los mensajes *broadcast*, lo que causa su congestión. Sin embargo, éstos no son los únicos problemas serios que comprometen la escalabilidad del protocolo, ya que ésta también se ve afectada por la utilización de invalidaciones no silenciosas (*non-silent invalidations*).

Mientras continue la tendencia a aumentar el número de nodos en los sistemas multiprocesadores de memoria compartida, se seguirá requiriendo la utilización de protocolos de coherencia de caché que sean eficientes y escalables. Teniendo en cuenta ésto, las contribuciones aportadas por esta tesis van dirigidas a mejorar el rendimiento y la escalabilidad de los protocolos de coherencia basados en *tokens*. Para conseguirlo, esta tesis aborda tanto los aspectos más negativos de los mecanismos de prevención de inanición como los de las invalidaciones no silenciosas. La contribución más destacada es la observación de que ciertos algoritmos de encaminamiento pueden evitar de forma natural que los protocolos que no están basados en redes totalmente ordenadas ni en directorios generen carreras de protocolo. Así, mediante la utilización de técnicas de encaminamiento basadas en caminos ordenados, podemos crear fácilmente un mecanismo de prevención de inanición eficiente y flexible. Además, esos algoritmos de encaminamiento nos permiten desarrollar técnicas que mejoren la escalabilidad del protocolo, reduciendo tanto los recursos de almacenamiento requeridos por el mecanismo de prevención de inanición como el tráfico de red generado por los mensajes *broadcast*. Adicionalmente, estas técnicas se pueden utilizar también para mejorar la parte de las invalidaciones no silenciosas.

Resum

Els protocols de coherència de caché basats en *tokens* són capaços de capturar simultàniament els principals avantatges dels dos esquemes que tradicionalment s'han utilitzat per a assegurar coherència: baixa latència en les fallades de caché (com en els protocols basats en *snooping*) i no dependència de xarxes d'interconnexió totalment ordenades (com en els protocols basats en directori). Els protocols basats en *tokens* poden unir eixes característiques gràcies en gran part a la utilització de peticions desordenades. Desafortunadament, les peticions desordenades introduïxen un problema nou ja que, en cas de competir per un mateix bloc de memòria, poden generar carreres de protocol, la qual cosa impedix la resolució de les fallades de caché. Per eliminar les carreres de protocol i poder assegurar que totes les peticions són finalment completades (assegurant així la resolució de totes les fallades de caché), els protocols basats en *tokens* requerixen la utilització d'un mecanisme de prevenció d'inanició.

Els problemes més significatius dels protocols de coherència basats en *tokens* són causats principalment pel mecanisme de prevenció d'inanició. Encara que fins ara s'han proposat diverses alternatives per a implementar aquests mecanismes, continuen presentant problemes seriosos: primer, són massa ineficients; segon, consumixen uns recursos d'emmagatzemament a cada node que creixen proporcionalment amb la grandària del sistema; i tercer, requerixen la utilització de missatges *broadcast*. Tots aquests problemes comprometen l'escalabilitat dels protocols basats en *tokens* perquè, conforme creix la grandària del sistema, el rendiment del protocol empitjora, la quantitat de recursos d'emmagatzemament requerits a cada node creix significativament i la xarxa d'interconnexió es congestiona cada vegada més a causa dels missatges *broadcast*. Malgrat això, aquests no són els únics problemes seriosos que

comprometen l'escalabilitat del protocol, ja que aquesta també es veu afectada per la utilització d'invalidacions no silencioses (*non-silent invalidations*).

Mentres seguisca la tendència d'augmentar el nombre de nodes en els sistemes multiprocessadors de memòria compartida, es continuarà requerint la utilització de protocols de coherència de caché que siguen eficients i escalables. Tenint en compte això, les contribucions aportades per aquesta tesi tenen com a objectiu la millora del rendiment i l'escalabilitat dels protocols de coherència basats en *tokens*. Per a aconseguir-ho, aquesta tesi aborda tant els aspectes més negatius dels mecanismes de prevenció d'inanició com els de les invalidacions no silencioses. La contribució més destacada és l'observació que certs algoritmes d'acarrerament poden evitar de forma natural que els protocols que no estan basats en xarxes totalment ordenades ni en directoris generen carreres de protocol. Així, mitjançant la utilització de tècniques d'acarrerament basades en camins ordenats, podem crear fàcilment un mecanisme de prevenció d'inanició eficient i flexible. A més, aquests algoritmes d'acarrerament ens permeten desenvolupar tècniques que milloren l'escalabilitat del protocol i reduir tant els recursos d'emmagatzemament requerits pel mecanisme de prevenció d'inanició com el trànsit de xarxa generat pels missatges *broadcast*. Addicionalment, aquestes tècniques es poden utilitzar també per a millorar les invalidacions no silencioses.

Chapter 1

Introduction

This chapter briefly describes the context in which this dissertation is set (Section 1.1) and mentions the reasons that motivate it. We then define the objectives that this dissertation aims (Section 1.2) and we show the contributions that such objectives have originated (Section 1.3). We conclude this chapter by presenting the structure of this dissertation (Section 1.4).

1.1 Context and Motivation

Nowadays database and web servers are required for many different tasks, such as sending/receiving electronic mails, business applications in departments or companies (products of SAP or Oracle), workgroup environments (Lotus Notes), databases, client-server applications, or scientific applications. Those services are often provided by servers, being their performance a key matter due to the fact that they are increasingly a part of our daily lives. To a large extent, the performance of servers depends on the performance of microprocessors. Historically microprocessors have been improved by both increasing their frequency and increasing the work performed in each cycle. However, increasing the frequency can not longer deliver performance improvements due to energy, heat, and wire delay issues [11]. Consequently, mainstream microprocessor vendors are focused on improving the performance of servers by increasing the work performed in each cycle. To this end, shared-memory multiprocessor systems such as SMPs, clusters of SMPs, or chip multiprocessors

(CMPs) [61, 108, 68, 71, 57] have been proposed. A shared-memory multiprocessor consists of a number of processors accessing one or more shared memory modules. The processors can be physically connected to the memory modules in a variety of ways, but logically every processor is connected to every module. This kind of system requires a cache coherence protocol to coordinate the different caches distributed through the system as part of providing a consistent view of memory to processors. Cache coherence protocols determine how the blocks of the shared memory are transferred between processors, caches, and memory, which will directly influence the performance of shared-memory multiprocessors.

The number of processors that integrate shared-memory multiprocessors is rapidly increasing. For example, in 2005, Intel and AMD offered dual-core products [101] and two years later they offered quad-core products [100]. Meanwhile, in 2005, Sun offered 8-core CMPs [61] and, in 2008, it offered 16-core versions [62]. Besides, Sun began to offer multiprocessor severs with up to 4 processors and nowadays it offers 64-multiprocessor servers [105]. In 1996, SGI offered mid-range servers (SGI Origin 2000) where the number of processors ranged from 2-8 processors in the SGI Origin 2100 up to 32-128 processors in the SGI Origin 2800. In 2000, it offered servers with up to 512 processors (SGI Origin 3800) and, in 2006, they made them expandable to over 1000 processors (in a shared-memory environment). Therefore, all these examples show that it is conceivable that the number of processors continue increasing exponentially, at the rate of Moore's Law [86] over the next decade. In fact, Intel is trying to integrate 80 processors onto a single chip [55] and Berkeley researches suggest that future multiprocessor systems could contain thousands of processors [17]. Hence, while the number of processors continue increasing, implementing low-latency and scalable cache coherence protocols in shared-memory multiprocessors will be a key issue to scale performance.

Although a lot of different cache coherence protocols have been proposed, most of them are based on two different approaches: snooping and directory. On the one hand, *snooping-based* protocols [50] are those protocols that broadcast requests to all processors using a bus or bus-like interconnect that offers global order to requests. By ordered broadcast requests, processors can directly communicate between themselves, thereby minimizing the latency of

cache-to-cache misses. In addition, the ordered requests unambiguously resolve the contention for the same memory block, preventing the occurrence of protocol races. The main advantage of snooping-based protocols is that they provide low-latency cache-to-cache misses. However, to get it, this class of protocol requires the interconnect system to provide ordered broadcasts. The interconnects that exhibit this behavior are known as totally-ordered interconnects. The main disadvantage of protocol based on snooping is that they are restricted to systems with totally-ordered interconnects, which entails a problem because their applicability is limited. Besides, totally-ordered interconnects do not scale well [40], making snooping-based protocols only suitable for systems with a small number of processors.

On the other hand, *directory-based* protocols [9] send requests only to the home memory where the directory is located. The directory then responds with data or forwards the request to one or more processors. Like snooping-based, directory-based protocols also put requests in order. However, in this case, it is done by establishing an ordering point (the directory). Thus, the occurrence of protocol races is prevented. Since the directory provides order for requests, directory-based protocols do not require totally-ordered interconnects. Rather, they can be implemented in systems with low-latency interconnects, which widens their applicability. Besides, point-to-point interconnection networks are more scalable than totally-ordered interconnects, which makes directory protocols suitable for medium/large-sized systems. The worst aspect of directory-based protocols is that processors do not communicate directly between themselves, since requests are first sent to the directory. This entails a serious problem known as indirection because both a directory lookup and a third interconnect transversal (the path from the directory to nodes) are placed in the critical path of cache-to-cache misses, which considerably increases their latency.

Since snooping-based and directory-based protocols present opposed features (low-latency cache-to-cache misses against not reliance on totally-ordered interconnects), to try to simultaneously capture their best attributes, a new class of cache coherence protocol, *token-based* protocols, has been proposed recently. Hence, token-based protocols aim to offer low-latency cache-to-cache misses without relying on totally-ordered interconnects. To this end, in token-

based protocols requests are directly sent to all processors (or a subset) through a point-to-point interconnect. Thanks to this direct communication between processors, low latency cache-to-cache misses can be obtained. Nevertheless, notice that requests in token-based protocols are not ordered messages because neither totally-ordered interconnects (like in snooping protocols) nor ordering points (like in directory protocols) are used. As a result, requests contending for the same memory block may generate protocol races, which cause those requests to fail at resolving cache misses. To solve races and guarantee the completion of all cache misses, a starvation prevention mechanism must be used.

In spite of having proposed several implementations, the starvation prevention mechanism is the component which causes the major problems of token-based protocols:

- The starvation prevention mechanism is inflexible because it does not allow to simultaneously serve several requests while it is active.
- It overrides the component that provides low-latency cache-to-cache misses, preventing the starvation by an inefficient method.
- The most efficient implementation requires non-scalable storage structures at each system node, while the other implementations require additional components that introduce indirection.
- It is based on broadcast messages which are not suitable for medium/large-sized systems.
- Explicit acknowledgments are used, which increases the network traffic and delays the request service.

Besides those problems inherited from the starvation prevention mechanism, token-based protocols also present another one: the non-silent invalidations. When a node invalidates a copy of a memory block in its cache, it has to send an acknowledgment which contains the tokens and the memory block. Invalidations can be caused because of two reasons: evictions and write requests. In case of evictions, the non-silent invalidations do not entail a serious problem. However, the invalidations caused by write requests will be a

problem since the messages generated as a result of the invalidation are placed in the critical path of writes.

In conclusion, although token-based protocols integrate the main advantages of traditional protocols, they still present some problems that jeopardize their performance and scalability. These problems are mainly caused by the starvation prevention mechanism and by the non-silent invalidations. Therefore, to improve the scalability of token-based protocols (and their performance) those problems must be solved.

1.2 Objectives

Token-based protocols are the only cache coherence protocols that simultaneously can avoid both indirection and reliance on totally-ordered interconnects. Therefore, they seem to be the best approach to implement cache coherence in current systems. However, they still present some disadvantages which make them inefficient in medium and large systems, being the main objective of this dissertation to tackle all those problems by taking advantage of some well-known routing strategies commonly used in interconnection networks.

The first objective is to develop a more flexible and efficient starvation prevention mechanism which ensures cache miss resolution without having the necessity of overriding the component that provides low-latency cache misses, which in turn may contribute to improve the performance of the mechanism.

To avoid indirection, the starvation prevention mechanism requires the use of non-scalable storage requirements at each node. Although in small systems they may be acceptable, the tendency to increase the system size motivates our second objective. Thus, the second goal is to develop a strategy that resolves the lack of scalability of the storage structures required by the starvation prevention mechanism.

The requests used by token-based protocols can be based on broadcast or multicast messages. However, the starvation prevention mechanism always is based on broadcast messages, which makes it non-scalable. Therefore, the third objective of this dissertation is to design an alternative strategy which provides the starvation prevention mechanism with certain scalability in terms of traffic generation.

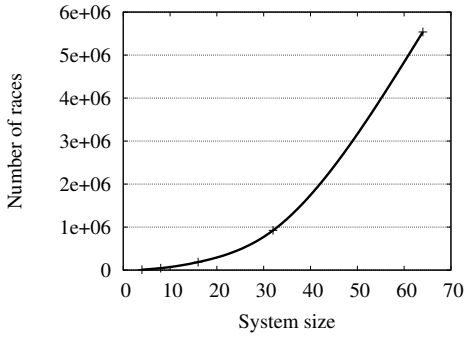


Figure 1.1: Protocol races according to the system size.

Finally, the last problem that we tackle is that caused by non-silent invalidations. Thus, the fourth objective is the development of a strategy that alleviates the problem of non-silent invalidations.

1.3 Thesis Contributions

The objectives pointed out in the previous section have originated several contributions, which are briefly described in this section. All the contributions but one target a different aspect of the starvation prevention mechanism given that it is responsible for the most relevant and serious troubles of token-based protocols. Besides, the other contribution targets the non-silent invalidation problem, which does not depend on the starvation prevention mechanism. Throughout this dissertation, we assume the use of Token Coherence [77] which is a framework to develop token-based protocols, when applied to a CC-NUMA environment. The starvation prevention mechanism used by Token Coherence is referred as the *persistent request mechanism*.

1.3.1 Priority Requests

In small systems, the starvation prevention mechanism required by token-based protocols will not be used frequently since the probability of contention among processors is low, thereby being generated few protocol races. Therefore, in small systems the performance of the whole protocol is hardly affected by the starvation prevention mechanism. However, as the system is getting

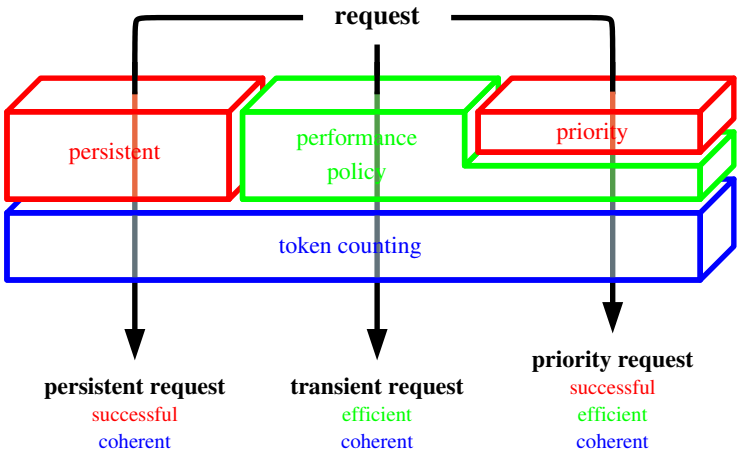


Figure 1.2: System components used by persistent requests, transient requests, and priority requests and their main properties.

larger, the probability of contention among processors increases and, consequently, the starvation prevention mechanism will be more and more used. This is shown in Figure 1.1, where it can be observed how the number of starvation situations varies exponentially regarding the system size. Thus, in medium/large systems the starvation prevention mechanism will significantly affect the overall performance.

Although several implementation options of persistent requests have been proposed up to now, all of them are too inefficient, mainly because they override the performance policy, which provides efficiency and low-latency, as shown in Figure 1.2. Besides, persistent requests are so strict that they do not allow the service of transient requests (requests commonly used in absence of starvation) while they are active, which is likely to cause unnecessary additional races. This two facts make the persistent request mechanism inefficient. Therefore, although persistent requests do not perform bad in small systems, in medium/large-sized systems they will seriously penalize the overall performance of Token Coherence.

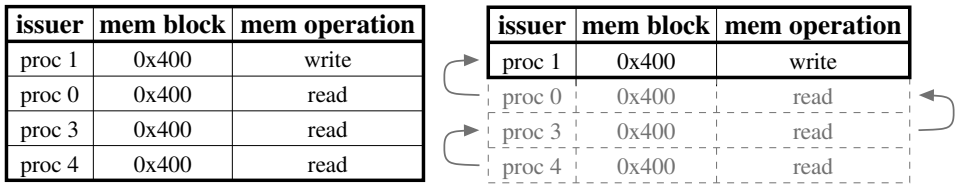
The primary contribution of Chapter 5 develops an efficient starvation prevention mechanism named *priority requests*. Like traditional protocols, the priority request mechanism relies on a global order of requests to avoid races. However, since we want to avoid both totally-ordered interconnects and

indirection overheads, priority requests use an alternative strategy to ensure in order reception. To this end, priority requests rely on routing algorithms to provide totally ordered requests. This is accomplished by sending requests through ordered paths. All priority requests routed through the same ordered path will be received in the same order by all processors. This is enough to prevent priority requests from causing protocol races and, therefore, their completion can be easily ensured. Nodes exploit the ordering properties of priority requests by completing them in the natural arrival order. Due to the fact that priority requests can not generate protocol races, there is no reason to override the performance policy. Thus, as shown in Figure 1.2, priority requests can use the performance policy, which allows to efficiently move tokens at all times while ensuring completion.

1.3.2 Limiting the Storage Requirements

Both persistent and priority request mechanisms require the system to maintain a table at each node to remember all the unsuccessful (or starved) requests. The size of those tables depends on the product of two factors: (1) the number of maximum outstanding requests per processor and (2) the system size (number of processors in the system). By limiting the number of outstanding requests to one, the size of tables is not very large, but it still depends on the system size, growing proportionally to the number of processors (shown in Figure 1.3(a)). This is a serious problem since currently there is a trend to increase the number of processors and that threatens the scalability of the whole protocol. Besides, those tables require associative search, increasing their access latency as the number of table entries increases.

The primary contribution of Chapter 6 develops a strategy to decouple the size of the starved request tables from the system size. To this end, tables only store information of a subset of all the starved requests. The information of those requests is replicated in all the tables so that all processors know the starved requests that need to be served as soon as possible. The other starved requests will not be able to be stored in the tables until the stored ones complete. The information about the not stored starved requests is not replicated in all nodes, but distributed through them. This simple technique allows to reduce (and limit) the size of tables, being able to reach single-entry



(a) size proportional to system size (b) single-entry table

Figure 1.3: Differences between a starved request table with as many entries as number of processors and a starved request table with just one entry.

tables at the expense of a slight performance degradation. Figure 1.3(b) shows a table with a single entry. The information in gray refers to the not stored starved requests. That information is distributed and not replicated. When the stored request completes, its issuer will inform to the issuer of the next request in the sequence (indicated by the arrows). Thus, when a node is informed about the completion of a stored request, it will be able to resend its request with total guarantee of storage.

1.3.3 Switch-based Packing Technique

Broadcast messages are a good solution for small systems. However, their bandwidth requirements increase quadratically to the system size. This entails a serious problem as the bandwidth provided by most of the point-to-point networks only increases linearly to the system size. Therefore, although broadcast messages may be a good solution in small systems, in medium/large systems they will require much more bandwidth than that provided by the network. This lack of scalability is clearly shown in Figure 1.4.

Token Coherence requires the use of broadcast messages (persistent/priority requests) to prevent starvation. As they are not scalable, Token Coherence only will be suitable for systems with a low number of processors. To improve this aspect of the protocol, Chapter 7 proposes a network switch-based strategy which increases the scalability of broadcasts. When only a few broadcasts are generated and the network has enough bandwidth to quickly process all of them, broadcast messages will hardly coincide in the input buffers of the traversed switches. However, when several broadcast messages are generated

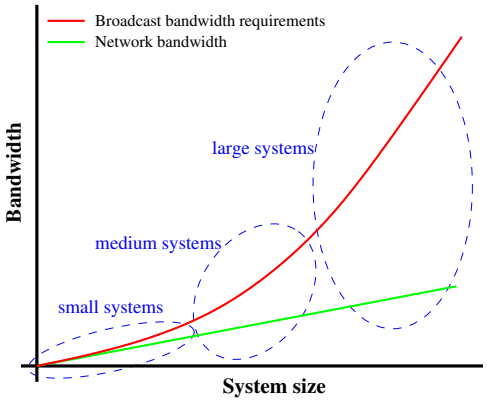


Figure 1.4: Comparison between the bandwidth provided by the interconnection network and the bandwidth required by broadcast messages depending on the system size.

and the network does not have enough bandwidth to quickly deliver all of them, they will be accumulated in the buffers of the traversed switches. The technique that we propose takes advantage of the fact that the broadcast messages accumulated at switches carry almost the same information. Therefore they can be packed in just one message. This packing of broadcast messages allows to drastically reduce the transmitted traffic. Besides, the more broadcast messages the nodes inject, the more packing the switches perform, which provides certain scalability to the protocol.

A secondary contribution of Chapter 7 develops a similar strategy to reduce the number of messages generated as response to non-silent invalidations. Given that, in case of an invalidation due to a write request, all the response messages are directed to the same processor, the idea is to concentrate (when possible) the tokens carried by several response messages in just one message. Thus, instead of having to wait for a message from each sharer, only few responses would be required.

1.3.4 Multicast Data Responses

Policies based on MOESI states select a processor which is the only one in charge of providing, at that time, a memory block to the processors that request it. Most systems assume the MOESI policy (or one based on a sub-

set/superset of its states) since it efficiently manages memory blocks. However, when several processors contend for the same memory block, the processor designated by the MOESI policy to provide such a block may become temporarily a bottleneck. This situation often occurs when the use of the starvation prevention mechanism is required, since starvation is usually caused by several nodes contending for the same block. This situation worsens as the number of processors involved in a starvation situation increases.

The primary contribution of Chapter 8 proposes a technique that resolves the aforementioned situation. Like some protocols that use a single message to invalidate several copies of a memory block, we introduce a new class of response message based on multicast techniques that allows to validate several copies of a memory block. In addition to considerably reducing the injected traffic, it reduces the latency to resolve the contention situations.

1.3.5 Summary

The main problems of token-based protocols are those inherited from the starvation prevention mechanism (inefficiency, non-scalable storage requirements, and need of broadcast messages) and the non-silent invalidations. The contributions of this dissertation aim to solve all these problems. In this sense, both the priority request mechanism (Chapter 5) and the multicast responses (Chapter 8) address the problem of inefficiency. The technique presented in Chapter 6 tackles the non-scalable storage requirements. Finally, Chapter 7 faces to the scalability problem of broadcast messages and non-silent invalidations.

1.4 Thesis Outline

This dissertation begins with this introductory and motivational chapter (Chapter 1), continues with a chapter that describes the fundamentals of cache coherence protocols and interconnection networks (Chapter 2). Chapter 3 includes the evolution and the main innovations on cache coherence protocols. Chapter 4 discusses the tools, methodology, and workloads used for the evaluation carried out in this thesis. Chapter 5 presents the priority request mechanism. Chapter 6 develops the strategy for limiting the storage requirements. Chap-

ter 7 introduces the switch-based packing technique. Chapter 8 presents the multicast responses. The dissertation ends with Chapter 9, which summarizes the proposals and identifies other advantages that are not evaluated here.

Chapter 2

Concepts and Background

This chapter describes the basics and terminology for understanding caches memories, interconnection networks, and the underlying problem of cache coherence in multiprocessor systems. Although this chapter reviews some concepts of cache coherence and interconnection networks, it is not an introduction to them. Rather, it is intended to give insight on the different concepts related to cache memories, which are assumed in this thesis. We refer the reader to the established textbooks on this topic for further background and introductory material (e.g., [96, 38, 53, 41]).

This chapter first discusses the need for caches (Section 2.1) and the problem of keeping the contents of caches coherent in shared-memory multiprocessor systems (Section 2.2). Next, we describe some strategies used by coherence protocols to address the coherence problem (Section 2.3) and some of the most widespread consistency models (Section 2.4). Finally, the chapter concludes with a description of various alternatives for implementing multiprocessor interconnection networks (Section 2.5).

2.1 Basis of Cache Memories

No one memory technology can supply all the memory needs of a computer since fast memories are usually low capacity memories (low bit density). As a consequence, they are expensive: cost per bit increases as access time decreases. Consequently, several memory types with very different physical pro-

perties placed at different levels of the memory hierarchy have to be used in typical computer systems. Main memory is a large (but slow) memory implemented with DRAM technology. To reduce the speed disparity between CPU and main memory, one or more intermediate small-sized memories called *caches* are used. The term cache refers to a fast intermediate memory within a larger memory system [109, 52]. Caches, which might be implemented with SRAM technology, directly address the Von Neumann bottleneck by providing the CPU with fast access to memory.

Caches store copies of items located in main memory. Memory words are stored in a *cache data memory* and are grouped into small pages called *cache blocks* or *lines*. The contents of the cache's data memory are thus copies of a set of main memory blocks. Each cache block is marked with its block address, referred to as a tag, so the cache knows to what part of the memory space the block belongs. The collection of tag addresses currently assigned to the cache is stored in the *cache tag memory*. Note that, for a cache to improve the performance of a computer, the time required to check tag addresses and access the cache's data memory must be much lower than the time required to access main memory.

When the CPU issues a memory address, the cache compares it to the contents of its tag memory. If a match (*hit*) occurs, the memory access is completed by the cache; otherwise (*miss*), a block that includes the addressed item is retrieved from main memory and placed into the cache. *Temporal locality* tells us that we are likely to need this word again in the near future, so it is useful to place it in the cache where it can be accessed quickly. *Spatial locality* tells us that there is a high probability that the other data in the block will be needed soon. Hence, because of locality principle and the higher speed of smaller memories, a memory hierarchy can substantially improve performance. A basic measure of this performance is the hit ratio, which is the fraction of all memory references that are satisfied by cache.

2.1.1 Block Placement Policy

When a block is retrieved from main memory, a *block placement* policy is used to know where the block can be placed into the cache. This policy influences when a tag address is presented to the cache, since it must be quickly compared

to the stored tags to determine whether a matching occurs. Depending on the restrictions on where a block can be placed, there exist three categories of cache organization:

- If each memory block has only one place where it can be allocated in the cache, the cache is said to be *direct mapped*. In this case, the cache is divided into sets, each of which stores a block. With direct mapping, each block in main memory is mapped into one specific block of cache. The main drawback of this organization is that the cache's hit ratio drops sharply if two or more frequently used blocks map onto the same region in the cache (known as collision), whereas the main advantage is its simplicity.

- If a block can be placed anywhere in the cache, the cache is said to be *fully associative*. Associative memories are also commonly known as *content-addressable memories* (CAMs). To implement fast tag comparison, the input tag can be compared simultaneously to all tags in the cache tag memory. The main disadvantage of this kind of memory is that they are expensive and complex.

- If a block can be placed in a restricted set of places in the cache, the cache is *set associative*. A set is a group of blocks in the cache. A block in main memory is first mapped onto a set, and then the block can be placed anywhere within that set. If there are n blocks in a set, the cache placement is called *n-way set associative*. This approach reaches a trade-off between the advantages and disadvantages of the two previous proposals. Thus, it is considered a reasonable compromise between the complex hardware needed for fully associative caches (which requires parallel searches of all tags), and the simple direct-mapped scheme. The main disadvantage is similar to that in direct mapped caches, since collisions may occur.

2.1.2 Replacement Policy

When a miss occurs, a cache block must be selected to be replaced with the block retrieved from main memory. The main advantage of direct mapped

policy is that hardware decisions are simplified since a replacement policy is not required: each block has only one place to be placed and only that block can be replaced. With fully associative or set-associative placement, there are many blocks to choose from on a miss. The most employed strategies for selecting the block to replace are:

- *Random.* The candidate block to replace is randomly selected.
- *Least-recently used (LRU).* Relying on the past to predict the future, the block replaced is the one that has been unused for the longest time.
- *First in, first out (FIFO).* Because LRU can be complicated to calculate, this approximates LRU by determining the oldest block rather than the least-recently used one.

2.1.3 Write Policy

Another important aspect of caches is the write policy. There exist two different strategies when a write is carried out: *write-through* and *write-back*. In write-through, the information is written to both the block in cache and to the block in main memory. This policy is easy to implement, and it guarantees that main memory never contains stale information. In write-back, the information is written only to the block in cache. This modified cache block is written to main memory only when it is replaced. This technique has the disadvantage of temporal inconsistency, that is, cache and main memory can have different data associated with the same physical address. In addition, the write-back technique complicates recovery from system failures. On the other hand, write-through results in more write cycles to main memory than write-back does.

To reduce the frequency of writing-back blocks on replacements, a feature called the *dirty bit* is commonly used. This status bit indicates whether the block is *dirty* (modified while in cache) or *clean* (not modified). If it is clean, the block is not written back on a miss, since identical information to the cache is found in main memory.

Table 2.1: Cache features in actual computers.

Model	L1 cache	L2 cache	L3 cache
Intel 386	off-die 64 KB	(none)	(none)
Intel 486	on-die 8 KB	(none)	(none)
Pentium MMX	split on-die 16 KB	(none)	(none)
Pentium Pro	split on-die 8 KB	cartridge 1 MB	(none)
Itanium 2	split on-die 16 KB	split on-die 1 MB	on-die 12 MB
Xeon MP	split on-die 8 KB	on-die 2 MB	on-die 16 MB
IBM Power 4	split on-die 96 KB	on-die 1 MB	off-die 256 MB
AMD Phenom	split on-die 64 KB	on-die 512 KB	on-die 2 MB

2.1.4 Structure

Table 2.1 illustrates some of the diversity of commercial cache types. As clock speeds separated from main memory speeds, fast and small cache memories began to be included to boost performance. Thus, early computers employed a single, multichip cache that occupied one level of the hierarchy between the CPU and main memory. These caches were external to the processor and located on the motherboard (some versions of the *386* processor could support up to 64 KB of external cache). Later, due to the feasibility of including part of the real memory space on a microprocessor chip and the growth in the size (but not in the speed) of main memory, more cache levels were introduced, which addressed the increase of the miss penalty. A Level 1 (L1) cache is an efficient way to implement an on-die memory. It was named like that to differentiate it from the Level 2 (L2) cache, which was still located on the motherboard (off-die). The L2 cache is slower than L1 cache, but it is much larger. In general, caches of levels close to the CPU are smaller, but faster than caches of higher levels. Hence, with the appearance of the *486* processors (and later in the *Pentium MMX*), an 8 KB cache began to be integrated directly into the CPU die. Later, the introduction of SDRAM to implement main memory

and the growing difference between the bus speed and the CPU clock speed caused on-motherboard cache to be only slightly faster than main memory, which forced a new evolution. Thus, some processors such as *Pentium Pro*, *Pentium II*, and the first *Pentiums III* incorporated the secondary cache into the same cartridge as the CPU, but out of the die.

The desirability of additional levels increases with the size of main memory. As main memory size increases further, the latency difference between main memory and the fastest cache becomes larger. This makes even more cache levels to be desirable (for example, a third level of cache). This level can be implemented on a separated chip from the CPU (the *IBM Power 4* series support up to 256 MB L3 cache off-chip) or incorporated in the same chip (*Itanium 2* incorporated a 12 MB L3 cache on-die, the *AMD Phenom* series of chips carries a 2 MB on-die L3 cache, and the *Intel Xeon MP* features 16 MB on-die L3 cache).

Multi-level caches can be classified in different types. A cache is said to be *strictly inclusive* when all data in L1 cache are also in L2 cache. Other processors (like the *AMD Athlon*) have *exclusive* caches, that is, a datum is either in L1 cache or in L2 cache, never in both.

Caches are also distinguished by the kind of information they store. An *instruction* or *I-cache* stores instructions only, while a *data* or *D-cache* stores data only. Separating the stored data in this way recognizes the different access behavior patterns of instructions and data. A cache that stores both instructions and data is referred to as *unified* (such as in the *PA-7100 LC* processors [2]). On the other hand, a *split cache* consists of two associated but largely independent units: an I-cache and a D-cache. While a unified cache is simpler, a split cache makes it possible to access programs and data concurrently.

2.2 Cache Coherence

Although the microprocessor performance has been improving at a rate of about 50% per year, it may be increasingly difficult that a single processor becomes fast enough to satisfy the applications demands for ever greater performance. An attractive solution can be the parallel machines, since they

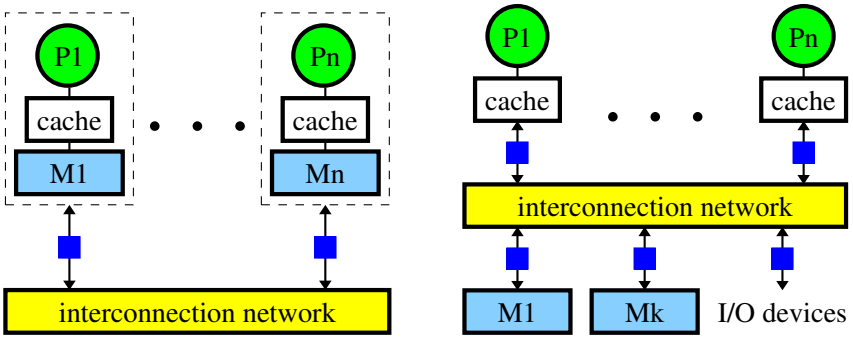


Figure 2.1: Examples of (a) multicomputer and (b) UMA architecture. Circles marked *Px* are processors, boxes marked *Mx* represent memory modules, and the little dark boxes are *communication assists* (CA) which are controllers or auxiliary processing units that assist in generating outgoing messages or handling incoming messages.

are built from multiple conventional, small, inexpensive, low-power, mass-produced processors. Parallel machines are based on the MIMD architecture (*Multiple Instruction stream, Multiple Data stream*) and are usually classified in two different types: *multicomputers* and *multiprocessors*.

In multicomputer systems, each processor has its own local memory. Therefore, the global memory of the system is physically distributed among all the processors as shown in Figure 2.1(a). Each processor is tightly coupled to its memory, which, besides being physically separate, is logically private from the memories of other processors. A global memory address does not exist; rather, each processor has its own private memory address space. This kind of system is also known as message-passing multicomputer, as it is the only way several processors can communicate among themselves.

A multiprocessor is a parallel system compound of several interconnected processors which share a global physical address space that can be accessed from any processor. This kind of system is also known as shared-memory system. Depending on how the memory is shared, multiprocessors may be classified in different types:

- In UMA (*Uniform Memory Access*) systems, the access to all shared

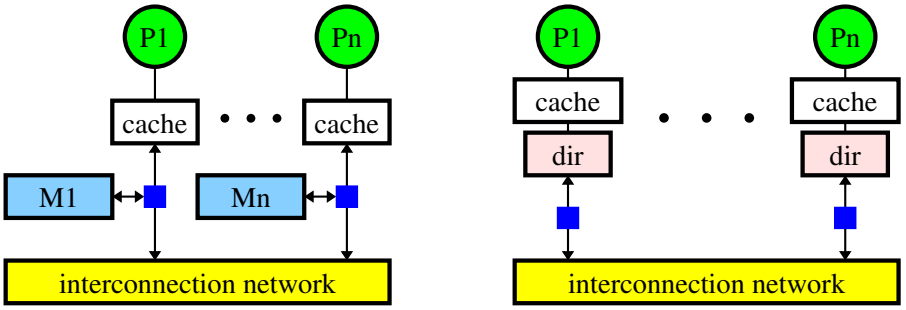


Figure 2.2: Examples of (a) CC-NUMA and (b) COMA architectures. Boxes marked *dir* represent a directory.

data of main memory from any processor is uniform, that is, the access latency does not depend on the location of the physical address. As Figure 2.1(b) depicts, every processor has its own private cache and all the processors and memory modules attach to the same interconnect. These systems are also known as SMP (*Symmetric Multiprocessing*). The UMA systems that incorporate cache coherence are usually named as CC-UMA (*Cache-Coherent Uniform Memory Access*).

- In NUMA (*Non-Uniform Memory Access*) architectures, processors and memory modules are closely integrated such that the access to the local memory is faster than the access to the remote memories. Figure 2.2(a) illustrates the NUMA model, where the global memory is shared but local to each processor. This model is also known as DSM (*Distributed Shared Memory*). The main advantage of the NUMA architecture is that the access to the local memory is faster than that in the UMA model, although the access to a non-local memory is slower. There exists a CC-NUMA (*Cache-Coherent Non-Uniform Memory Access*) model with distributed shared memory and cache directories to implement coherency. Besides, there exists another alternative, NCC-NUMA (*Non Cache-Coherent Non-Uniform Memory Access*) where data are storable in the processor's cache only if those data belong to its local memory, thereby not requiring to maintain coherence.

- In COMA (*Cache Only Memory Access*) architecture, the local main memory is managed as a hardware cache, providing replication and coherence at cache block granularity. In COMA machines, every memory block in the entire main memory has a hardware tag associated with it. There is no fixed node where space is always guaranteed to be allocated for a memory block. Rather, data dynamically move to and are replicated in the main memories that access. These main memories are organized as caches, shown in Figure 2.2(b). Some authors consider this model as a special kind of NUMA machine where the distributed local memories become caches memories. The main advantage of the COMA model is that it frees parallel software from worrying about data distribution in main memory. However, COMA machines require a lot of hardware support, they have extra memory overhead, and the required coherence protocols are complex.

In uniprocessor systems, there exists a single processor that accesses the cache. Therefore, whatever read of a location will return the latest value written to that location. This is the fundamental property of the memory abstraction which sequential programs rely on. This property should be fulfilled even when a shared address space is used. A read should return the latest value written to the location regardless of the processor that wrote it. However, when two processors see the shared memory through different caches (such as in the aforementioned multiprocessor systems), a danger exists since one may see the new value in its cache while the other still sees an old value. This is called the cache coherence problem. To address this problem while sharing a global space, multiprocessors use *cache coherence protocols* to (1) notify the changes over shared data, (2) to avoid the access to stale data, and (3) to facilitate the access to the updated data.

There exist two different options to implement cache coherence protocols. On the one hand, the protocols that invalidate cache copies (other than the writer's copy) on a write are called *invalidation-based protocols*. On the other hand, the protocols that update cache copies are called *update-based protocols*. In both cases, the next time the processor with the copy accesses the block, it will see the most recent value, thereby ensuring a coherent view of the memory system. Since invalidation-based coherence has been used in most recent

systems (e.g., [123, 99, 93, 35, 113, 18, 26]), this dissertation only considers this kind of implementation.

2.3 Coherence Models

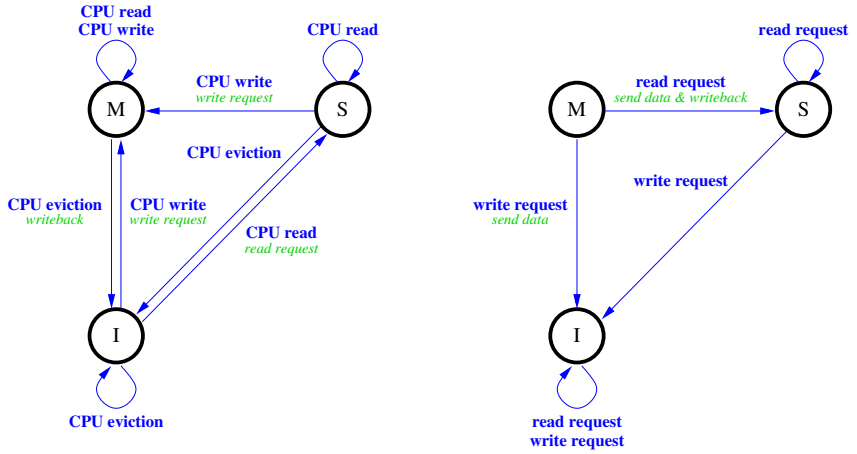
Cache coherence protocols can use different policies for establishing the rights (read or write) and duties (supply the data to other caches) that a specific node has over each block. Following, some of the most used policies are described.

2.3.1 MSI

MSI is a simple invalidation-based protocol for write-back caches. It is very similar to the protocol that was used in the Silicon Graphics 4D series multiprocessor machines [23]. The MSI protocol defines three states, *modified* (M), *shared* (S), and *invalid* (I), to distinguish valid blocks that are unmodified (clean) from those that are modified (dirty). Invalid means the block is not present in cache. Shared means the block is present in cache in an unmodified state, main memory is up-to-date, and zero or more other caches may also have an up-to-date (shared) copy. Modified means that only this cache has a valid copy of the block and the copy in main memory is stale. An invalid block can not be neither read nor written, a shared block can be read, but not written, and a block in the modified state can be read and written.

Before an invalid block can be read or before a shared/invalid block can be written, the processor has to order such an operation (read or write) upon the block. To this end, the MSI protocol defines two different classes of requests: *write requests* and *read requests*.

On a write miss, a write request is used to tell other caches about the impending write and to acquire an *exclusive* copy of the block. A cache is said to have an exclusive copy of a block if it is the only cache with a valid copy of it (main memory may or may not have a valid copy). Therefore, a write request serves to both order the write and cause the invalidation of all copies. The memory system (possibly another cache) supplies the data to the requester. Once the requester acquires the exclusive copy, the write can be performed in its cache.



(a) processor events (b) incoming messages

Figure 2.3: State transition diagram for the MSI model. The arcs are transitions due to either (a) processor events or (b) incoming messages. In italic, it is shown the action performed depending on the state and the event.

On a read miss, that is, when there is not intention to modify the copy of a block, a read request is issued. The memory system (possibly another cache) supplies the data.

Figure 2.3 shows the state transition diagram that governs a block in each cache for the MSI protocol. As shown, a processor read to a block that is invalid causes the issue of a read request to service the miss. The newly loaded block transitions from invalid to shared in the requesting cache, as shown in 2.3(a). Any other caches with the block in the shared state that observe the read request take no special action, allowing main memory to respond with the data. However, if a cache has the block in the modified state and it receives a read request, then it must respond with the data, update the copy in main memory, and its copy of the block transitions to the shared state, as shown in Figure 2.3(b). It is also possible not to update the copy in main memory, leaving memory still out-of-date, but this requires more states [115].

On a write miss (writing into an invalid or shared block), a write request is issued. This request causes all other cached copies of the block to be invalidated, thereby granting the requesting cache exclusive ownership of the block.

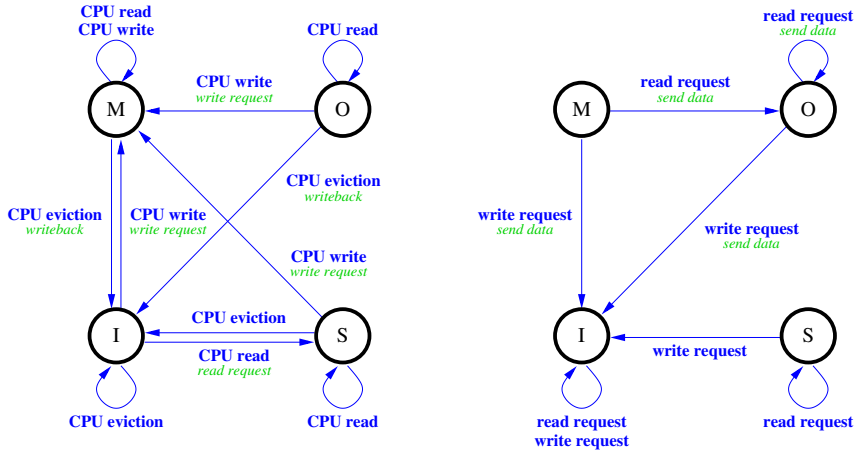
The block in the requesting cache transitions to the modified state, and the desired bytes are then written into it. A common optimization to reduce data traffic is to introduce a new request, called *upgrade request*. An upgrade request obtains exclusive ownership just like a write request, by causing other copies to be invalidated, but it does not cause main memory or any other device to respond with the data for the block. Upgrade requests are useful on write misses for shared blocks.

A replacement of a block from a cache causes its eviction. This replacement causes the state machine for two blocks to change states: the one being replaced changes to invalid, and the one being brought in changes either to shared or to modified. If the block being replaced was in modified state, the block is written back to main memory. However, if the block being replaced was in shared state, a silent eviction is performed (it is not necessary to inform about the eviction).

2.3.2 MOSI

The main advantage of the MSI model is its simplicity, but it has numerous drawbacks. For instance, when a cache block transitions from modified to shared, the block has to be written back to main memory, which may generate a lot of data traffic. Besides, the requests for blocks shared by two or more processors are always served by main memory, which is slow (memory-to-cache transfer). To improve these aspects, some models add a new *owned* state (O). This state in a processor's cache allows read only access to the block (much like shared), but also signifies that the value in main memory is not up-to-date. In addition, a cache is said to be the owner of a block if it must supply the data upon a request for that block [115]. This permits that in some implementations of the MOSI model (such as those based on IBM NorthStar/Pulsar processors [27, 28, 63]) the latency of cache misses lowers since data are usually supplied by caches (cache-to-cache transfer) instead of main memory (memory-to-cache transfer).

Figure 2.4 shows the state transition diagram for the MOSI protocol. As illustrated, if a cache holds a block in modified state and it receives a read request for it, it must provide the data to the requester and its copy transitions to owned. Note that, unlike the MSI model, the block is not written back to



(a) processor events (b) incoming messages

Figure 2.4: State transition diagram for the MOSI model.

main memory, leaving memory still out-of-date, thereby lowering the data traffic.

For a memory block, only one cache can have a copy of it in owned state, while the other copies of the block can be in shared state. The cache holding a block in owned state is in charge of supplying the data to all the caches that request a copy. Note that, if a read request has been observed, the owned cache remains in the same state, but if a write request has been observed, the block transitions from owned state to invalid state.

Like in modified state, the replacement of a block in owned state causes the block to be written back to main memory.

2.3.3 MESI

Another aspect of the MSI model susceptible to be improved is the following: a cache with a block in modified state does not distinguish between an exclusive copy that has been modified and an unmodified exclusive copy that is only held by that cache (since any other cache does not currently have a valid copy). This situation can lead to unnecessary data traffic, as the replacement of unmodified exclusive blocks cause the blocks to be written back to main memory. Besides this problem, another concern arises when the MSI model is

used in a multiprocessor running a sequential application. In this case, when a processor reads in and modifies a memory block, the MSI model generates two consecutive cache misses (even though there are no sharers), since the first cache miss retrieves the block in shared state and the second it is necessary to convert S state to M state.

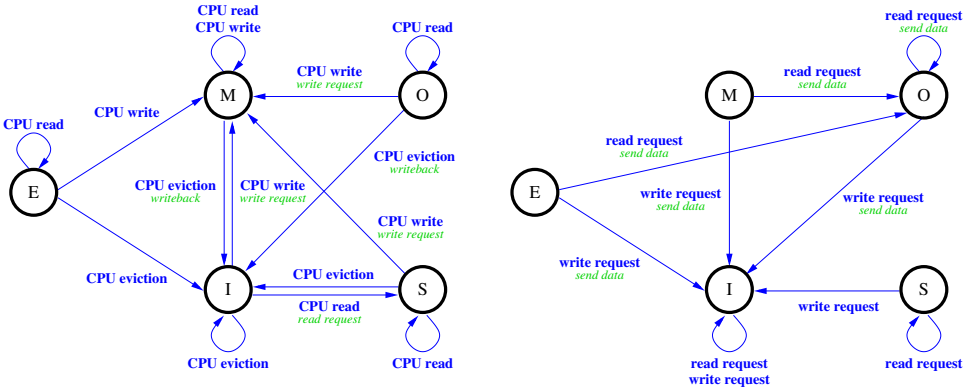
The two aforementioned situations are avoided by adding a state indicating that the block is the only (exclusive) copy but it is not modified. This new state, called *exclusive* (E), indicates an intermediate level of binding between shared and modified. It is exclusive, so unlike the shared state, the cache can perform a write (directly transitioning to the modified state). However, the exclusive state does not imply ownership (memory has a valid copy), so unlike the modified state, the cache does not need to reply when observing a request upon the block. Variants of this MESI protocol [95] are used in may microprocessors, including the Intel Pentium, PowerPC 601, and the MIPS R4400 used in the Silicon Graphics Challenge multiprocessors.

2.3.4 MOESI

To join the major advantages of MOSI and MESI models, the MOESI protocol was proposed. Figure 2.5 shows the state transition diagram for this model. The final definition of the states is as follows. A cache has a block in *modified* state when it is the only valid copy of the block in the system. This copy has been modified and the copy in main memory is stale. A cache with the block in modified state can read and write that block. On a replacement, the block has to be written back to main memory. The modified state implies ownership. Therefore the data must be supplied to both read requesters (transitioning to owned) and write requesters (transitioning to invalid).

A cache has a block is in *owned* state when that cache and, at least, another one have a valid copy of the block. The copy in main memory may be stale, therefore, on a replacement, the block is written back to main memory. A cache with the block in owned state can only read it. Like the modified state, this state implies ownership. Therefore, it must supply the data when observing a read request (remaining in owned state) or a write request (transitioning to invalid state).

A cache has a block in *exclusive* state if it is the only cache with a valid



(a) processor events (b) incoming messages

Figure 2.5: State transition diagram for the MOESI protocol.

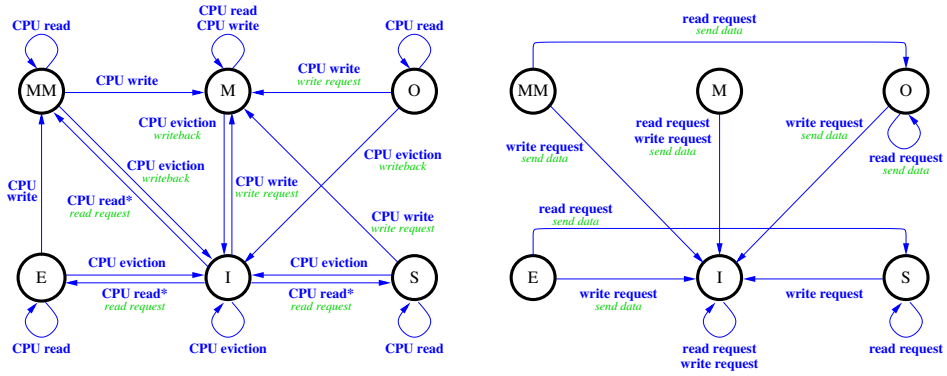
copy and its value matches with the value in main memory. On a replacement, the block does not have to be written back to main memory. A cache with a block in exclusive state can write it (making a silent transition to modified state) and read it (remaining in the exclusive state). Unlike the exclusive state defined in the MESI model, the exclusive state in the MOESI model implies ownership. Therefore, a cache with a block in that state has to serve both read requests (transitioning to owned state) and write requests (transitioning to invalid state) upon the block.

A block is in *shared* state when there exist several valid copies of the block throughout the system. A cache with a block in shared state can only read it. This state does not imply ownership. Therefore, it is not in charge of serving requests.

A block is in *invalid* state when the cache does not have a valid copy of it. The issue of a request will be required to be able to access the block.

2.3.5 Optimization for Migratory Sharing

Although the MOESI model is, in general, efficient, some researchers proposed a modification to optimize it for migratory sharing patterns [37, 110]. Migratory sharing patterns are common in many multiprocessor workloads and they result from data blocks that are read and written by many processors over time, but by only one processor at a time [119]. This pattern in



(a) processor events (b) incoming messages

Figure 2.6: State transition diagram for the MOESI protocol with the migratory sharing optimization.

systems implementing the MOESI model would result into read-then-write sequences, generating a read miss followed by a write miss. Thus, to target this read-then-write pattern, we use in this dissertation an additional *modified migratory* (MM) state, similarly to that proposed in [76]. This state signifies the block has been previously modified by another processor (not by this) and that the copy in main memory is stale. A cache in modified migratory state silently transitions to M when it writes the block. Hence, if a workload contained only these read-then-write patterns, the policy of always responding with migratory data would perform well; however, this policy substantially penalizes other sharing patterns (e.g., widely shared data). To find a balance, we employ the heuristic of only sending migratory data when the responding processor is in the modified state. In contrast, a cache block in the modified migratory state behaves as a standard MOESI protocol.

Figure 2.6 shows the state transition diagram for the MOESI model with the migratory sharing optimization. As shown, if a processor modifies a copy in modified migratory state, it will transition silently to modified state. The main difference with the standard MOESI model is that the transition from the modified state when a read request is received is to the invalid state. Due to the majority of applications has a migratory sharing pattern, this optimization has been assumed by many protocols, such as those in [76, 84, 65].

While some old processors used a subset of the MOESI states (for example, IBM's PowerPC 755 supports the MEI (Modified, Exclusive, Invalid) protocol and the Intel IA-32 processor family supports the MESI protocol [1]), modern processors use several variants of MOESI (such as Sun Ultrasparc's MOESI protocol and the AMD 64's MOESI protocol).

2.4 Consistency Models

To write correct shared-memory programs, programmers need a precise notion of how memory behaves with respect to read and write operations from multiple processors. This formal specification of how the memory system will behave is provided by a *memory consistency model* [87]. Consistency models place restrictions on the values that can be returned by a read or a write operation in a shared-memory program. According to these restrictions, the most relevant models are briefly described following.

Atomic Consistency. This is the strictest of all consistency models since any read to a memory location X must return the value stored by the most recent write operation to X.

Sequential Consistency. Sequential consistency is the most commonly assumed memory consistency model. It is slightly weaker than atomic consistency. It was first formally defined by Lamport as follows [66]: *"A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"*. The implication of this definition is all processors must agree on the order of the operations.

Processor Consistency. In this model writes done by a single processor are received by all other processors in the order in which they were issued, but writes from different processors may be seen in a different order as long as those writes are to different locations.

Weak Consistency. This model classifies memory operations into two categories: data operations and synchronization operations. Accesses to synchronization variables are sequentially consistent. Accesses to a synchronization variable are not allowed to be performed until all the previous writes have com-

pleted everywhere. Furthermore, data accesses are not allowed to be performed until all previous accesses to synchronization variables have been performed.

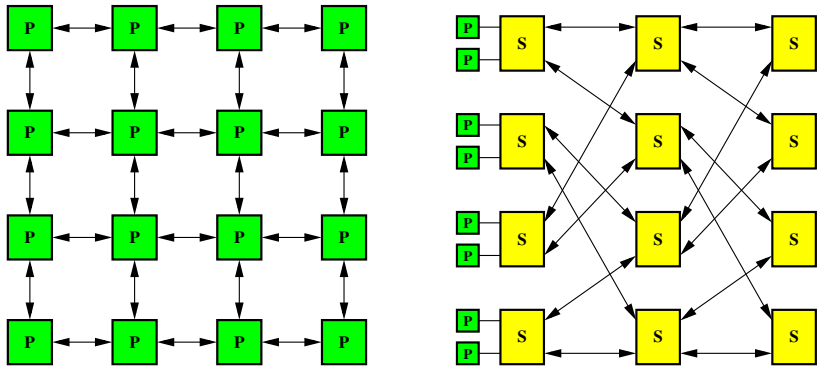
Release Consistency. Instead synchronization operations, release consistency models consider locks on areas of memory. Locks are managed by two operations: *acquire* and *release*. Before an access to a shared variable is performed, all previous acquires must have completed successfully. Before a release is allowed to be performed, all previous reads and writes must have completed. The acquire and release accesses must be sequentially consistent.

2.5 Interconnection Networks

The goal of a multiprocessor interconnect is to provide low-cost, low-latency, high-bandwidth, and reliable message delivery between processors and memory modules. Although many factors influence on the achievement of such goals, they mainly depend on the network topology, the routing technique, and the switching technique. This section briefly highlights the design considerations that are relevant to our proposals and describes two concrete interconnects that we use in our later evaluations. A more complete and detailed discussion of interconnects can be found in a book dedicated to this subject by Duato et al. [41].

2.5.1 Network Topology

The topology defines how system components (processors and memory modules) are interconnected. Topologies can be classified in several types. In this dissertation we consider just two different types: direct and indirect networks. *Direct* (also known as point-to-point) networks consist of a set of system components, each one being *directly* connected to a (usually small) subset of other components in the network. Each component usually has a switch (or router), which handles message communication among components. Each switch has direct connections to the switch of its neighbors. Usually, two neighboring components are connected by a pair of unidirectional channels in opposite directions. Although the function of a switch can be performed by the local processor, dedicated switches have been used in high-performance multicomputers, allowing overlapped computation and communication within



(a) Mesh interconnect (b) MIN interconnect

Figure 2.7: Interconnection network topologies. In (a) boxes marked as "P" represent highly-integrated nodes that include a processor, caches, memory controller, coherence controllers, and a switch. In (b) boxes marked as "P" are similar to those in (a), but they do not include the switch, which is represented by boxes marked as "S".

each node. As the number of nodes in the system increases, the total communication bandwidth, memory bandwidth, and processing capability of the system also increase. Thus, direct networks have been a popular interconnection architecture for building large-scale parallel computers. Figure 2.7(a) illustrates a mesh interconnect which is an example of direct network. One interesting property of this topology is the simplicity of its routing.

In *indirect* (or switch-based) networks, the communication among system components has to be performed through some switches. Each component has a network adapter that connects to a network switch, which provides a set of ports. Each port consists of one input and one output link. A set of ports in each switch may be connected to processors, whereas the remaining ports are connected to ports of other switches to provide connectivity among the components. The interconnection pattern of those switches (switch fabric) defines the network topology. A wide range of topologies has been proposed, ranging from regular topologies used in multicomputers and shared-memory multiprocessors to irregular topologies, commonly used in NOWs and PC clusters. Single-switch crossbars and some hierarchical indirect interconnects use a centralized root switch. Such interconnects are desirable because they provide

ordering properties required for snooping protocols (which are described in the next chapter). Figure 2.7(b) illustrates an example of indirect interconnect. This interconnect is a perfect shuffle bidirectional multistage interconnection network (MIN). It uses discrete switches, which are located in multiple stages. Components are only connected to the first stage of switches. One of the key advantages of MINs is that they allow to easily implement collective communications such as broadcast or multicast messages [36, 97].

2.5.2 Switching Techniques

The switching techniques establish when network switches are set to connect each pair of nodes and they also determine how the information between components is transferred. Several techniques have been proposed until now, being the most known ones: circuit switching, packet switching, virtual cut-through (VCT), and wormhole.

In circuit switching, before a source can transmit information to a destination, a physical path must be reserved. Once the path is reserved, all the information transmitted by the source is delivered to the destination by following the reserved path. Only the source node will be able to use the reserved path until it releases it. This technique is advantageous when messages are infrequent and long, since the time required to set the path is considerable.

In packet switching (also known as store-and-forward), messages are partitioned and transmitted as packets. Each packet has a header which contains routing and control information. Each packet is individually routed from source to destination and is completely buffered at each intermediate switch before being forwarded to the next switch. This technique is advantageous when messages are short and frequent since many packets belonging to a message can be transmitted simultaneously.

In virtual cut-through switching messages can be routed as soon as their header is received. Thus, unlike packet switching, it is not necessary to wait for the complete message to route it. Hence, the message advance is pipelined through the network. However, if the message header blocks at a switch, the complete message will be stored at the switch buffers. The main advantage of this technique is that, in the absence of blocking, the latency of each packet reduces considerably.

Wormhole switching is similar to virtual cut-through switching. However, unlike virtual cut-through, in wormhole messages remain in the network, maintaining the possession of the network resources used when the message header blocks. In this case, messages are broken up into flits (the unit in flow control). Buffers at switches can typically store a few flits, thereby reducing the buffer requirements at each switch. As messages usually are too large to be completely buffered within a switch, they can occupy buffers along several switches.

The cited switching techniques usually assume that switch buffers are implemented as FIFO queues. Therefore, when the packet placed at the head of the buffer is blocked, no other packets can access the physical channel. However, several virtual channels multiplexed across the physical channel can be used to decouple buffer allocation from physical channel bandwidth. Each virtual channel is implemented with an independently managed buffer. This decoupling prevents a blocked packet from holding channel bandwidth idle. A virtual network is a subset of channels that are used to route packets toward a particular set of destinations. The channel sets corresponding to different virtual networks are disjoint. Depending on the destination, each packet is injected into a particular virtual network, where it is routed until it arrives at its destination.

2.5.3 Routing Techniques

In order to efficiently route packets through a network, a routing algorithm must be used. Routing algorithms can be *deterministic* or *adaptive*. In deterministic routing the path followed between a given source-destination pair is always the same. This is achieved by the switches providing only one routing option (output port) for a packet. With adaptive routing several routing options may be provided by a switch to forward a packet. The selection of the routing option is usually made based on the current status of the links. Thus, with adaptive routing, packets can avoid congested areas in the network.

According to the number of destinations, routing algorithms can be classified into unicast routing, multicast routing, and broadcast routing algorithms. The implementation of multicast and broadcast algorithms can be done by splitting the multicast/broadcast message into several unicast messages, as

many as number of destinations. However, this may be too inefficient. Alternatively, the interconnect can initiate a single message that fans out across the interconnect.

A key issue in the design of routing algorithms is how to prevent deadlock. Deadlock occurs when no packet can advance toward its destination because the requested network resources (buffers/channels) are not released by the packets possessing them. In a deadlock situation, waiting packets are involved into a cycle of resource dependencies[1]. Deadlocks can be avoided by eliminating the cycles in the resource dependency graph. This can be achieved by imposing some restrictions on routing, for example, imposing an ordering in the allocation of the resources as in Dimension-Ordered Routing (DOR) and Up*/Down* routing. DOR has been proposed for meshes and tori networks. This routing algorithm forwards every packet through one dimension at a time, following an established order of dimensions. So, for example, in a 2D mesh the algorithm is also known as XY routing because packets must traverse first all the required channels in X dimension before traversing, if required, the channels in Y dimension. On the other hand, Up*/Down* routing is based on an assingment of direction labels ("up" or "down") to the links in the network graph. Cyclic channel dependencies are avoided because a packet is not allowed to traverse a link in the "up" direction after having traversed one in the "down" direction. Some routing schemes may require the use of virtual channels to remove cycles in the resource dependence graph. This would be the case of the DOR algorithm when applied to tori.

[1]There exists a dependence from i to j if a packet possessing the resource i requests the resource j.

Chapter 3

Cache Coherence Mechanisms

This chapter intends to give insight on the evolution of the cache coherence protocols through the last years. Traditionally two different approaches to coherence have been used: snooping-based protocols (Section 3.1) and directory-based protocols (Section 3.2). These approaches have antithetical benefits. Thus, while snooping-based protocols obtain low-latency cache-to-cache misses thanks to a direct communication among processors, directory-based protocols can use scalable interconnection networks. Alternatively to these traditional approaches, other approaches have been proposed and they are briefly described in Section 3.3. Section 3.4 describes in more detail a recently approach to coherence which is based on tokens. We focus on this approach because it integrates the main advantages of traditional protocols and it is the approach on which this dissertation is based. We conclude this chapter with a brief summary (Section 3.5).

3.1 Snooping-based Protocols

A simple and elegant solution to cache coherence arises from the very nature of buses. A bus is a single set of wires connecting several devices, each of which can observe every message on the bus. When a processor issues a block to its cache, the cache controller examines the state of that block and takes suitable action, which may include sending a request to access memory. Coherence is maintained by allowing all cache controllers to "snoop" (hence the name

snooping protocol) on the bus and to monitor the requests [50]. A snooping cache controller may take action if a message on the bus involves a memory block of which it has a copy in its cache.

Snooping protocols exploit the nature of buses by directly implementing the MOESI model (or a derived one). In these systems, cache controllers begin the coherence process by arbitrating for the interconnect. Once granted access to the interconnect, the cache controller can send a request. Each of the other cache controllers snoops the interconnect and, if it holds a valid copy of the requested block, its state changes according to the transitions defined by the coherence model, supplying the data when necessary. Simultaneously, the memory can determine if it should respond by either storing the state for each block in the memory [46, 16] or by observing the responses generated by the other cache controllers. Only after the requesting cache completes its request another cache is allowed to initiate a new one.

The main advantage of snooping protocols is the low average miss latency, especially for cache-to-cache misses. Since requests are sent directly to all processors and memory modules in the system, the responders quickly know they should respond. However, the main disadvantage of snooping protocols is that the cache coherence overhead and the speed of shared buses limit the bandwidth needed to broadcast messages to all processors. To improve this aspect, many solutions have been proposed. For example, for more efficient interconnect usage, the requesting processor may be allowed to release the bus while waiting for a response [47]. Systems can also use multiple address-interleaved buses [32] and separate address and data subnetworks to multiply available bandwidth [32]. Even more aggressive systems avoid the electrical limitations of shared-wire buses entirely and implement a virtual bus using point-to-point links, dedicated switch chips, and distributed arbitration [32] or, even, use optical interconnection networks [72]. Asynchronous caches [98] are implemented by using a deeply pipelined memory system with parallel-link interconnection and queues through which the memory and the processors communicate. Timestamps snooping protocols [79] reduce the execution time by broadcasting requests over indirect interconnection networks. Although each of these enhancements add significant complexity, many snooping systems use them to create high-bandwidth systems with dozens of processors (e.g.,

Sun's UltraEnterprise servers [31, 33, 107]).

Some works point out that rings may be an attractive interconnect for current multiprocessors because they can provide faster links than buses. Thus, in [111] a family of adaptive forwarding and filtering snooping algorithms for rings are proposed. However, other works claim that the ordered interconnects such as buses or rings are not suitable and implement snooping on interconnects that do not provide any ordering capability. To alleviate the damage caused by the broadcast of requests in those interconnects, in [10] it is proposed that the network itself can order the requests in a distributed manner, creating a global order among them. Alternatively, in [44] it is proposed a coherence protocol that relies on a virtually ordered network and, instead of using broadcast messages, the use of multicast messages is proposed. Assuming this last approach, in [56] a multicast router design to improve network performance while reducing network activity is proposed.

Another problem of snooping protocols is that they are inefficient from the point of view of power dissipation, as caches need to monitor ("snoop") the bus, send different signals, and to enforce data consistencies. This mean that caches must be in active (non-sleep) mode, so that the requests can be properly received and acted upon, which entails a significant power consumption. Recently, several authors have addressed this problem by, for example, invalidating the copies of some caches when they are sleeping [12]. Other works try to reduce the power consumption of broadcast messages by predictions [21].

3.2 Directory-based Protocols

Directory protocols address the scalability and interconnection constrains of snooping protocols at the expense of increasing the latency of cache-to-cache misses. There are many systems that use protocols based on directory such as the Stanford's DASH [69, 70] and FLASH [64], MIT's Alewife [8], SGI's Origin [67], the AlphaServer GS320 [49] and GS1280 [39], Sequent's NUMA-Q [73], Cray's X1 [4], and Piranha [22].

To avoid the broadcast of requests, when a cache miss occurs, the request is only sent to the home memory module of the requested block. Each memory module contains a *directory* which encodes the state of the block and

a superset of the processors that have a copy of it. Thus, when the memory module receives the request, it can use the information stored in its directory to respond with the requested block and/or forward the request to other processors.

In addition to tracking sharers, the directory also provides a per-block ordering point to handle conflicting requests or eliminate various protocol races. A protocol race can occur when multiple requests contend for the same block at the same time. Since the directory observes all requests for a given block, the order in which requests are processed by the directory unambiguously determines the order in which these requests will occur in the system. Many directory protocols delay subsequent requests for the same block by queuing or negatively acknowledging (nacking) them while a previous request for the same block is still being served. Only when the first request has completed subsequent requests are allowed to proceed past the directory. The performance of a directory protocol will depend on how often the requests are delayed.

One important advantage of directory protocols is that they scale much better than snooping protocols. This is perhaps its most discussed and studied advantage. By only contacting those processors that might have copies of a cache block (or a small number of additional processors when using an approximate directory implementation), the traffic in the system grows linearly with the number of processors. In contrast, the traffic generated by broadcast messages grows quadratically. Combined with a scalable interconnect (one whose bandwidth grows linearly to the number of processors), a directory protocol allows a system to reach to hundreds or thousands of processors. However, at large system sizes, two scalability bottlenecks arise. First, the amount of directory state required becomes a major consideration. Second, if the interconnect is not well designed, it will not scale. Both of these problems have been studied extensively, and actual systems that support hundreds of processors exist (e.g., the SGI Origin 2000 [67]).

The second and perhaps more important advantage of directory protocols is the ability to exploit arbitrary point-to-point interconnects. In contrast, snooping protocols are restricted to systems with virtual bus interconnects. Arbitrary point-to-point interconnects exhibit often high-bandwidth,

low-latency, and are able to easily exploit the levels of integration by including the switch logic on the main processor chip.

Directory protocols have two primary disadvantages. First, the extra interconnect traversal and directory access is on the critical path of cache-to-cache misses. On the contrary, memory-to-cache misses do not incur a penalty because the memory lookup is normally performed in parallel with the directory lookup. In many systems, the directory lookup latency is similar to that of main memory DRAM, and placing this lookup on the critical path of cache-to-cache misses significantly increases cache-to-cache miss latency. While the directory latency can be reduced by using fast SRAM to cache directory information, the extra latency due to the additional interconnect traversal is more difficult to mitigate. These two latencies often combine to dramatically increase cache-to-cache miss latency. With the prevalence of cache-to-cache misses in many important commercial workloads, its higher-latency can significantly impact system performance. These problems have been studied by many authors. Thus, for example, some works propose the use of predictors to reduce the latency of cache-to-cache misses [5] or to accelerate the upgrade misses [6].

The second disadvantage of directory protocols involves the storage and manipulation of directory state. This disadvantage was more pronounced on earlier systems that used dedicated directory storage (SRAM or DRAM) which added to the overall system cost. However, several recent directory protocols have used the main system DRAM and reinterpretation of bits used for error correction codes (ECC) to store directory state without additional storage capacity overhead (e.g., the S3mp [92], Alpha 21364 [89], UltraSparc III [54], and Piranha [22, 48]). Storing these bits in main memory does, however, increase the memory traffic by increasing the number of memory reads and writes [48]. In [7] a scalable directory architecture is presented to reduce the size of the directory in medium/large systems without degrading performance.

To reduce the average memory latency and improve the performance scalability of directory protocols, an in-network cache coherence protocol [42] has been proposed, which moves the directories into the network.

3.3 Non-traditional Protocols

Recently several protocols have appeared, but they are not easily classified as either snooping or directory protocols (e.g. IBM's Power5 [108] and xSeries Summit [29] systems, and AMD's Hammer [58]). These systems have a small or moderate number of processors, use a tightly-coupled point-to-point interconnect, and broadcast all requests. Unfortunately, these systems are currently not well described in the academic literature.

In the Sun Fire 3800-6800 systems [34], both snoopy broadcast coherency and point-to-point directory coherency are built. The snoopy part of the protocol provides low memory latency for small and medium-sized systems. The point-to-point part of the protocol allows very-large systems to be implemented.

In [80] a hybrid protocol is proposed. This protocols behaves like snooping protocols when there is enough bandwidth and it behaves like directory protocols when there is a lack of it.

In the Intel E8870 and E9870 chipsets the Scalability Port (SP) protocol [19] is used. The SP protocol is an invalidation based protocol that uses an MESI policy. It supports either a convectional directory based design or a snoop-filter based design. The snoop filter is a centralized component that tracks valid blocks or lines in caches. To maintain the track, it sends snoop probes to caching nodes and collects the snoop responses. Thus, it can keep the information about the state and presence of memory blocks at the caching nodes. On a cache miss, a read/write request is sent to the snoop filter. This component filters the request to remote nodes that do not contain a copy of the block, but the ones that hold a valid copy will receive the request. Hence, when the processor holding the requested memory block receives the request, it will provide a valid copy through a different virtual network.

3.4 Token-based Protocols

Token-based protocols are proposed to simultaneously capture the best aspects of snooping protocols and directory protocols: low-latency cache-to-cache misses and not reliance on totally-ordered interconnects. Token Co-

herence [77] is a framework proposed to easily develop token-based cache coherence protocols. Token Coherence is composed of three components. The *token counting mechanism* ensures data are read and written in a coherent fashion. The *persistent request mechanism* solves protocol races and prevents starvation. These two mechanisms form the correctness substrate which provides correct operation in all cases. However, to make the protocol fast and bandwidth-efficient, a third component is used: the *performance policy*.

3.4.1 Token Counting

The system associates a fixed number of tokens with each block of shared memory. A processor is only allowed to read a cache block when it holds at least one of its tokens or write a cache block when holding all the block's tokens. This token-counting approach directly enforces the single-writer or many-reader coherence invariant, thereby ensuring correct operation in all cases. Note that, one of the primary benefits of token counting is that it allows to ensure safety without relying on request ordering.

During system initialization, the system assigns each block a fixed number of tokens, T. The number of tokens for each block (T) is generally at least as large as the number of processors. Tokens are tracked per block and can be held in processor caches, memory modules, coherence messages, and input/output devices. A *coherence message* is any message sent as part of the coherence protocol. Initially, the block's home memory module holds all the block's tokens. Tokens (and data) are allowed to move between system components as long as the substrate maintains these four rules:

- **Rule 1 - Conservation of tokens:** At all times, each block has T tokens in the system, one of which is the *owner token*.
- **Rule 2 - Write rule:** A processor can write a block only if it holds all T tokens for that block.
- **Rule 3 - Read rule:** A processor can read a block only if it holds at least one token for that block and has valid data.
- **Rule 4 - Data transfer rule:** If a coherence message contains the owner token, it must contain data.

Rule 1 ensures that tokens are never created or destroyed. Rules 2 and 3 ensure that a processor will not write a block while another processor is reading it. Rule 4 ensures that the processor holding the owner token always has a valid copy of the memory block. Besides, this rule allows coherence messages with non-owner tokens to omit data, but it still requires that messages with the owner token contain data (to prevent all processors from simultaneously discarding data). These rules provide the following guarantee: each memory block can have either a single writer or multiple readers (but not both at the same time). Although this guarantee concerns only a single block and consistency involves the ordering of read and writes to many blocks, it is sufficient to enforce sequential consistency or any weaker consistency model.

Token possession maps directly to MOSI states: holding all T tokens is *modified* (M); at least the owner token, *owner* (O); at most T - *1* tokens (but not the owner token), *shared* (S); and no tokens, *invalid* (I). Processors and memory modules maintain a valid bit, to hold non-owner tokens without valid data. The valid bit is set when a message with data and at least one token arrives and the valid bit is cleared when tokens are not longer held.

Tokens are held in processor caches, memory (e.g., encoded in ECC bits [48]), and coherence messages. Since the protocol does not track which processors hold tokens but only count them, tokens can be stored in $2 + log_2 T$ bits (valid bit, owner-token bit, and non-owner token count). For example, encoding 64 tokens with 64-byte blocks adds one byte of storage (1.6% overhead).

3.4.2 Persistent Requests

It must be ensured that all attempts to read or write a block will eventually succeed, thereby preventing starvation. The flexibility provided by Token Coherence is one of its key advantages, but this same flexibility is directly responsible for complicating starvation prevention. To prevent starvation, the correctness substrate uses a starvation prevention mechanism called *Persistent Requests*. This mechanism must guarantee completion in all cases not explicitly disallowed by the token counting rules. For example, tokens can be delayed arbitrarily in transit, tokens can ping-pong back and forth between processors, or many processors may wish to access the same block at the same

time.

According to the persistent request mechanism, a processor issues a persistent request when it detects it may be starving. The substrate arbitrates among the outstanding persistent requests to determine the current active request for each block. The substrate sends the active persistent requests to all system components. These components must both remember all active persistent requests and redirect their tokens (those tokens currently present and those to be received in the future) to the requesting processor until the requester explicitly deactivates its own persistent request. The initiator deactivates its requests when it has received sufficient tokens to perform the intended memory operation.

Several design options of the persistent request mechanism have been proposed. The simplest implementation uses a *single centralized arbiter*. According to it, when a starvation situation is detected, a persistent request is directed to the arbiter. The arbiter stores all the received persistent requests and activates a single one by informing all processors and the block's home memory module. These components each remember the active persistent request using a hardware *persistent request table*. In this approach, each persistent request table has a single entry with four fields: a valid bit, a physical address, a read/write bit (to distinguish between read and write requests), and a processor number. While a persistent request is active for a block, each component must forward all the block's tokens to its requester. The components will also forward tokens that arrive later, because the request persists until the requester explicitly deactivates it. Once the requester has (1) received enough tokens to perform the memory operation, (2) received valid data, and (3) observed the activation of its own persistent request, it sends a message to the arbiter. The arbiter deactivates the persistent request by informing all components, which delete the entry from their tables.

Although the single centralized arbiter is a correct implementation, the arbiter may become a bottleneck in medium/large systems. Therefore, to address this problem, a *banked-arbitration* approach was also proposed. In this case, there may be several arbiters. Each arbiter is responsible for a fixed portion of the global address space. The components each must have a persistent request table that contains one entry per arbiter, which limits the

number of active persistent requests per arbiter. Each table entry encodes the same information as the previous single centralized arbiter.

Both single centralized arbitration and banked-arbitration approaches require that processors first send persistent requests to an arbiter. Like in directory protocols, the arbiters introduce indirection and increase the latency of persistent requests. To avoid indirection, the *distributed-arbitration* approach was proposed. Unlike previous proposals which limit the number of active requests per arbiter, this approach relies on limiting the number of outstanding requests per processor. In this case, the table at each node has as many entries as the maximum number of simultaneous outstanding requests in the system. When a processor detects possible starvation, it sends a persistent request directly to all processors and the home memory module. Each component records the received requests in its local persistent request table. This table needs as many entries as the maximum number of processors in the system (assuming one outstanding persistent request per processor) and each entry contains an address, a read/write request type bit, a valid bit, a marked bit, and a processor number[1]. Since the table can have multiple entries that contain the same address, the entry for the processor with the lowest number is appointed the active persistent request. Similar to previous approaches, all components send tokens (and data) to the initiator of the active persistent request. When a processor completes its request, it sends a deactivation message directly to all processors, which clears the corresponding entry in the table. This deactivation implicitly activates the next persistent request (the request with the next lowest processor number).

To prevent higher-priority processors from starving out lower-priority processors, this approach uses a simple mechanism (inspired by techniques used to enhance multiprocessors bus arbitration [118]) that prevents a higher-priority processor from issuing a new persistent request for the block until all the lower priority processors have completed their persistent requests. When a processor completes a persistent request, it sets a marked bit. A processor is not allowed to invoke a persistent request for any block with a marked entry in its table. The processor will eventually receive the deactivation messages for all the marked entries, allowing the processor to issue a persistent request for

[1] If the table is indexed by processor number, the processor number can be omitted.

that block.

The main advantages of the distributed-arbitration scheme are (1) the latency lowers with respect to the arbiters implementations and (2) arbiters are not required. On the other hand, the banked-arbitration option increases the complexity, size, and access latency of the table and the fixed-priority scheme may create load imbalance.

Both the arbiter-based and the distributed-arbitration mechanisms assume that activation and deactivation messages are never reordered (the activation message always has to be received before its corresponding deactivation one). To get it, several options can be implemented: point-to-point ordering (deterministic routing instead of adaptive), explicit acknowledgments, or large sequence numbers.

3.4.3 Performance Policy

Both the token counting mechanism and the starvation prevention mechanism ensure correct operation in all cases. In order to make Token Coherence fast the performance policy component is defined. The performance policy is the set of specific policies the system uses to instruct the correctness substrate to move tokens and data throughout the system. When it is not overridden by a persistent request, the performance policy decides when and to which processors the system should send coherence messages. Since the correctness substrate guarantees safety and prevents starvation, Token Coherence allows for many possible performance policies.

Initially three performance policies with different attributes were proposed: TokenB (Token-Broadcast), TokenD (Token-Directory), and TokenM (a hybrid that uses predictive multicasting to create a low-latency and bandwidth-efficient protocol). TokenB uses three policies:

- On a cache miss, a transient request is broadcast (i.e., sent to all processors and the block's home memory module).
- Processors and the home memory module respond to transient requests as they would do in most MOESI protocols. A component with no tokens (I state) ignores all requests. A component holding only non-owner tokens (S state) ignores transient read requests, but responds to

transient write requests by sending all its tokens in a data-less message. A component with the owner token but not all other tokens (O state) sends the data with one token (usually not the owner token) on a read request, and it sends the data and all its tokens on a write request. A component with all the tokens (M or E state) responds in the same way as a component in O state.

- If a transient request has not completed after a timeout interval, the processor invokes the persistent request mechanism. To adjust to different interconnect topologies and congestion, the transient request timeout interval is set to twice the processor's average miss latency. Using twice the average miss latency prevents a slightly delayed response from causing a persistent request, but it also invokes the persistent request mechanism quickly enough to avoid to damage performance.

The main advantage of TokenB is the low-latency thanks to the direct communication among processors. However, TokenB is only suitable for moderate-sized systems where bandwidth is plentiful due to the bandwidth requirements of broadcast messages. To improve this aspect, TokenD performance policy was proposed, which is an efficient directory-like protocol. According to the TokenD performance policy, processors send transient requests to a directory at the home memory module. This directory forwards the transient requests to one or more processors that are likely to hold the requested tokens. Processors respond to these forwarded transient requests by using the same policy as that described for TokenB. Like in TokenB, transient requests may also fail to complete (e.g., when forwarded transient requests are reordered or a request is forwarded to an insufficient set of processors). To handle the occasional failure of transient requests, if after twice the processor's average miss latency the transient request does not complete, the persistent request mechanism is invoked.

Although TokenD is bandwidth-efficient, the average latency of request completion increases dramatically due to the indirection. Thus, to obtain a protocol that provides both low-latency and bandwidth-efficiency, TokenM was proposed. TokenM is a hybrid protocol that uses predictive multicast to capture most of the latency benefits of TokenB, while capturing some of

the bandwidth efficiency of TokenD. In TokenM, when a processor issues a request, it sends the transient request to a predicted destination set. Components react to incoming transient requests in the same manner as TokenB and TokenD, and when the recipients respond with sufficient tokens, the requester completes its request without indirection and without broadcast. When the predicted destination set is insufficient, a soft-state directory forwards the request to all processors that it believes the requester left out of the destination set. Token Coherence uses a destination-set predictor to improve the bandwidth/latency characteristics of the system. In the limit of perfect prediction, TokenM provides a near-ideal bandwidth/latency design point, capturing both the low-latency of TokenB and the bandwidth efficiency of TokenD. However, when the prediction is not accurate, the latency of requests may considerably increase. Like in previous proposals, when transient requests fail to complete, the use of the persistent request mechanism is required.

There exist another implementation of Token Coherence [83] that assumes a ring interconnect to simplify the protocol. Although they are correct, we do not consider it because its applicability is limited just to systems with that kind of interconnect.

3.5 Summary

In this chapter we briefly discussed many aspects of the general approaches to coherence, showing their main advantages and disadvantages. Next chapters describe the evaluation methodology and the proposals we make to solve the worst aspects of the token-based protocols by (1) improving the performance (Chapter 5 and Chapter 8), (2) using scalable structures (Chapter 6), and (3) providing scalability to broadcast and non-silent invalidation messages (Chapter 7).

Chapter 4

Evaluation Methodology

This chapter presents the simulation tools (Section 4.1) used for evaluating the goodness and the relative behavior of our proposals. Since our goal is not to evaluate our proposals on all possible system configurations, we have performed a relative accurate comparison between the different approaches by using a simulation model of a current multiprocessor system, which is described in Section 4.2. Finally, a description of the performance metrics used can be found in Section 4.3.

4.1 Simulation Tools

We use full-system simulation to evaluate our proposals. Full-system simulation lets us to evaluate the proposed systems running realistic scientific applications on top of actual operating systems. It also captures the subtle timing effects not possible with trace-based evaluation.

To perform the analysis, we use the Simics full-system multiprocessor simulator [74] extended with the Wisconsin GEMS simulation environment [81], which in turn is extended with a multiprocessor interconnection network simulator developed by the Parallel Architecture Group (GAP) [51]. Simics is a system-level architectural simulator developed by Virtutech AB [3] that can run unmodified commercial applications and operating systems. Simics only provides an interface to support the memory hierarchy model provided by GEMS. GEMS is a set of modules that extends Simics with timing fidelity. It

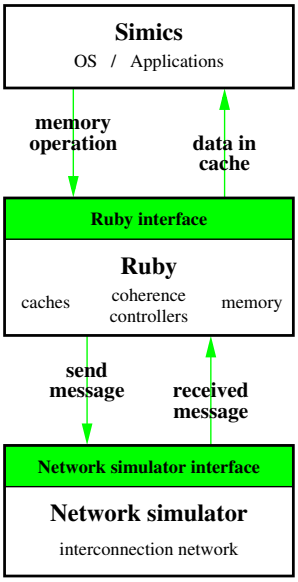


Figure 4.1: A view of the interconnection between the several simulation tools used.

consists of two primary modules (Ruby and Opal), but here we only use Ruby. Ruby models memory hierarchies. However, due to the fact that its interconnection network model was very limited and non-realistic, we extended it with our own simulator, which provides detailed simulation of the interconnection network.

4.1.1 Simics Simulator

Figure 4.1 depicts the overview of the simulation tools used. On the top, we can find Simics. Simics is a full-system functional simulator that allows to boot an unmodified operating system and to execute actual applications. Simics models a simple in-order processor. Simics passes all the load and store instructions and the instruction fetch requests to Ruby, which performs the cache access to determine if the operation hits or misses. On a hit, Simics continues executing instructions. On a miss, Ruby stalls Simics' request from the issuing processor, and then simulates the cache miss. Contention, latency of messages, and other factors will determine when the request completes. By

controlling the timing of when Simics advances, Ruby determines the timing-dependent functional simulation in Simics. Note that Ruby only interacts with Simics to determine when Simics should execute an instruction, whereas the result of the execution of the instructions is determined by Simics.

4.1.2 Ruby Module

Under Simics, we can find Ruby. Ruby is a module for modeling both protocol-independent components (cache arrays, memory arrays, message buffers, and assorted glue logic) and protocol-dependent components (cache controllers and memory controllers).

Ruby uses a queue-driven event model to simulate timing. Cache and memory controllers communicate using message buffers. When the next message is available to be read from the buffer, the recipient is scheduled to wake up.

On a cache miss, Ruby generates the events required by the implemented protocol to solve the miss. When an event implies an exchange of messages through the network, the messages are stalled in the message buffers and they are delivered to the network simulator, which will detailedly simulate their transmissions. Once a message reaches the recipient component, a wake up is scheduled.

4.1.3 Network Simulator

The network simulator is a detailed event-driven interconnection network simulator used as a substrate to communicate between cache and memory controllers. A controller communicates by sending messages to other controllers. The network simulator models the timing of the messages as they traverse the interconnect. It simulates the movement of packets on a per-flit basis, assuming virtual cut-through switching technique. Switches are IQ (Input Queued). Each physical channel is split into several virtual channels. The routing algorithm can use whatever of those virtual channels. The transference of flits inside a switch is performed by means of a crossbar. The crossbar has as many inputs and outputs as physical channels. The queues associated to the same physical channel access to the crossbar by a multiplexer. An input of a multiplexer assigned to a packet will not be able to be assigned to another

packet until the transmission of the first packet completes.

Although the network simulator simulates the packets flit per flit, actually it does not simulate the transference of each individual flit. Like Ruby, it has been optimized considering that the transmission of a packet through a physical channel is not multiplexed. Besides, a packet is only routed when its storage in the next switch is guaranteed. As a result, after routing a packet, the simulator only takes into account the arrival of the header to the next switch, the release of the buffer, and the release of the physical channel. All these events happen after a fixed time once the packet has been successfully routed.

We are considering pipelined channels, since the transmission of flits through the virtual channels is segmented, that is, a flit can be transmitted without having to wait for the previous flit to arrive.

The network simulator holds three variables which will determine the speed of the interconnect (switches and links). *Network link latency* is the fly time of links. *Switch cross latency* is the latency through the crossbar. *Network routing latency* is the time required for a switch arbiter to route a packet. Arbiters use a rotative priority to select the next message to route.

4.1.4 Interconnection between Simulators

The interface provided by the network simulator to connect to Ruby is mainly made up of two functions: *TransmitMessage* and *ProcessMessages*. The function *TransmitMessage* indicates that certain system component wants to send a message. The parameters of this function are composed of the source component, the destination (or destinations) component, the message type, and the message size. On the other hand, the function *ProcessMessages* indicates that all the pending events previous to a certain cycle can be processed. This function has one input parameter (the current cycle) and two output parameters (the messages that have reached at least one destination and the destinations reached).

Ruby and the network simulator are attached in two different points. First, when a cache or a memory controller have to send a message (a request or a token response), the *TransmitMessage* is called. Thus, at that point, Ruby indicates to the network simulator all the details of the message that is going

to be transmitted. While the network simulator simulates the movement of messages throughout the system, the message is stalled in a queue. Once the network simulator determines that a message has completely arrived at its destination, the message is inserted in the destination queue and a wake up is scheduled. Second, after Ruby processes all the events scheduled for certain cycle and before continuing with the events of the subsequent cycle, the function *ProcessMessages* is called. Thus, the network simulator processes all the pending events up to that moment and schedules the wake ups for the next cycle in Ruby (if any).

4.2 Simulated System

To evaluate our proposals, we simulate a multiprocessor server running scientific applications in systems with different interconnects. We simulate a 4-processor, 8-processor, 16-processor, 32-processor, and 64-processor SPARC v9 systems with highly integrated nodes that include split first level instruction and data caches, unified second level cache, coherence protocol controllers, and a memory controller for part of the globally shared memory. Processors are in-order, single-issue processors. The systems implement sequential consistency using invalidation-based cache coherence.

Table 4.1 lists the system parameters for both the memory system and the interconnection network. The parameters related to the memory system are similar to those selected in other works [25, 103, 122] that also simulate shared-memory systems. The only significant difference with respect to the parameters assumed in those works is the main memory latency. We have assumed a latency much lower (80-cycle) than them, which is quite optimistic. The reason to use such a latency is just to speed up the simulations because, as we comment later, running Simics with Ruby and the network simulator will slow down the execution by about 300x. Thus, by using this value, the simulations can run slightly faster.

The simulated systems use an MOESI coherence protocol with the migratory sharing optimization (Section 2.3.5). Coherence is maintained on memory blocks. Unless otherwise specified, all request, acknowledgment, invalidation, and data-less token messages are 8 bytes in size (including the 40 bit phys-

Table 4.1: Simulation parameters.

processors	in-order, single-issue SPARC v9
cache block size	64 bytes
physical address	40 bits
L1 cache	32 KB 4-way set associative split, 3-cycle access latency
L2 cache	8 MB 4-way set associative unified, 6-cycle access latency
coherence protocol	MOESI with the migratory sharing optimization, TokenB
main memory	4 GB, 80-cycle latency
memory controllers	6 cycles
network link latency	6 cycles
switch cross latency	1 cycle
network routing latency	4 cycles
interconnect interface	2 cycles
data-less messages	8 bytes (10 bytes if *completed PR* field)
data messages	72 bytes (74 bytes if *completed PR* field)

ical address and token count if needed). Data messages include this 8-byte header and 64 bytes of data[1]. Messages do not include any extra bits used by the interconnect to detect and correct bit errors. The underlying coherence protocol that we use is a token-based cache coherence protocol developed with Token Coherence based on the TokenB policy. For comparison purpose, we assume the distributed-arbitration implementation of the persistent request mechanism with point-to-point ordering and a fixed-priority scheme.

[1]Some of our proposals need to code additional information in the messages (2 bytes), thereby increasing the message size slightly. These increments in size have been modeled in the simulator.

We chose this configuration because, as shown in [78, 82], it is the most efficient approximation regardless of the system size. Although the banked-arbiter implementation may be interesting in terms of storage requirements of the starvation prevention mechanism at nodes, in terms of performance it is worse than distributed-arbiters since it simultaneously captures the worst aspects of the traditional protocols: persistent requests are sent first to the arbiter (indirection) and, then, they are broadcast. Notice that, the arbiter can not be understood as a directory because only the persistent requests are sent through the arbiters (but not the transient requests) and, therefore, the arbiters can not track memory blocks with total guarantee.

We use the two interconnects described in Section 2.5.1: an MIN interconnect with the perfect shuffle permutation based on 4 x 4 switches and a 2D mesh interconnection network. We selected these two interconnects because they offer the view of two different examples of high-speed point-to-point interconnects, one direct and another indirect. Messages are multiplexed over a single shared interconnect using 3 virtual channels, each of which is used by a different type of message:

- Virtual channel #0 (VC0). This VC is used to transmit response messages (token messages) and resending notifications. They are point-to-point messages[2]. In the MIN interconnect, an adaptive routing algorithm is used to route the messages transmitted through this VC, whereas in a mesh interconnect, the XY routing algorithm is assumed.

- Virtual channel #1 (VC1). Transient requests are assigned to the VC1. On an MIN, a broadcast adaptive routing algorithm is assumed. Given that the transient request messages that we assume are broadcast-based, the messages in this VC always have to reach the last stage of switches. Therefore, they can adaptively go up and then they go deterministically down for all the possible downlinks. On a mesh, a tree-based broadcast is made, being the root of the tree the node which sent the message.

- Virtual channel #2 (VC2). This VC is only used to route persistent/priority requests. On the one hand, since persistent requests require point-to-

[2]In Chapter 8 response messages are multicast messages, therefore the VC0 is dedicated to multicast.

point ordering (according to the implementation we assume throughout the dissertation), the messages in this VC use a deterministic routing algorithm with disjoint paths in order to provide better traffic balance. On the other hand, in case of priority requests, this VC is used to implement an ordered routing algorithm as described in detail in Section 5.2.

When a switch has several messages in different (physical) links that can be routed at a certain time, a switch arbiter decides the next message to route according to a round robin basis. In addition, since each link has several virtual channels, the switch arbiter must also decide the next message to route between the various messages in the VCs of a link. In this case, the arbiter is based on a priority scheme: VC1 doubles the priority of VC0 and, in turn, VC2 doubles the priority of VC1. Thus, for example, if all the VCs of a link contain enough messages, the switch will route 4 messages from VC2, 2 messages from VC1, and 1 message from VC0.

Both interconnects use bandwidth-efficient tree-based multicast routing [41] when delivering messages to multiple destinations. Thus, messages sent to multiple destinations (such as broadcasts and multicasts) use traffic-efficient multicast-based routing to fan out the messages to the various destinations as described in many works [75, 94, 112, 117, 36]. The provided virtual channels avoid routing deadlock and provide several virtual networks. Like many recent systems [32, 67, 89, 116, 77], we assume that these interconnects provide reliable message delivery.

4.3 Performance Metrics

In this dissertation, we evaluate the overall performance of our proposals by measuring the time necessary to complete certain amount of work (runtime). Notice that other works have used instructions-per-cycle (IPC) as a metric to judge performance improvements instead of runtime. Nevertheless, IPC is not a good metric for evaluating the coherence protocols and systems because system timing effects of multiprocessor workloads can change the number of instructions executed and, therefore, running the simulator for a fixed number of instructions and measuring IPC is not guaranteed to reflect the performance

of the system [15]. This happens because the instruction path of multithreading workloads running on multiple processors can vary substantially due to the synchronization mechanisms used by the operative system.

We use the *runtime* of applications to conclude that "protocol A is X % faster than protocol B" using the formula $X = (runtime(B)/runtime(A) - 1.0)100$. To avoid measuring thread forking we begin the measurement at the start of the parallel phase. To this end, we initialize the system state using a full-system checkpoint (to provide a well-define starting point) and simulate the execution until the parallel phase is done. We record the number of cycles needed to complete the parallel phase and we refer to this number of cycles as application runtime.

In addition to reporting runtime, we measure and report the traffic in following terms:

- *Interconnect traffic* (in packets) indicates the amount of controller bandwidth required to handle outgoing messages.
- *Endpoint traffic* (in packets) indicates the amount of controller bandwidth required to handle incoming messages.
- *Link utilization* (in cycles) indicates the amount of link bandwidth consumed by messages as they traverse the interconnect.
- *Starved requests* (in packets) refers to the amount of misses that require the use of the starvation prevention mechanism.
- *Starvation control messages* (in packets) indicates the amount of controller bandwidth required to handle the outgoing messages related to starvation control.

Finally, we measure and report the *starvation latency* (in terms of cycles) which indicates the time required to solve a starvation situation. It includes the elapsed time from a starved request is detected up to its requester completely receives the requested tokens and data.

To address the variability in scientific applications, we adopt the approach of simulating each design point multiple times with small, pseudo-random perturbations of request latencies [13, 14]. These perturbations cause alternative

operating system scheduling paths in our deterministic simulations. For each data point in our results, we present the average of multiple simulations. The number of the simulations will vary between 15 and 25 points depending on the variability. Besides, figures show the 95% confidence intervals for each data.

4.4 Workload Descriptions

Our benchmarks consist of several applications from the SPLASH 2 suite [45]. We have not been able to simulate all the applications from the suite due to their time requirements. The group of selected applications is composed of Barnes, Cholesky, FFT, LU, Ocean, Radix, and Volrend. Following, we briefly describe these applications. More complete descriptions are available at [45].

Barnes [106]. It implements the Barnes-Hut method to simulate the interaction of a system of bodies (galaxies or particles, for example) under the influence of gravitational forces. Each body is modeled as a point mass and exerts forces on all other bodies in the system. The simulation proceeds over time-steps using the Barnes-Hut hierarchical N-body method. Each step computes the net force on every body, updating the body's position and other attributes. The Barnes-Hut algorithm is based on a hierarchical octree representation of space. The root of this tree represents a space cell containing all bodies in the system. Leaves contain information on each body and internal nodes represent space cells. Most of the time is spent in partial traversals of the octree to compute the forces on individual bodies. The communication patterns are dependent on the particle distribution and are quite unstructured. No attempt is made at intelligent distribution of body data in main memory, since this is difficult at page granularity and not very important to performance.

The main data structure in the application is the Barnes-Hut tree. Since the tree changes every time-step, it is implemented in the program with two arrays. Data locality is provided by exploiting physical locality in the problem domain: a partition should be spatially contiguous and, ideally, equally sized in all directions. Such partitions minimize interprocessor communica-

tion and maximize data reuse. The program uses either *Costzones* or *ORB* to provide both load balancing and data locality. Barriers are used to maintain dependences across some phases. ORB partitioning also uses some barriers internally. Another type of synchronization used is mutual exclusion through locks, which are used to protect global variables. Arrays of locks are used to protect cells. Finally, event synchronization using regular variables as flags is used when a cell has to wait for its children.

Cholesky [102]. It performs blocked sparse Cholesky Factorization on a sparse matrix. That is, given a positive definite matrix A, the program finds a lower triangular matrix L such that $A = LL^T$. Cholesky factorization proceeds in three steps: ordering, symbolic factorization, and numerical factorization. The numerical factorization is very efficient, due to the use of supernodal elimination techniques. Supernodes are sets of columns with nearly identical non-zero structures, and a factor matrix will typically contain a number of often very large supernodes. The primary data structure in this program is the representation of the sparse matrix itself. The data sharing patterns for each supernode are as follows. A supernode may be modified by several processors before it is placed on the task queue. Once this happens, it is read by a single processor and used to modify other supernodes. After it has completed all its modifications to other supernodes, it is no longer referenced by any processor.

The only interactions between processors occur when they attempt to dequeue tasks from the global task queue and when they attempt to perform a number of simultaneous supernodal modifications to the same destination column. Both of these cases are handled with locks.

FFT [20]. It is a complex, one-dimensional version of the radix-$\sqrt{n}$ "six-step" FFT optimized to minimize interprocessor communication. The data set consists of the n complex data points to be transformed, and another n complex points referred to as the *roots of unity*. Both sets of data are organized as $\sqrt{n}$ x $\sqrt{n}$ matrices partitioned so that every processor is assigned a contiguous set of rows which are allocated in its local memory. Communication occurs in three matrix transpose steps, which require all-to-all interprocessor communi-

cation. Every processor transposes a contiguous submatrix of $\sqrt{n}/p$ x $\sqrt{n}/p$ from every other processor, and transposes one submatrix locally. The transposes are blocked to exploit cache line reuse. To avoid memory hotspotting, submatrices are communicated in a staggered fashion, with processor i first transposing a submatrix from processor $i+1$, then one from processor $i+2$, etc.

LU [121]. It factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense n x n matrix A is divided into an N x N array of B x B blocks ($n = NB$) to exploit temporal locality on submatrix elements. To reduce communication, block ownership is assigned using a 2-D scatter decomposition, with block being updated by the processors that own them. The block size B is large enough to keep the cache miss rate low, and small enough to maintain good load balance. Two different implementations of LU have been used. In the implementation referred as *LU1* elements within a block are allocated non-contiguously. In the implementation referred as *LU2* elements within a block are allocated contiguously. In both cases, to improve spatial locality benefits, blocks are allocated locally to processors that own them.

Ocean [120]. This application simulates large-scale ocean movements based on eddy and boundary currents. A cuboidal ocean basin is simulated, using a discretized quasi-geostrophic circulation model. Wind stress from atmospheric effects provides the forcing function, and the impact of friction with the ocean walls and floor is included. The simulation is performed for many time-steps until the eddies and mean ocean flow attain a mutual balance. The work done every time-step essentially involves setting up and solving a set of spatial partial differential equations. The equations are set up and solved on grids representing horizontal cross-sections of the ocean basin. Grids are conceptually represented as 4-D arrays, which are partitioned into square-like subgrids allocated contiguously and locally in the nodes that own them. The parallel program uses a red-black Gauss-Seidel multigrid equation solver. The principal data structures are two 4-D double precision floating point arrays.

Mutual exclusion, enforced with locks, is required in: (1) obtaining a process identifier, (2) when every process accumulates its private sum, and (3)

Table 4.2: Application parameters.

Application	Parameters
Barnes	16384 particles
Cholesky	Input file tk29.O
FFT	65536 complex data points
LU	512x512 matrix, block size of 16
Ocean	258x258 ocean
Radix	256K keys, radix of 1024
Volrend	Input file head_scaledown2

when processors communicate though a shared flag. Synchronization is also needed to preserve dependences across grid computations which is accomplished by inserting barriers. Barriers are also used when a globally determined value is subsequently to be used by all processors.

Radix [24]. It implements an integer radix sort based on the method described in [24]. The algorithm is iterative, performing one iteration for each radix r digit of the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram. The local histograms are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. This permutation step requires all-to-all communication. The permutation is inherently a sender-determined one, so keys are communicated through writes rather than reads.

Volrend [91]. It renders a three-dimensional volume onto a two-dimensional image plane using a ray casting technique. The volume is represented as a cube of volume elements and an octree data structure is used to traverse the volume quickly. The program renders several frames from changing viewpoints, and early ray termination and adaptive pixel sampling are implemented, although

adaptive pixel sampling is not used in this study. A ray is shot through each pixel in every frame, but rays do not reflect. Instead, rays are sampled along their linear paths using interpolation to compute the color for the corresponding pixel. The main data structures are the volxels, octrees, and pixels. Data accesses are input-dependent and irregular, and no attempt is made at intelligent data distribution.

The parameters used for each application are shown in Table 4.2. In 64-processor systems, we only run the Barnes, Cholesky, FFT, LU1, LU2, and Radix applications because the simulation time of Ocean and Volrend in such systems is extremely large. Depending on the protocol choice and configuration, running Simics with Ruby with the network simulator will slow down by about 300x, which makes the simulation of Ocean and Volrend difficult.

Chapter 5

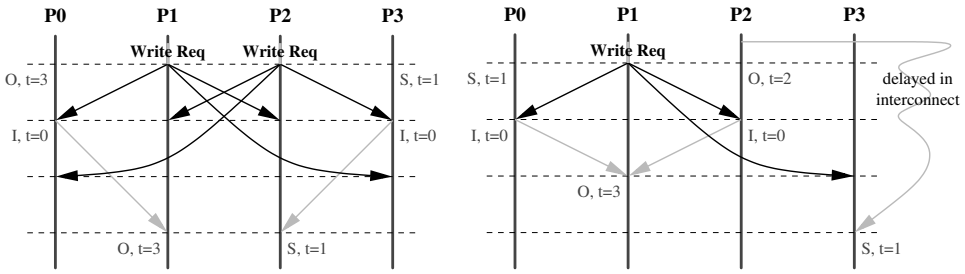
The Priority Request Mechanism

Starvation prevention mechanisms proposed up to now are too strict and inefficient mainly due to two facts: (1) they always override the coherence model (MOESI states) that efficiently handles memory blocks and tokens and (2) they use explicit acknowledgments which increase both network traffic and the protocol latency. In this chapter we propose an alternative mechanism that prevents starvation in such natural and elegant way that (1) the coherence model is not overridden and (2) explicit acknowledgments are not required.

5.1 Introduction

In token-based protocols, two different causes can prevent (transient) requests from succeeding in resolving cache misses:

- **Protocol Races.** A protocol race can occur anytime multiple transient requests contend simultaneously for the same memory block. Since transient requests are unordered messages, the order in which they are received may be different for each component. As a result, all processors do not unambiguously serve transient requests, causing some of them do not succeed in getting the memory block and enough tokens. Figure 5.1(a) shows an example of a protocol race occurrence. In this example, *P1* and *P2* each broadcast a transient write request to collect all tokens,



(a) Protocol Race (b) Tokens Unavailability

Figure 5.1: Examples of transient requests failing to get the requested tokens.

which are shared by *P0* (O state) and *P3* (S state). As transient requests are unordered, *P0* receives the requests in {*P1*,*P2*} order whereas *P3* receives them in {*P2*,*P1*} order. Therefore, *P0* forwards its tokens to *P1* and *P3* forwards them to *P2*. As a result, neither *P1* nor *P2* are able to collect all the block's tokens and, therefore, they can not modify the block. If this situation is not resolved, it will lead to a starvation situation.

- **Tokens Unavailability.** Another situation that can prevent transient requests from completing is when they request tokens that, at that moment, are in transit. Transient requests are served as soon as they arrive. If at their arrival a processor can not serve them because at that moment it does not hold any requested token, then the node discards the transient request. Consequently, transient requests for memory blocks being transmitted through the interconnect will not be able to be served, thereby failing at collecting the necessary tokens. Figure 5.1(b) illustrates an example of token unavailability. *P1* requests all tokens by a transient write request. When it arrives at nodes, they forward all the tokens held at that moment to the requester. However, since there is one token in transit traveling from *P2* to *P3* when the transient request is received, that token is not forwarded to the requester. Therefore, as transient requests are not remembered, when the token in transit finally arrives at *P3*, it does not forward it. As a result, *P1* can not collect all the block's tokens, leading to a starvation situation.

Transient requests can not prevent those situations from happening. Therefore, when they occur, the system has to use a mechanism that resolves those situations, thereby ensuring the completion of all cache misses. That mechanism is known as the starvation prevention mechanism. In this chapter, we propose a starvation prevention mechanism called priority requests which ensures that all attempts to read or write a block will eventually succeed by resolving the aforementioned situations. Thus, when a processor detects that one of its transient requests is involved in a starvation situation, it sends a priority request. Unlike transient requests, priority requests are sure to succeed in resolving cache misses since (1) they can not generate protocol races because they are ordered messages and (2) they are remembered in tables until being sure about their completion which resolves the tokens unavailability problem. Hence, since the two possible situations that can cause starvation are solved by priority requests, they can ensure completion to all cache misses.

The general working scheme followed by processors is as follows:

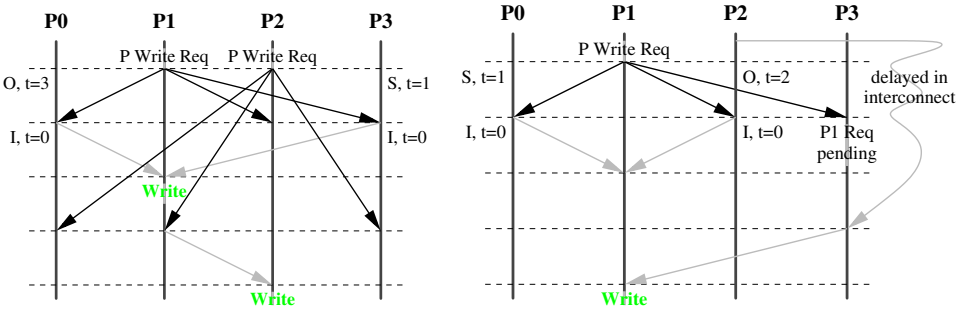
1. When a processor detects possible starvation, it broadcasts a priority request to all system components.

2. Priority requests are transmitted through ordered paths. As a result, all priority requests are received in the same order.

3. At their arrival, components assign a global identifier or priority level (this is why we refer to them as priority requests) and store both the request and its identifier in a priority request table.

4. Components (processors and memory modules) holding tokens requested by any of the stored priority requests (determined by the performance policy) forward just those tokens to the requester with the highest priority that needs them.

5. If a component supplies all the requested tokens, it marks the served priority request as completed. Next, if (1) its table contains one or more not-completed priority requests and (2) it still holds tokens requested by them, it proceeds to serve them until completing all of them or running out of requested tokens.



(a) Avoiding Protocol Races (b) Solving Tokens Unavailability

Figure 5.2: Resolving starvation situations by priority requests.

6. When the starved processor receives all the requested tokens, it performs the memory operation and marks its priority request as completed. Next, if its table contains not-completed priority requests and it holds any of the tokens that they request, it proceeds to serve them.

Figure 5.2 shows how priority requests solve the two starvation situations depicted in Figure 5.1. Figure 5.2(a) illustrates how priority requests avoid generating protocol races thanks to all nodes receive them in the same order. Therefore, since all components receive the priority requests in {*P1*,*P2*} order, all components serve first the *P1*'s request and then the *P2*'s request. Figure 5.2(b) shows that, as priority requests are remembered until being completed, although tokens are delayed in the interconnect, when they finally arrive at their destinations, they will be forwarded to the processors that request them. Next, we give more insight in all the details related to the priority request mechanism.

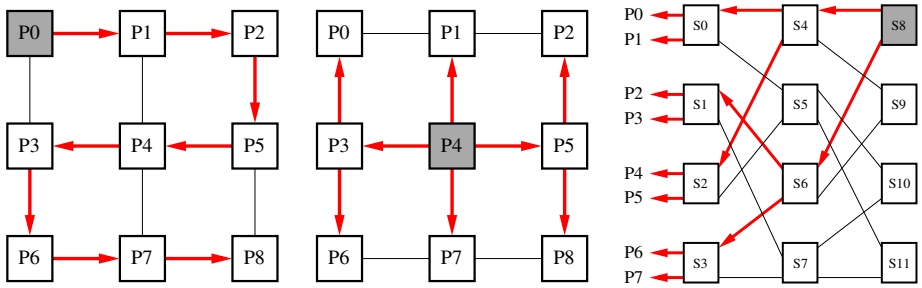
5.2 Ordered Paths

Traditionally two different methods have been used to put requests in order: totally-ordered interconnects and a centralized component that decides the order. However, those methods have serious disadvantages. While totally-ordered interconnects are not scalable, centralized components cause indirection. To achieve order for priority requests without trusting on those methods,

in this section we propose to use the routing algorithm. When required, the routing algorithm can be used to ensure that a set of requests, in this case the priority requests, will be received in the same order by all system components. This routing algorithm can be used over any interconnect. Therefore, we do not limit their applicability to any kind of interconnect. Besides, indirection is avoided since there is no any centralized component that establishes the order. To this end, the routing algorithm of priority requests aims at routing them through *ordered paths*. We define an ordered path as a sequence of switches such that from each of them there is at least one link to the next switch in the sequence. The sequence must include all switches where system components are connected to, since priority requests are broadcast messages. The first switch in an ordered path is called the *root* switch. All messages that follow the same ordered path will be received in the same order.

Routing through ordered paths is divided in two stages. In the first stage, priority requests go from their senders to the root switch. Along this path, priority requests are not delivered to the system components connected to the switches they cross. In the second stage, priority requests go from the root switch to all the switches that form part of the path. In this stage, priority requests are delivered to the system components connected to the switches they cross. Note that, unlike directories, this strategy does not introduce indirection because the root switch does not consume nor process the priority requests. Rather, the root switch is only used to route priority requests through it. Thus, the single difference between the routing algorithms of transient and priority requests is the use of use minimal or non-minimal paths.

Depending on the network topology, there may be several ways to define an ordered path (at least, there will be one). In this dissertation, we consider two different types of interconnects: a direct interconnection network using a mesh topology and a multistage interconnection network (MIN) with the perfect shuffle permutation. On a mesh interconnect, several algorithms can be used to establish ordered paths. One option is to use Hamiltonian paths. A Hamiltonian path is a sequence of switches and links which, from the root switch, establishes a route that visits just one time all the switches in the system. All priority requests traveling along the same Hamiltonian path will be received in the same order by each node. Figure 5.3(a) illustrates an example



(a) Hamiltonian Path (b) Tree in Mesh (c) Tree in MIN

Figure 5.3: Example of ordered paths in interconnection networks with different topologies. The nodes or switches in dark represent the root.

of a Hamiltonian path in a mesh. The switch in gray (*P0*) represents the root switch and the arrows indicate the path to reach all the system nodes.

Although Hamiltonian paths successfully ensure a global order, they present a clear disadvantage: nodes are visited one by one. Whereas in small systems this will not be a serious problem, in medium/large systems it will be too inefficient. Alternatively, there exist other more efficient algorithms to define ordered paths in a mesh, such as spanning trees. A spanning tree is a graph in which any two switches are connected by exactly one path. The spanning tree includes all the switches in the network, but it can not contain any cycle. Unlike Hamiltonian paths, spanning trees make an excellent use of the network bandwidth and they are scalable. Figure 5.3(b) shows an example of a spanning tree in a mesh. In this case, *P4* is the root. Like in Hamiltonian paths, priority requests go from their issuer to the root switch and, from there, they are delivered to all nodes by following the spanning tree indicated by the arrows. As you can observe, in this case, the ordered path is much shorter than the Hamiltonian path. Whereas in the Hamiltonian path the average number of hops to reach all the system nodes is 4 (the maximum is 8), in the spanning tree the average number of hops is 1.33 (the maximum is 2).

The ordered paths in MINs do not require to include all the network switches, since system components are only connected to the switches in the first stage. Taking into account that from whatever switch of the last stage all system components can be reached, the root switch of the ordered path

must be chosen among those belonging to the last stage. The ordered path will consist of all the switches that can be reached using the downlinks (links connected a switch of a lower stage). Note that this routing algorithm is an extension of the up/down routing algorithm [104] commonly used in MINs. Figure 5.3(c) shows an example of an ordered path in an MIN interconnect. In this case, the root is the *S8* switch. Following all the downlinks from the root, all the system components can be reached.

Since the routing algorithm used by priority requests may differ from that used by another kind of messages, they may require a dedicated virtual channel to ensure a deadlock-free routing scheme. Thus, for example, the routing algorithm for Hamiltonian paths differs a lot from the XY routing [114] commonly used in mesh networks. Therefore, in that specific case, priority requests would require a dedicated virtual channel. On the contrary, the routing algorithms for spanning trees in meshes or in MINs do not differ from that used for non-priority requests. Rather, the routing algorithm is the same, being the only difference the use of minimal or non-minimal paths. Therefore, in those cases, an additional virtual channel would not be required.

5.3 Priority Request Table

By putting priority requests in order, the problem of protocol races is resolved. However, that does not resolve the token unavailability problem. To solve it, priority requests are remembered in tables, at least, until being sure about their completion. Thus, if a priority request can not be served at its arrival because the tokens that it requests are in transit, the priority request will be remembered. When the transmission of the tokens in transit finishes, the recipient will realize that there is a pending priority request that needs those tokens and it will forward them. To implement this strategy, like in the persistent request mechanism, each system component has a *priority request table* where priority requests are stored at their arrival. Once they complete, they can be removed from the tables.

Each table entry stores the information about a certain priority request. Specifically, each entry consists of the following fields:

- **Valid.** It is a bit that indicates whether the entry is valid or not.

- **Issuer.** It is the issuer's identifier (processor number).
- **Address.** It is the physical address of the requested memory block.
- **Identifier.** It is the priority level (arrival order) of the priority request. The arrival order can be used to unequivocally identify it.
- **Operation.** It distinguishes between read or write requests.
- **State.** It indicates if the priority request is completed or pending.

When a component receives a priority request, it stores the required information in its table: the processor requester is inserted in the *Issuer* field, the requested memory block address is inserted in *Address*, the arrival order is stored in *Identifier*, the *Operation* bit is set according to the request type (read or write), the *State* field is set to pending, and the *Valid* bit is set to 1.

In order to the mechanism works perfectly, all the received priority requests must be able to be stored in the tables. Since a lot of priority requests can be sent and tables of infinite size can not be used, we rely on limiting the maximum number of simultaneous outstanding priority requests per processor (like the distributed-arbitration implementation of persistent requests). By limiting the number of outstanding priority requests per processor, for example, to one, each table will have as many entries as the maximum number of processors in the system. Assuming, for example, a system with 64 processors, the size of each table entry would be approximately 10 bytes. Therefore, in that case, each system component would require a 640 bytes table.

Assuming just one outstanding priority request per processor, each table entry corresponds to exactly one processor and, therefore, the table entries do not need to explicitly encode the number of the processor that issued the request, as the processor number becomes the index for the table. Since this table can have multiple entries that contain the same address, the entry for the request with the lowest value in the *Identifier* field and marked as pending corresponds to the request with the highest priority.

```
boolean pause(new_priority, oldest_priority)
    diff = new_priority – oldest_priority
    if diff > MAX_COUNT/2
        return diff <= MAX_COUNT
    else if diff <= –MAX_COUNT/2
        return diff <= –MAX_COUNT
    else
        return diff <= 0
```

Figure 5.4: Algorithm for determining when a recipient should pause the generation of new priority requests. If the *pause()* function returns *true*, the recipient postpones the sending of new priority requests until receiving an acknowledgment indicating that it can continue. In the pseudo-code above, *new_priority* is the identifier assigned to the incoming priority request, *oldest_priority* is the identifier of the oldest priority request, and *MAX_COUNT* is the maximum identifier allowed.

5.4 Priority Request Identifier

System components assign each received priority request a priority level, which will be used to (1) unequivocally identify it and (2) know the order in which priority requests must be served. Since this identifier is related to the arrival order, the lower the request identifier is, the higher priority the request has. To determine the identifier, each system component uses a local counter. Initially, the counter is set to 0. When a new priority request is received, the value of the counter at that moment is assigned to it (by storing it in the *Identifier* field of the table) and the counter is increased in 1. Since all priority requests are received in the same order, all system components will assign the same value to each request.

The sequence numbers provided by the counters are large enough, in practice, so that priority requests always have unambiguous identifiers. Thus, by using a 2-byte counter, the identifier space would wrap around every 65536 priority requests. Therefore, once a priority request is received, it should be

completed and removed from the tables before finalizing the completion of the subsequent 65535 priority requests. Otherwise, two different priority requests could be assigned the same identifier, leading to an error. Taking into account that priority requests are completed in order, that situation is very unlikely to happen. However, if a priority request took that long to complete and be removed from the tables, the processors should temporarily pause the generation of new priority requests until the delayed one was completed and removed from the tables. To determine when a processor must postpone the sending of a new priority request and to correctly handle the identifier wrap-around, system components use an algorithm based upon algorithms for handling finite sequence numbers in retransmission schemes. This algorithm is given in Figure 5.4. When a processor receives a priority request and the *pause()* function returns *true*, it will postpone the sending of new priority requests (if any) until receiving an explicit acknowledgment that informs about the completion and removal of the delayed request. To this end, the processor that issued the delayed priority request will broadcast an acknowledgment which will remove from the tables the delayed priority request. After that, the mechanism continues working normally.

We want to point out that, although that situation could happen, it would be very unlikely. In fact, we have never detected such a situation to happen in any of the simulations performed in the dissertation or the published papers.

5.5 Removing Completed Priority Requests

According to the persistent request mechanism, when a persistent request is completed, its issuer broadcasts an acknowledgment (or deactivation message) to remove that request from all nodes' tables. These acknowledgments are broadcast messages. Therefore, their use significantly increases the network traffic and slows down the starvation prevention process. To avoid such disadvantages, the priority request mechanism does not use explicit acknowledgments to remove priority requests from tables[1]. However, if priority requests are not removed, the storage of a new received request could cause tables to

[1] Only in the exceptional and unlikely case commented in the previous section an explicit acknowledgment would be required.

destination	type	size	requester	address	operation	completed PR
1 byte	2 bits	1 byte	1 byte	4 bytes	1 bit	2 bytes

Figure 5.5: Format of priority request messages.

overflow. To avoid it, our mechanism implements a simple strategy that ensures that all priority requests will be able to be stored without overflowing tables. This strategy takes advantage of the limitation in the maximum number of simultaneous outstanding priority requests per processor. According to this limit, each processor has a fixed number of entries associated to it in each table where its priority requests will be stored. When the limit is one priority request per processor, each processor will have just one associated entry in each table. Therefore, in that specific case, the strategy to store priority requests is straightforward, since each priority request can only be placed in the entry associated to its issuer. However, when the limit is two or more priority requests per processor, each processor has more than one associated entry. As a consequence, the issuer needs to decide in which of all its associated entries its priority request will be stored, since a priority request only can be replaced when it completes. To this end, the header of the priority request messages themselves includes a field in which the sender indicates which of all its priority requests has already been completed and, therefore, can be replaced. We will refer to this field as the *completed PR* field. The format of priority requests is shown in Figure 5.5. The format is similar to that of persistent requests assumed in other works [76]. The single difference is the presence of the *completed PR* field. *Destination* stands for the home memory module of the requested memory block. The *type* field is used to code the message type: 0 for responses, 1 for transient requests, and 2 for persistent/priority requests. The other fields are defined as in [76].

To suitably set the *completed PR* field, before broadcasting a priority request, the sender looks in its table for a valid entry that contains a priority request issued by itself and marked as completed. Notice that once a priority request has been completed (the issuer always knows its completed requests), it is not necessary to maintain such information in the table and, therefore, its occupied table entry can be removed or replaced. Thus, the issuer includes the

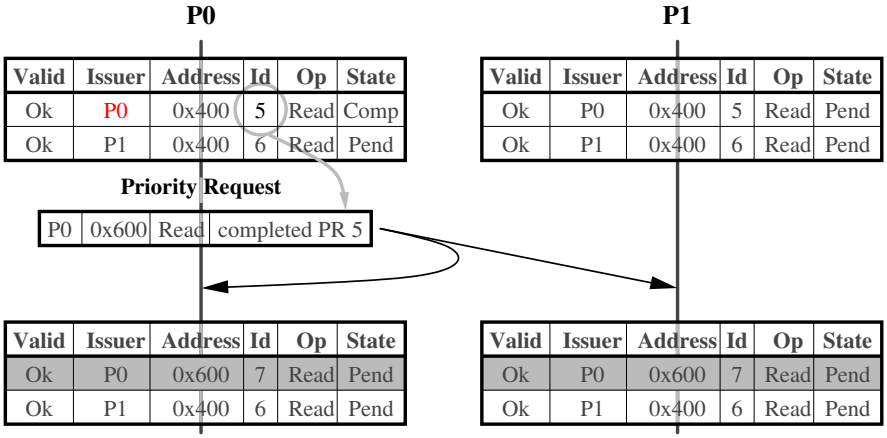


Figure 5.6: Example of attached information in priority request messages.

Identifier field of that entry in the *completed PR* field of the priority request that is going to send and the table entry is marked as invalid. The issuer will always find such information because of the imposed limit in the number of outstanding request allowed. This limit ensures that, if a processor reaches the limit, it will not be able to send a new priority request until one of the previous ones has been completed.

When system components receive a priority request, they look in their tables for an entry whose *Identifier* field matches the *completed PR* field of the received request. Once it is found, they replace that entry with the information about the received priority request.

Figure 5.6 shows an example of how this mechanism works. We assume a system with 2 processors and only one outstanding priority request per processor (which indicates that a processor is not able to broadcast a priority request until completing its prior priority request). Each system component (in the figure we only show the processors to simplify the example) holds a priority request table with two entries (one entry per processor). Initially, tables are full. Now, let us assume that *P0* broadcasts a priority read request. In its header, it indicates the priority request to replace in the tables. To this end, before sending it, *P0* looks in its table for a valid priority request issued by itself that it is marked as completed. It finds that its priority request with identifier 5 has already been completed, proceeding to include such an

identifier in the *completed PR* field. When system components receive the *P0*'s priority request, they look in their tables for an entry holding a priority request with the identifier indicated by the *completed PR* field and replace that entry with the information about the new received priority request.

Note that when a processor is composing a priority request and it finds the information required for the *completed PR* field, the table entry holding such information must be marked as invalid. This is done to prevent such an identifier from being included in another priority request issued by the local processor, which would cause the mechanism to fail.

5.6 Avoiding Serving Completed Priority Requests

The acknowledgments (or deactivation messages) sent by the persistent request mechanism not only are used to remove persistent requests from tables, but they also inform about the completion of persistent requests, which avoids serving again the completed persistent requests. Since the priority request mechanism is not based on explicit acknowledgments, a strategy must be used to avoid serving again the priority requests that have already been completed.

Let us analyze separately the situations for a priority read request and for a priority write request. When a priority read request is broadcast, only the owner processor[2] will be in charge of serving it. Thus, when that processor receives the request, it is stored in the table and served it by sending a data response (a copy of the requested memory block and one token) to the requester, next marking it as completed. The rest of processors (different from the owner) will keep their tokens and will not need to serve it. Therefore, they will not need to know about the completion of the priority read requests, as they can keep their tokens. Nevertheless, if any of those processors becomes in the future the owner, it will need to know **at that moment** about the completed priority read request (to avoid serving it again). To this end, when the owner processor forwards the owner token to another component, it will indicate in such a message the priority requests that have already been completely served. In that way, when the recipient receives the owner token, it will realize of the priority requests that do not need to be served.

[2]We refer to the owner processor as the processor holding the owner token.

On the other hand, when a priority write request is broadcast, all the processors holding tokens will be in charge of serving it. Therefore, all of them send their tokens and, when the requester has received all of them, it will mark its priority request as completed. Unlike in the previous case, only the issuer of a priority write request will be sure about its completion. The rest of processors may know it or may not. Note that, since the issuer of a write request must receive all the block's tokens, it will become the owner processor and, therefore, it will be in charge of serving the requests that arrive later. Since the rest of processors may not know anything about the completion of the priority write request, the owner processor (that is, the issuer of the write request) should inform about it each time it sends a token to another processor. Otherwise, a processor receiving a token could serve again the priority write request whose its completion has not been informed about.

From the analysis of those two situations, we conclude that, to avoid serving again the completed priority requests, all the responder processors should indicate in the responses the priority requests that have already been completely served. To this end, the header of response messages includes a field (*completed PR*) that indicate the completed priority requests. To appropriately set the value of that field, before sending a response, the responder looks in its table for the priority request (belonging to the same memory block as the token/s included in the response) marked as completed and with the highest identifier. That priority request will be the last stored request at being completed and its identifier will be included in the *completed PR* field of responses. Taking into account that priority requests are completely served in order, it is sure that all the priority requests previous to that indicated by *completed PR* have already been completed too. Therefore, *completed PR* does not only indicate one completed request, since it also indicates that all the previous ones have already been completed too. Thus, when a component receives a response message, it will know that all the priority requests with an identifier lower or equal to that indicated by *completed PR* have already been completely served. Hence, on a response reception, processors first update their tables by marking as completed all the priority requests indicated by *completed PR*, then consume the tokens, and finally forward them to the pending priority requests (if any). This ensures that each priority request will be served just

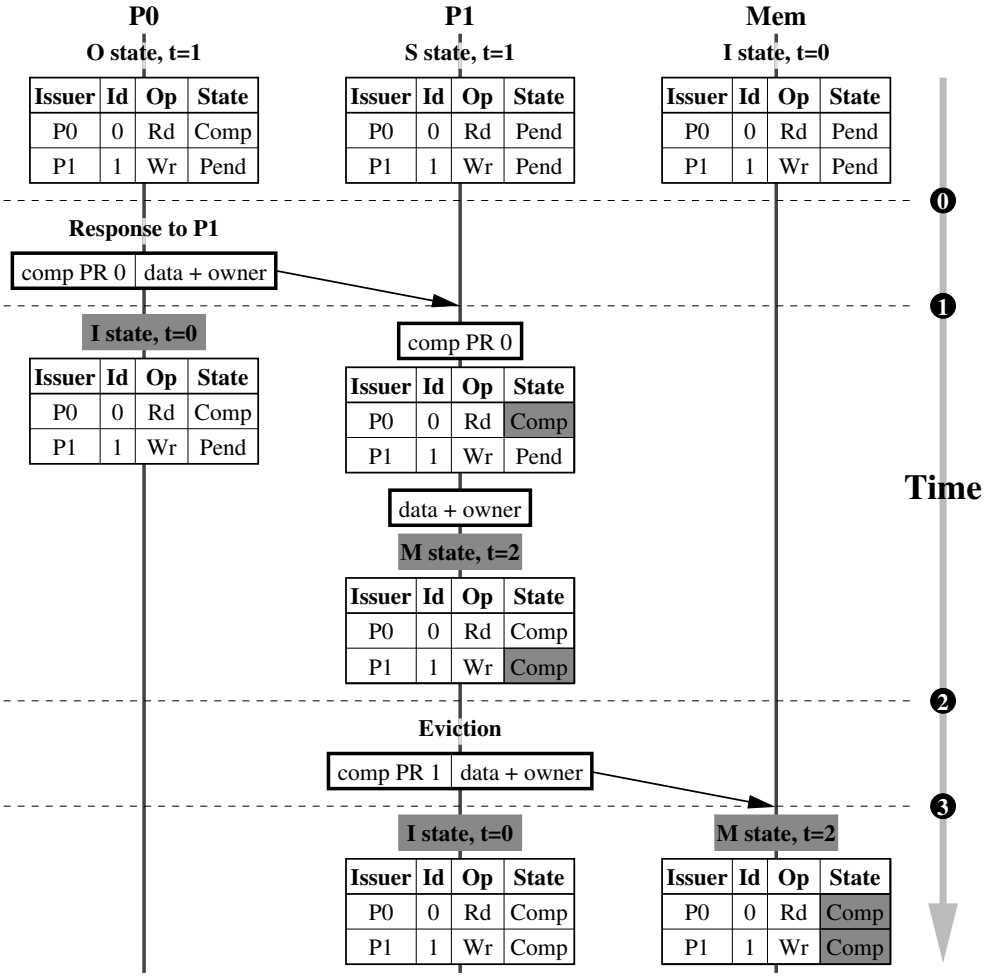


Figure 5.7: Example of *completed PR* field in data responses.

once in spite of not using explicit acknowledgments.

Figure 5.7 shows an example of how this mechanism works. In this example, the system is made up of two processors (*P0* and *P1*) and a memory module (*Mem*). Initially, *P0* is in O state, since it holds the Owner token; *P1* is in S state, as it holds one no Owner token. Finally, *Mem* is in I state, since it does not hold any token. As shown in *P0*'s table, it issued a priority read request that is already completed. However, *P1*'s priority write request is still pending. Since *P0* is in charge of serving it and its table does not hold any pending request with a higher priority, it forwards all its tokens to *P1*. In

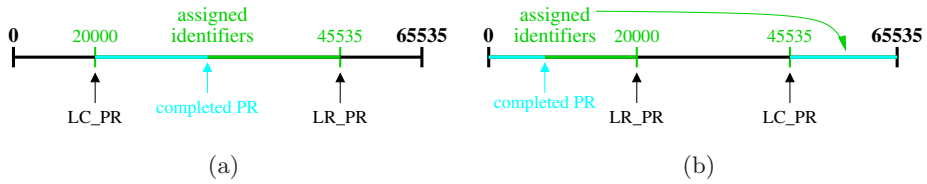


Figure 5.8: Managing the wrap-around when the completed priority requests are updated. *LC_PR* stands for the identifier assigned to the Last Completed Priority Request, whereas *LR_PR* stands for the identifier assigned to the Last Received Priority Request.

the *completed PR* field of the response message, *P0* indicates that its priority request with identifier 0 is already completed. When *P1* receives the response message, it uses the information in *completed PR* to update its table, thereby marking as completed all the priority requests with an identifier equal to or lower than that received in *completed PR*. Hence, *P1* marks the *P0*'s request as completed. Note that if *P0* had not included such information, when *P1* had received the response, it would have thought that the *P0*'s request was pending and, since it has higher priority than its own request, it would have unnecessarily served it again. Once *P1* updates its table, it realizes that its own priority request is the one with the highest priority. Since it has received all the tokens that it requires, it performs the write and marks its request as completed. Later, *P1* sends its tokens and the data back to memory due to an eviction. In that response message, *P1* indicates that its priority request with identifier 1 is completed. Therefore, when *Mem* receives the response message, it realizes that the priority request with identifier 1 and all the previous ones are completed, updating its table accordingly. After the updating, since the *Mem*'s table does not contain any pending request, the memory keeps the tokens and an updated copy of the data.

Figure 5.8 shows how the wrap-around is managed to correctly update the state of priority requests by using the *completed PR* field of responses. Thus, if $LC\_PR < LR\_PR$ (Figure 5.8(a)), all the priority requests with an identifier lower than *completed PR* will be marked as completed. When $LC\_PR > LR\_PR$ and $completedPR > LC\_PR$, all the priority requests with an identifier higher than *LC_PR* and lower (or equal) than *completed PR* are marked as

completed. Finally, if $LC\_PR > LR\_PR$ and $completedPR < LR\_PR$ (Figure 5.8(b)), all the priority requests with an identifier lower than *completed PR* and all the priority requests with an identifier higher than LC_PR are marked as completed.

5.7 Coding Identifiers in Messages

The *completed PR* field adds an overhead to response messages. On the one hand, the overhead for data messages is low because the identifier coded in *completed PR* is much smaller than data blocks. As a result, the size of data messages increases in less than 3%. On the other hand, the data-less token messages are significantly smaller than messages that carry data. Therefore, the overhead of encoding the *completed PR* field is also larger. For our systems assumptions, data-less token messages are eight bytes in size. By increasing in two bytes the size results in 25% overhead. Fortunately, data-less token messages are usually used together with a data message (probably sent by another node). Therefore, when data-less token messages are paired with data messages, the amortized overhead is approximately 6%.

To alleviate the overheads caused by the inclusion of the *completed PR* field in response messages, following we propose several options.

The first alternative is that all the response messages contain the *completed PR* field. This is the safest alternative since it is sure that the components will always receive updated information together with the tokens. However, it is the alternative that causes the largest overhead, since all responses include the *completed PR* field.

Another alternative is that only the data messages contain the *completed PR* field. Thus, the overhead caused by the inclusion of the *completed PR* field in responses is smaller. Although this is a correct solution and the starvation prevention is not at risk, some scenarios may cause the generation of unnecessary data-less responses, thereby increasing the network traffic. This happens when a component receives a data-less response and its table contains a priority request that has already been completed but whose completion it is not aware of.

Figure 5.9 shows an example where such a situation happens. Initially,

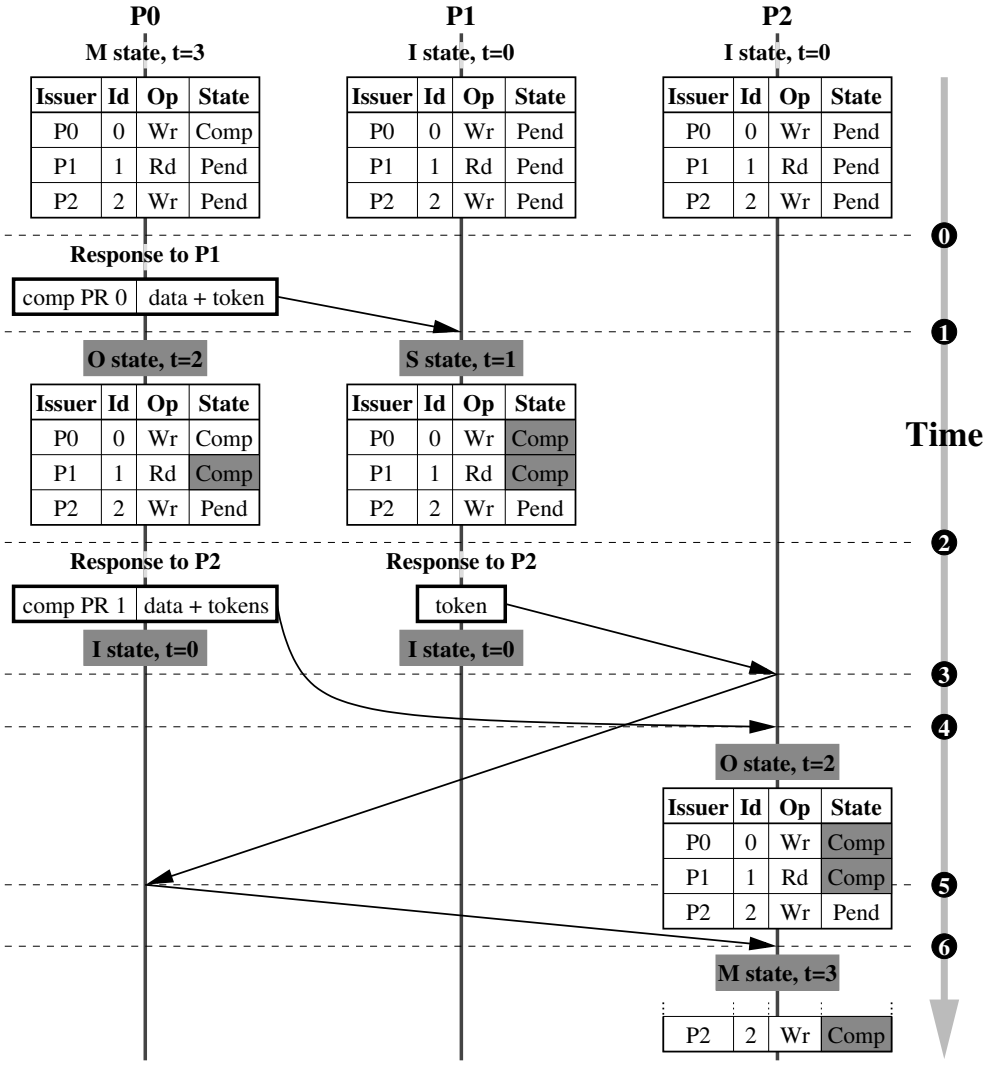


Figure 5.9: Example of unnecessary data-less responses.

there are three processors and each one has sent a priority request. *P0*'s priority request has just been completed because it has received all the requested tokens and, therefore, its priority request is marked as completed in its table. Since its request is completed and the highest priority request that is pending is the *P1*'s read request, at time 0 *P0* serves the *P1*'s request by sending it a response. The response includes the data, one token, and the *completed PR* field set to 0. Since *P0* completely serves the *P1*'s request, it marks that request

as completed and proceeds to serve the next pending request, that is, *P2*'s request. Therefore, *P0* sends to *P2* a response which includes all its tokens (because it is a write request), the data block, and the *completed PR* field set to 1. In this case, *P0* does not marks the *P2*'s request as completed because it has not been able to sent all the requested tokens. At time 1, *P1* receives the *P0*'s response. First, it updates its table by marking the *P0*'s request as completed (thanks to the *completed PR* field) and then it processes the data and tokens, marking its request as completed too. Now, the pending request with the highest priority in the *P1*'s table is the *P2*'s request. Therefore, *P1* sends all its tokens to *P2*. Since it sends a data-less response, it does not include the *completed PR* field. Due to network congestion, we suppose that *P2* receives first the *P1*'s response and later the *P0*'s response. However, when *P2* receives the *P1*'s response at time 3, *P2* is not aware of the *P0*'s request completion. Besides, since the *P1*'s response does not indicate anything about it, at time 3, when *P2* receives the *P1*'s response, it bounces it to the issuer of the pending request with the highest priority that, according to *P2*'s table, is *P0*. At time 4, *P2* receives the *P1*'s response. Thanks to the *completed PR* fields it realizes that *P0*'s and *P1*'s requests have already been completed and marks them in its table. When *P0* receives the bounced *P2*'s response at time 5, it bounces it again to the processor with the highest priority (according to its table), that it, to *P2*. At time 6, when *P2* receives the response, it holds all the block's tokens and performs the memory operation, marking its request as completed. As shown by this example, if the data-less responses do not include the *completed PR* field, the starvation prevention still works correctly, but some unnecessary data-less responses may be generated.

Finally, the third alternative is that all the response messages that carry at least one token associated to any of the priority requests in the table (marked as valid) contain the *completed PR* field. However, responses for transient requests upon memory blocks not requested by (completed) stored priority requests can omit the *completed PR* field. The main advantage of this option is that only the responses that require to inform about the completion of priority requests will do it. But the responses that do not require it will not include such information. Therefore, the overhead due to the *completed PR* field would decrease.

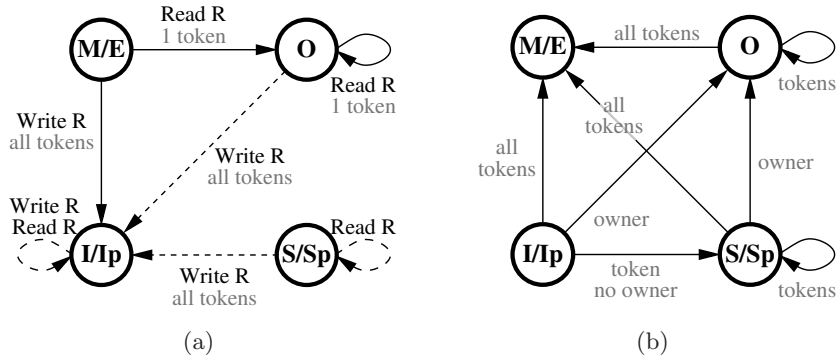


Figure 5.10: State transition diagram (a) due to request reception and (b) due to response reception.

5.8 The Performance Policy

Although several performance policies have been defined for Token Coherence, in this dissertation we use the TokenB policy adapted to priority requests. We use this policy because it avoids indirection and the performance bottlenecks of totally-ordered interconnects. The final version of TokenB is as follows:

- On a cache miss occurrence, the processor broadcasts a transient request to all processors and the home memory module.
- If after twice the processor's average miss latency the transient request has not been completed, the processor will broadcast a priority request to all processors and memory modules.
- Processors and memory modules respond to both transient and priority requests as they would do in a traditional MOESI protocol.

Figure 5.10(a) shows the state transition diagram due to the reception of transient/priority requests. Together with each transition, it is indicated whether the processor has to serve the request by sending a response message (in gray). *Read R* refers to read requests and *Write R* refers to write requests. The dotted arrows indicate that from that state, the received request can not be completely served. Figure 5.10(b) depicts the state transition diagram due to the reception of response messages.

The two additional states *Sp* and *Ip* are equivalent to *S* and *I*, respectively. The single difference is that the states marked with the p subscript (priority states) indicate that there is, at least, one pending priority request in the table. In Figure 5.10(a), from non-priority states (*M*, *O*, *S*, and *I*), processors transition to non-priority states in case of (1) receiving a transient request or (2) receiving a priority request and being able to complete it (solid lines). From non-priority states, processors transition to priority states (*Sp* or *Ip*) when receiving a priority request and not being able to complete it (dashed lines). A block transitions between priority states until (1) an implicit acknowledgment (in a response or in a priority request) marks as completed all the pending priority requests for that block or (2) the processor receives enough tokens to completely serve all the pending priority requests for that block.

5.9 Guaranteeing Starvation-Freedom

To probe that this mechanism prevents starvation, we argue that (1) the system will eventually deliver all messages, (2) priority requests are received in order, which avoids the occurrence of protocol races, (3) priority requests are remembered in tables at least until being completed, (4) the pending request with the highest priority will always receive all the tokens that it requires, and (5) all priority requests will eventually be marked as completed.

5.9.1 Deadlock-Free Message Delivery

We trust on reliable, deadlock-free, and livelock-free interconnects, which ensures that all the messages injected in the interconnect will eventually be delivered. To provide such features, the interconnect may require some well-understood techniques [41] such as virtual channels. Since several types of messages are used and each type may require a different routing algorithm, the transmission of all the messages through the same virtual channel may cause deadlock. Thus, to avoid it, the implementation of the Token Coherence protocol assumed in this thesis uses three different virtual channels: the first one is used for response messages, the second one is for transient requests, and the third one is for priority requests. Given that each type of message may require a different routing algorithm, the use of three virtual channels

will prevent deadlock. To ensure a fair policy, a round robin policy (explained in Section 4.2) is applied between the different virtual channels.

5.9.2 Ordering Delivery Guarantee

Definition 1 An ordered path is defined as a sequence of switches and links, where each pair of switches is connected through a single path that does not contain any cycle.

Lemma 1 All the components in the system will receive priority requests in the same order.

Proof 1 Given that an ordered path only provides a single path between each pair of switches, a priority request being transmitted between two switches will never be able to overtake another priority request that is being transmitted through those switches, provided a FIFO policy is applied at switch buffers. As a result, those pair of switches will receive all the priority requests in the same order. As this is applicable to any pair of switches in the network, a global order for the reception of priority requests is guaranteed. ■

5.9.3 Priority Request Storage Guarantee

Lemma 2 All the received priority requests will be able to be stored in the tables.

Proof 2 The priority request mechanism limits the number of simultaneous outstanding priority requests per processor. Each component holds a table which has as many entries as total number of simultaneous outstanding priority requests in the system. Therefore, if an outstanding priority request can not be stored, it is because all the entries associated to that processor are occupied by outstanding priority requests from it. However, that is not possible because in such a situation, it would have broken the limitation in the maximum number of outstanding priority requests per processor. Thus, if a processor has occupied all its associated entries with outstanding priority requests, it will not be able to send another one until completing one of them. When it is completed, it will be able to send a new priority request, which will always be able to replace the completed one. ■

5.9.4 Requested Token Reception Guarantee

Lemma 1 Given any two consecutive priority requests, the proposed mechanism ensures that both requests will always receive all the tokens they request and it is also sure that they both will be marked as completed.

Proof 1 To demonstrate that whatever pair of priority requests will receive all the requested tokens and will be marked as completed, we analyze the four possible combinations for two consecutive priority requests:

- *read - read.* The owner component provides one token and the data to the first read and marks it as completed. After that, the second read request becomes the pending request with the highest priority. Therefore, the owner serves it and marks it as completed. In the second response, the owner component indicates that the first read request was completed. It knows such information because it was the component which completed it.

- *read - write.* The owner component provides one token and the data to the first read request and marks it as completed. After that, the write request becomes the highest priority request. Therefore, the owner serves it by sending all its tokens. In this response, it is indicated that the read request was completed. When the write requester receives the response it realizes that the read request is completed and, therefore, it does not serve it. Instead, it waits for the rest of tokens. When the read requester receives the token, it performs the read, marks its request as completed, and the write request becomes the highest priority request. Therefore, it serves it by sending its token. Once the write requester receives all the tokens, it will mark its request as completed.

- *write - read.* The owner and the shared components forward all their tokens to the write requester. When the recipient receives all of them, it performs the write and marks its request as completed. The read request becomes the pending request with highest priority. Therefore, the writer, which is the new owner, sends one token and the data to the read requester indicating that the write is completed and marking the read request as completed. When the response is received by the

Table 5.1: Number of links used by priority requests in an 8-processor system with an MIN interconnect.

Number of Ordered Paths	Number of Used Links
1	6 (37,5%)
2	8 (50%)
3	14 (87,5%)
4	16 (100%)

read requester, the write request is marked as completed, the read is performed and its request is marked as completed too.

- *write - write*. The owner and the shared components forward all their tokens to the first write requester. When they are received, the write is carried out and the first write request is marked as completed. The writer becomes the new owner and it sends all the tokens to the requester of the request with the highest priority (the second write). In the response it indicates that the first write is completed. Thus, when the requester of the second write receives all the tokens, it marks the first request as completed, performs the write, and marks its write request as completed.

From the analysis of these four situations we deduce that every pair of priority requests will be completely served and marked as completed. Thus, we can extend this result to a set of priority requests, thereby demonstrating that all priority requests will be completely served and marked as completed. ■

5.10 Using Several Ordered Paths

A single ordered path for all the priority requests may cause the root switch to become a bottleneck in large systems. Fortunately, a system can use multiple ordered paths to scale priority request throughput at the cost of increasing the complexity of the routing algorithm. The proposed priority request mechanism only requires the requests for the same memory block to be received in the

same order because those requests could generate protocol races when they are received in a different order. However, priority requests for different memory blocks can not generate protocol races and, therefore, they do not need to be received in a global order. We can take advantage of this fact to establish several ordered paths instead of just one. Thus, depending on, for instance, the requested memory block, the priority requests can use different ordered paths. This provides two main advantages. On the one hand, several root switches can be used, which alleviates the bottleneck of a single root. On the other hand, the traffic due to priority requests can be distributed through all the network instead of traversing only a reduced part of the network. Table 5.1 shows the number of links used by priority requests in an 8-node MIN interconnect. When priority requests use only one ordered path, all priority requests are transmitted through a single root switch and they only use 37,5% of the links. If priority requests use two ordered paths, they would be transmitted through two different root switches, which can alleviate the congestion of a single root. In addition, priority requests would use 50% of the links. The best scenario in that system corresponds to use four different ordered paths, since they would use the four possible root switches and 100% of the links.

5.10.1 Selecting the Ordered Path

The issuer of a priority request decides the ordered path that its priority request will follow before sending it. All processors must use the same criteria to ensure that all the priority requests for the same memory block use the same ordered path. Otherwise, processors could receive the priority requests for the same memory block in a different order. The decision of what ordered path is used by each priority request can be based on several criteria. Thus, for instance, the ordered path can be selected according to the home memory module which the requested memory block belongs to. In this case, all the priority requests for a memory block belonging to the same home memory module would use the same ordered path. Another alternative is to select the ordered path according to the memory address of the requested block. Processors use the following formula to obtain the path:

$$(memory\_address/block\_size)\%num\_paths$$

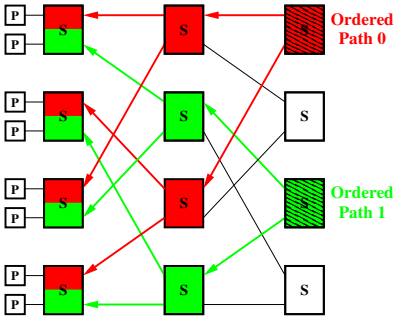


Figure 5.11: Two different ordered paths. Striped boxes represent the root switches of the ordered paths.

memory_address stands for the address of the requested memory block, *block_size* is the size of each memory block, *num_paths* is the number of different ordered paths, and % refers to the module operation. This expression provides the processors the identifier of the ordered path through which the processor must send the priority request. Processors include such an identifier in a field of the priority request header. Network switches will use it to route priority requests through the specified path. Figure 5.11 shows an example of an MIN interconnect where two different ordered paths are defined.

5.10.2 Priority Request Identifier

When using a single ordered path, all the priority requests arrive in a global order. In that case, each component uses a counter to determine the arrival order and to unequivocally identify each priority request. When several ordered paths are used, components can not use a single counter for all the priority requests as the reception order between priority requests following different ordered paths may differ. Hence, in this case, components must use as many counters as the number of different ordered paths. In this way, when a component receives a priority request, it assigns to it an identifier determined by the counter associated to the ordered path followed by the priority request.

5.10.3 Storing Priority Requests

Previously we commented that the *completed PR* field of priority requests is used to select the completed priority request stored in tables that must be replaced by the new one. However, two different priority requests can have now the same identifier, since different counters have been used for generating them. Therefore, this entails a problem because it is possible that at the arrival of a priority request a component does not know which priority request must be replaced. To solve it, we use a different approach from that proposed in the previous section. As commented, each processor is assigned a fixed number of entries where its priority requests can be stored. Therefore, instead of using the *completed PR* field to include the identifier of a completed priority request, that field indicates which of all the entries associated to the processor will be where the priority request is inserted in. Thus, for example, assuming two outstanding priority requests per processor, the *completed PR* field can be set to 1 to indicate that the request must be stored in the first entry associated to the issuer or it can be set to 2 to indicate that the request must be stored in the second entry associated to it. This strategy solves the aforementioned problem and, in addition, it reduces the overhead of the *completed PR* field, since the number of maximum simultaneous outstanding priority requests per processor is much lower than the maximum count.

Given that two different priority requests from the same issuer and for different blocks may be received in different order (because they may follow different ordered paths), a strategy must be used to avoid that a completed priority request replaces a outstanding one. To this end, the header of the priority request messages includes the *SEQN* field which is a sequence number generated by the issuer. A priority request with a low *SEQN* can not replace another priority request with a higher one.

5.11 Discussion: Persistent Vs Priority

In this section we provide deeper insight into the priority request mechanism showing how it is able to overcome the drawbacks presented by the persistent request mechanism and the main benefits that could be derived from its application. This section is not strictly necessary for understanding the prior-

ity request mechanism. Thus, the reader may go forward to the next section without a conceptual disconnect.

The initial idea behind priority requests was to replace the persistent request mechanism by a more elegant and flexible mechanism able to prevent starvation in a natural way. However, priority requests not only represent a more elegant and flexible approach, but they are also able to reduce both the runtime of the applications and the generated network traffic. A key factor for its success is the fact that all nodes receive priority requests in the same order. This simple but effective feature lets priority requests overcome all the shortcomings that prevent persistent requests from efficiently handling starvation situations. In what follows, we analyze in detail how the different drawbacks exhibited by persistent requests are overcome.

There exist two different approaches to implement persistent requests: arbiter-based mechanisms and distributed arbitration mechanisms. The implementations based on arbiters present the serious disadvantage of indirection, which significantly increases the latency of cache misses and the generated network traffic. Due to the fact that one of the main goals of Token Coherence is to avoid indirection, we rule out those approaches based on arbiters, focusing exclusively on the persistent request mechanism based on distributed arbitration, which besides is the most efficient implementation.

The first drawback exhibited by persistent requests is related to their lack of order. Because of that, each component may receive persistent requests in a different order. Thus, if they were served according to their arrival order, protocol races would be generated and they would not prevent starvation. Therefore, the persistent requests have to be served according to a fixed-priority scheme, such as the processor identifier. Although correct, this solution creates load imbalance.

The second drawback of persistent requests is related to the need of deactivation messages. When a persistent request completes, its issuer must generate a message notifying the other components of it, which will proceed to deactivate the completed persistent request. Given that persistent requests are served according to a fixed-priority scheme, temporal race situations could arise while they are being served. This is because a persistent request with a higher priority could arrive just after serving one with lower priority, which

would immediately deactivate the served persistent request and may momentarily prevent its completion. Although the starvation prevention is still ensured, it is clear the inefficiency of this strategy. In addition, in this situation, it is not possible to know if a persistent request has been eventually completed, unless an explicit deactivation message notifying this fact is broadcast to the other nodes.

The third drawback is related to the fact that persistent requests can not be efficiently served by the performance policy component. Specifically, the need of sending deactivation messages every time a persistent request completes influences the way in which persistent requests are served. In particular, taking into account that a starved processor will be the first one at being aware of the completion of its own persistent request, it would be convenient that it was in conditions to serve the next persistent request as soon as the deactivation message is issued, which reduces the latency of the persistent request service. To this end, the issuer of the active persistent request gathers as many tokens as possible. To this end, all processors send as many tokens as they can to the issuer of the active request, although that issuer only needs just one token, increasing network traffic. Notice that this strategy differs a lot from the efficient MOESI policy used by transient requests. Furthermore, this prevents the service of other persistent or transient requests generated during the service of a persistent request because the data and the tokens are traveling toward the starved processor. Thus, the latency of completing persistent requests is usually much higher than the latency of completing transient requests. Additionally, the transient requests received while a persistent request is active can not be served, causing new unnecessary starvation situations.

On the contrary, the priority request mechanism completely removes the referred drawbacks. This is due to the fact that priority requests are delivered in the same order at every component. This fact allows to naturally serve priority requests according to their arrival order without causing protocol races. As protocol races can not occur, the completion of priority requests is ensured, not requiring the use of deactivation messages. Notice, though, that some kind of completion notification may continue being necessary, especially if we consider that priority requests are stored in a table at each component. However, unlike the persistent request mechanism, the completion notification

is not implemented by deactivation messages. Rather, taking advantage of the in-order priority request service, the completion of a certain priority request would implicate, in its turn, the completion of all the priority requests with a higher priority. Therefore, specific completion notifications for each priority request would be no longer strictly necessary, relaxing the procedures by which priority requests become aware of their completion.

Additionally, if deactivation messages are not used, it is no longer necessary to use a performance policy different from that used by transient requests. Therefore, priority requests will be able to be served like transient requests but with a higher priority. This fact has important consequences. First, the priority read requests can be marked as completed as soon as they are served (in the responder node). In the case of a priority write request, marking it as completed will be possible only if the responder is able to provide the data and all the tokens (*Modified* state). Notice that marking a priority request as completed after being served means that completion notifications will be no longer necessary. Second, the service rate of priority requests will noticeably increase because a new priority request can be served as soon as the previous one is marked as completed. Furthermore, the transient request service can be resumed as soon as all the priority requests upon the same memory block are marked as completed. Indeed, priority requests are served like transient requests, but with a higher priority. This fact, as we will observe, may have a significant impact on performance.

Since deactivation messages are removed, an alternative mechanism will be necessary to notify the priority request completion to the other nodes. In particular, we have observed that it is possible to use the data responses themselves to this end. Given that the reception of a data response depends on the previous existence of some request, it may be thought that this notification mechanism could delay the service of the subsequent priority requests. However, the priority request service can not be delayed for this reason at all. Indeed, what really delays the request service is the absence of data/tokens. Notice that this will occur only after serving a priority write request. Taking into account that when a priority write request is served all the tokens available in the node are sent to the writer, then, the next priority request will not be able to be served until the node gets again the data and at least

one token. Notice that they will be obtained only on the arrival of the first response message with data referred to that data block. However, it can be guaranteed that the completion notification required to resume the priority request service will arrive at the latest with the arrival of the first response message containing the data and the tokens required to serve the next priority request. Therefore, the applied mechanism for completion notification does not delay the priority request service, that is, it is not placed on the critical path.

Moreover, it is not possible that a node serves a priority request that was completed by other nodes as a consequence of not having received yet the corresponding completion notification. In particular, in the case of a priority read request, the node holding the owner token is the first node that becomes aware of its completion, whereas the rest of nodes are not in charge of serving that request, according to the performance policy. Therefore, in a strict sense, they would not require to be notified. In the case of a priority write request, its completion requires the requester node collects all the tokens, thus preventing the rest of nodes from serving any priority request corresponding to the same data block. In both cases, the rest of the nodes will be informed of it by completion notifications embedded into the response messages with data sent by the owner node[3]. When a node receives a response message with data, it will mark as completed the priority requests that were completed as indicated by the response message. When the owner node varies, the new owner node inherits from its predecessors by means of the coherence messages the updated information about the priority request completion.

5.12 Priority Request Summary

This chapter has introduced priority requests as one approach for preventing starvation in Token Coherence. Due to the large design space for implementing priority requests, we selected a single proof-of-concept design point for use in our quantitative evaluations, detailed following.

[3]Notice that after completing a priority write request the requester node becomes the new owner.

5.13 Evaluation

We evaluate the performance of the two presented approaches: priority requests using a single ordered path and several ordered paths (Section 5.10).

5.13.1 Target System and Parameters

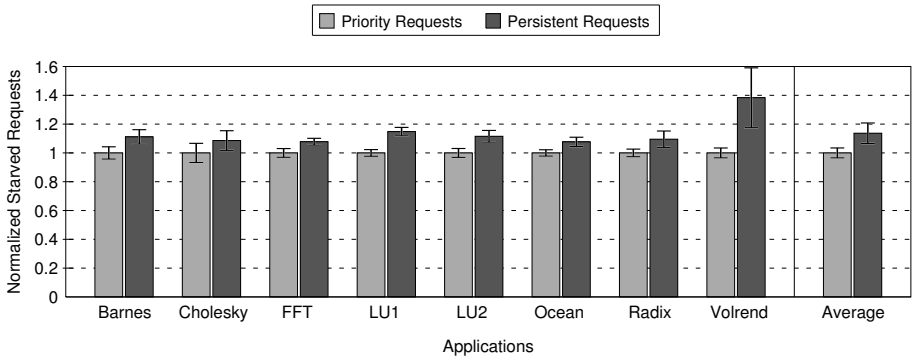
The target systems for evaluation consist of those systems previously described in Section 4.2. As commented, the referred systems are evaluated using two different interconnects: an MIN with a perfect-shuffle topology and a 2D mesh. In the MIN interconnect, when using a single ordered path, priority requests are always routed through the first switch of the last stage (root switch). To perform a fair comparison, given that the implementation of persistent requests that we assume requires point-to-point ordering, they are always transmitted through the first switch of the last stage. When priority requests use several ordered paths, to perform a fair comparison, the persistent requests use disjoint paths. In the 2D mesh, the comparison is similar. When using a single ordered path, the root switch is one of the switches not located in the edge of the mesh (if possible).

To code the *completed PR* field in response messages, we assume the third optimization explained in Section 5.7, according to which only the responses upon memory blocks requested by a valid priority request will include the *completed PR* field. We select this optimization because it reduces the overhead of *completed PR* field without generating unnecessary data-less responses.

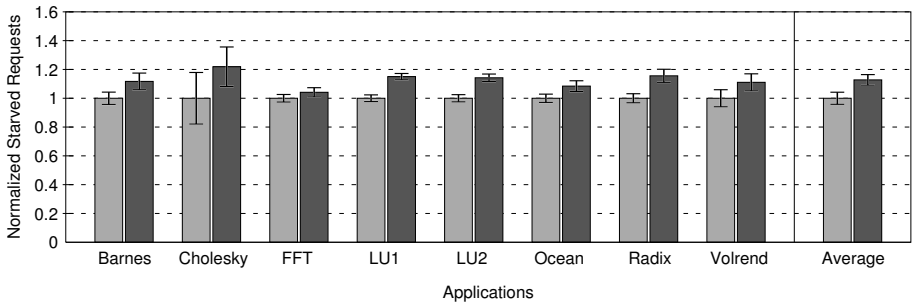
All the data shown in the figures of this section are normalized to those data obtained while using the priority request mechanism.

5.13.2 Starved Requests

Figure 5.12 shows the normalized number of requests that suffer from starvation in a 32-processor system. As shown, when we use the priority request mechanism to solve starvation, the number of starved requests decreases in more than 10% in average. This reduction is due to two main factors. First, while a persistent request is active, transient requests can not be served. Therefore, all the transient requests that are generated while there exists at least one active persistent request will suffer from starvation and they will have to



(a) MIN interconnect

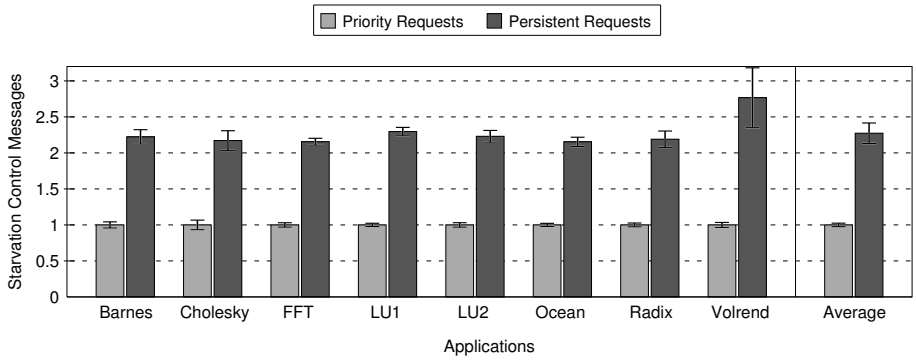


(b) Mesh interconnect

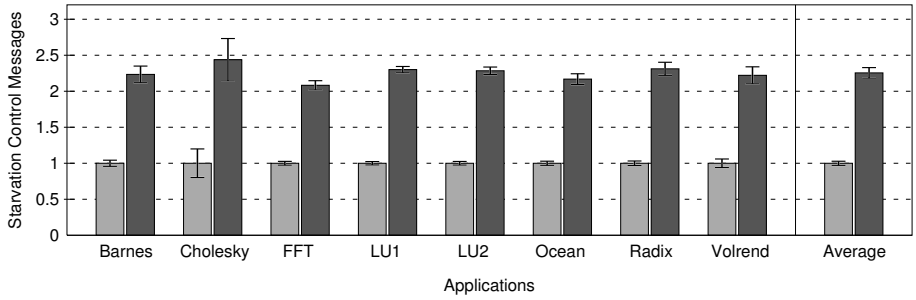
Figure 5.12: Normalized number of starved requests in a 32-processor system.

be served by the persistent request mechanism. However, when the priority request mechanism is assumed, transient requests can be served regardless of the existence of priority requests. That is, priority requests do not block the service of transient requests. Thus, the transient requests generated during a starvation situation may be served, thereby avoiding generating new starvation situations.

The second reason of the reduction in the number of starved requests is because, when the priority request mechanism is assumed, the performance policy is used at all times. Therefore, tokens are handled in an efficient way without taking into account the presence of priority requests. On the other hand, persistent requests override the performance policy and they force the nodes to send their tokens to the issuer of the active persistent request (although it does not really need those tokens). Therefore, some nodes forward



(a) MIN interconnect



(b) Mesh interconnect

Figure 5.13: Normalized number of starvation control messages in a 32-processor system.

tokens they will need, which will cause new cache misses and the sending of new transient requests, which are likely to fail due to the presence of persistent requests.

5.13.3 Starvation Control

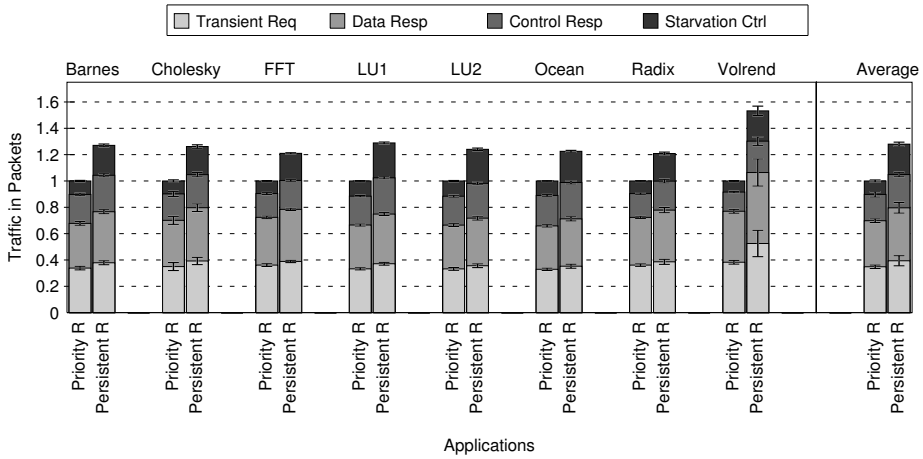
Figure 5.13 illustrates the normalized number of messages used to control the starvation situations happened during the execution of the applications. We use the term *starvation control* because this kind of message not contains data nor tokens. Thus, when using the persistent request mechanism, the starvation control messages are made up of activation messages and deactivation messages. On the contrary, when the priority request mechanism is used, the starvation control messages consist of only the priority requests. As shown in

the figures, the priority request mechanism considerably reduces the number of control messages, using less than half the messages used by the persistent request mechanism. This reduction is due to two reasons. First, the persistent request mechanism requires two messages per each starved request (one activation and one deactivation), while the priority request mechanism only requires one message per starved request (one priority request). Besides, as we showed in the previous figure, the priority request mechanism reduces the number of generated starved requests, which in turn helps to reduce the number of generated starvation control traffic. This reduction is very important since the starvation control messages used by both mechanisms are broadcast messages and their use entails to flood the interconnect.

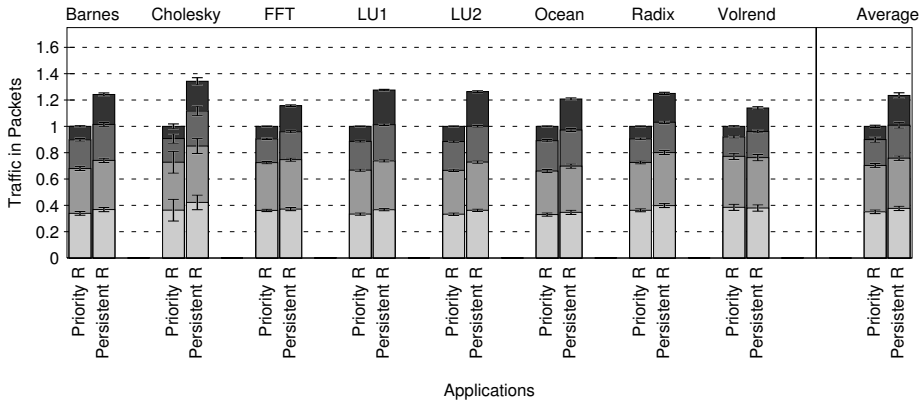
5.13.4 Network Traffic

Figure 5.14 depicts the normalized network traffic generated during the execution of the applications. The traffic of the Token Coherence protocol is made up of transient requests (which are sent when a cache miss occurs), data responses (which are response messages that contain data), control responses (which are data-less responses), and the previously described starvation control messages. While the transient requests and the starvation control messages are broadcast messages, the data responses and control responses are point-to-point messages.

As shown in the figure, the total traffic generated by the system reduces about 25% in average when we use the priority request mechanism. This is because, unlike persistent requests, priority requests do not override the performance policy. As a result, an efficient policy is being used at any time and, therefore, tokens and memory block are efficiently handled. Consequently, the number of cache misses decreases as it is reflected in the number of transient requests, which slightly decreases. Previously we commented that the number of starved requests decreases too. Therefore, the reduction in the transient requests and the priority requests causes a reduction in the number of both data responses and control responses (as the number of requests decreases, less responses will have to be sent to serve them). Finally, the number of starvation control messages lowers because of the reasons previously cited. Hence, the priority request mechanism gets a reduction of the total generated traffic.



(a) MIN interconnect

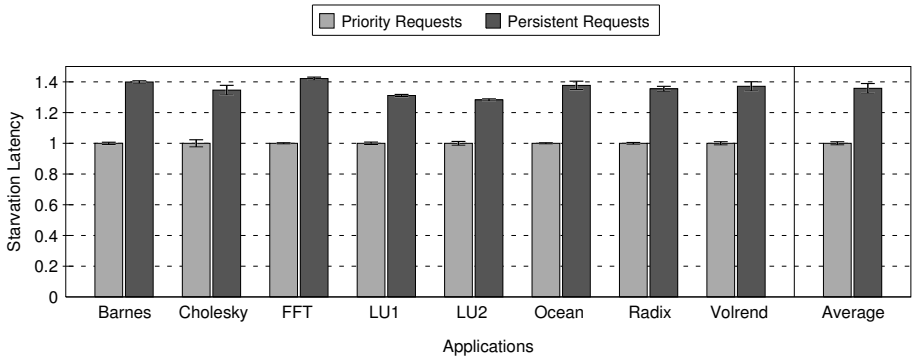


(b) Mesh interconnect

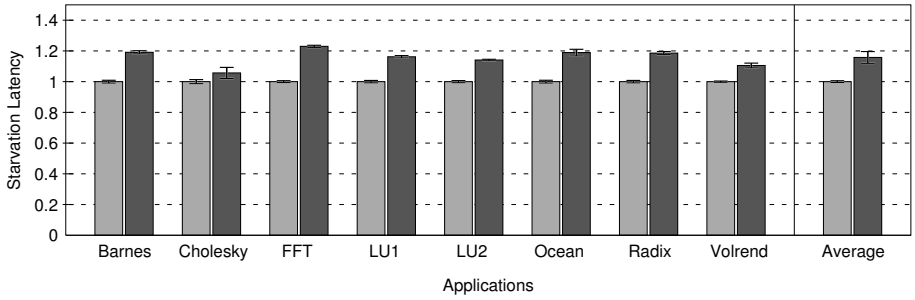
Figure 5.14: Normalized network traffic in packets in a 32-processor system.

5.13.5 Starvation Latency

Figure 5.15 shows the average latency of completing a starved request. It includes the elapsed time from a starved request is detected up to the starved processor receives all the requested tokens. According to the figure, priority requests reduce the starvation latency about 40% in average in the MIN and about 20% in average in the mesh. This reduction is due to two factors. First, since priority requests reduce network traffic, messages suffer less contention and their latency reduces. As a result, both requests and responses arrive



(a) MIN interconnect

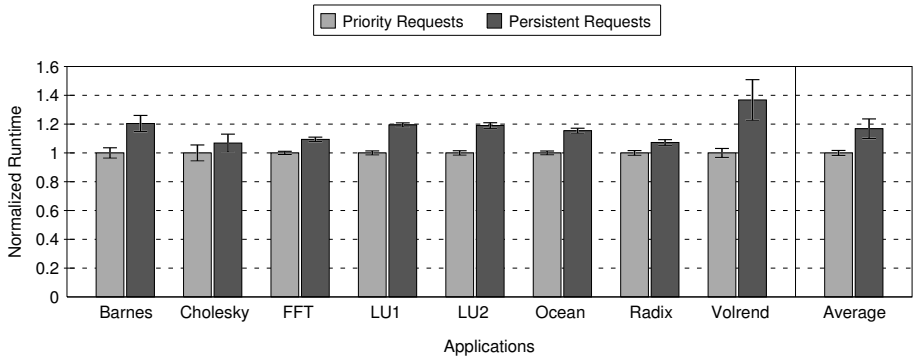


(b) Mesh interconnect

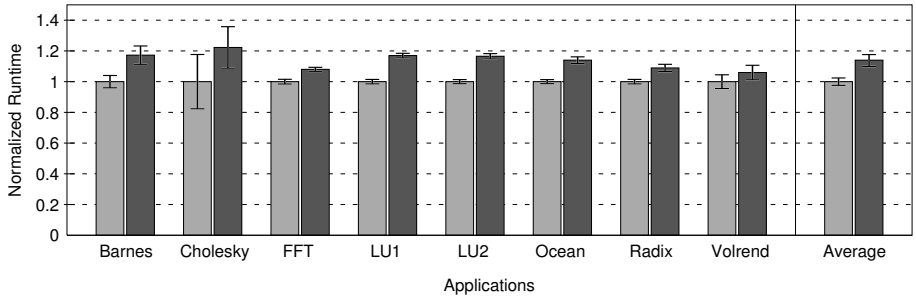
Figure 5.15: Normalized starvation latency in a 32-processor system.

sooner at their corresponding destinations, which reduces the latency of completing a request. In addition, priority requests do not need to use explicit acknowledgments, which can accelerate the service of the starved requests. While a persistent request can not be served until both the active persistent request is completed and an acknowledgment (or deactivation message) is received, priority requests can be served consecutively, without the need of waiting for acknowledgments. If a component holds some tokens required by a priority request, the tokens will be immediately sent to its issuer. The component will never delay their sending waiting for an acknowledgment. Thus, this lack of explicit acknowledgments accelerate the service and, therefore, the completion of starved requests.

Note that priority requests get a higher reduction of the starvation latency in MINs than in meshes. This is due to the fact that a mesh in a 32-processor



(a) MIN interconnect



(b) Mesh interconnect

Figure 5.16: Normalized runtime in a 32-processor system.

system is not a square mesh. As a result, this implementation lacks some of the properties provided by square meshes and the priority requests can not take advantage of all their properties, which causes the reduction to be significantly smaller.

5.13.6 Runtime

Figure 5.16 illustrates the normalized runtime of the applications. As shown, the priority request mechanism contributes to reduce the runtime of the applications about 20% in an MIN and 15% in a mesh (in average). This reduction is the result of combining all the previously commented advantages: efficient policy every time, reduction of network traffic, and reduction of the latency to complete requests. In this case, the reduction of the runtime in a mesh is slightly lower because the reduction of the starvation latency is also lower.

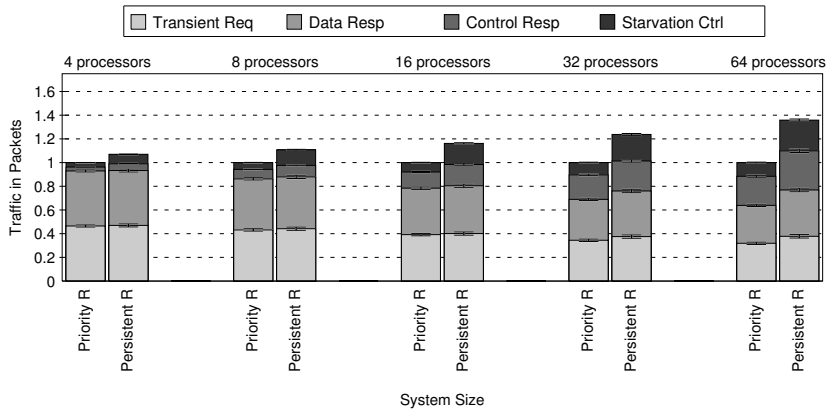


Figure 5.17: Normalized network traffic depending on the system size.

5.13.7 Scalability

Up to now, we have shown that the priority request mechanism performs better than the persistent request mechanism in a 32-processor system. In this section we want to show that priority requests scale better than persistent requests. Thus, following we analyze the scalability by focusing on three parameters: the total interconnect traffic, the starvation latency, and the runtime. In this case, we have simulated 5 different systems using an MIN interconnect: a 4-processor, 8-processor, 16-processor, 32-processor, and 64-processor systems. The figures shown in this section only include the results in average for the simulated applications instead of the results for each individual application to reduce the number of figures and for the sake of clarity. Note that, unlike in the previous results, we have only taken into account 4 applications to obtain the average results: Barnes, Cholesky, FFT, and LU1. We have only selected these four applications because the simulation of a 64-processor system is extremely slow and it is not feasible the simulation of some of the most complex applications due to their time requirements. Thus, although the data of the following figures may not be as complete as those in previous figures, we think they are enough to provide an idea of the scalability of the priority request mechanism.

Figure 5.17 depicts the normalized total traffic in packets depending on the system size. As shown, the reduction in the interconnect traffic grows

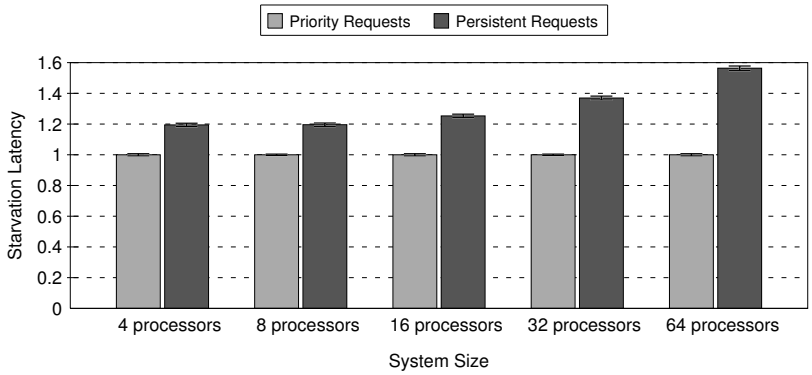


Figure 5.18: Normalized starvation latency depending on the system size.

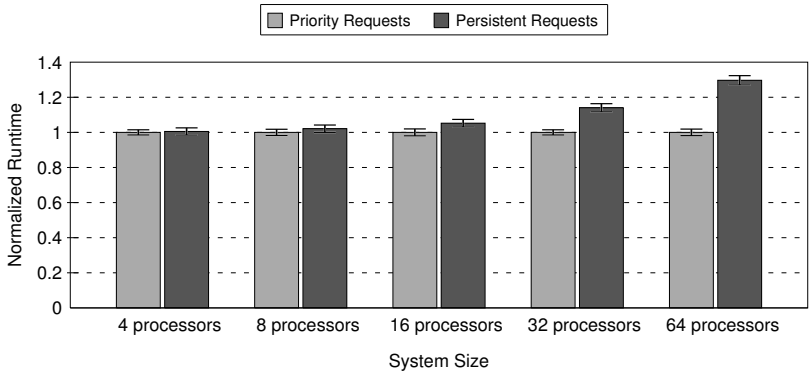


Figure 5.19: Normalized runtime depending on the system size.

as the number of processors increases, reaching about 35% of reduction in a 64-processor system.

Figure 5.18 shows the normalized starvation latency depending on the system size. The reduction in a 4-processor and in an 8-processor system is approximately the same, reducing about a factor of 0,2. In a 16-processor system, the reduction slightly increases (0,25). As the number of processors increases, the reduction in the starvation latency becomes more significant, reaching 0,37 in a 32-processor system and 0,56 in a 64-processor system.

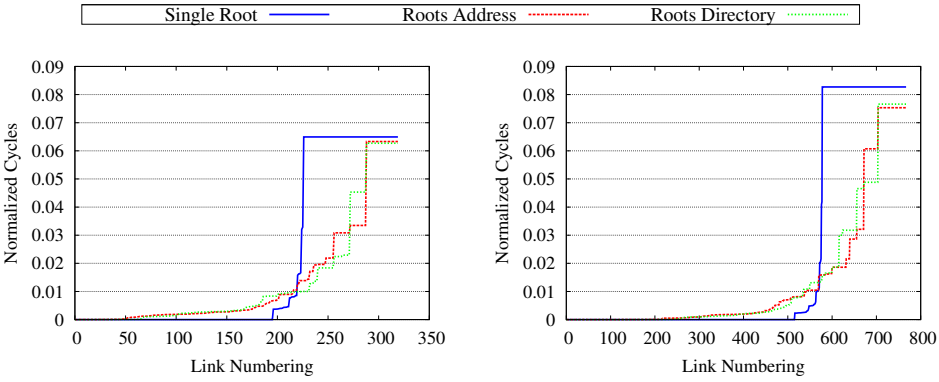
Finally, Figure 5.19 illustrates how the runtime changes depending on the system size. Thus, in a 4-processor and in an 8-processor systems, the runtime of the applications is more or less the same independently of the used starvation prevention mechanism. This is because in those systems the number of

protocol races is not significant and, therefore, the starvation prevention mechanism has little impact on the global performance. However, as the number of processors increases, the contention between processors becomes more significant and the use of the starvation prevention mechanism is more frequent. Hence, in a 16-processor system we can observe as the use of the priority request mechanism slightly improves in a factor of 0,05 the performance of the Token Coherence protocol. For medium systems, the performance of the starvation prevention mechanism is a substantial part of the overall performance. Consequently, the priority request mechanism improves the performance of Token Coherence in a factor of 0,15 in a 32-processor system and in a factor of 0,30 in a 64-processor system. Therefore, we can see that, besides performing better, the performance is increased in a higher and higher factor, which gives an idea of the scalability of the proposed mechanism.

5.13.8 Several Ordered Paths

Although a single ordered path is a good solution for small systems, in medium and large systems the root switch may become a bottleneck. To solve this problem and improve the scalability of priority requests, several ordered paths can be used as proposed in 5.10. In the results shown up to now, we have assumed a single ordered path for all the priority requests. In this section we show how the performance of priority requests scales better when several ordered paths are used. To this end, we have implemented two different options. In the first option labeled as *Roots Address* in the charts, a different root switch is selected according to the address of the requested memory block. In the second option, labeled as *Roots Home* the root switch is selected according to the home memory module of the requested memory block. Note that, since we simulate a system with a MIN interconnect, in an X-processor system there will be $X/2$ ordered paths, that is, as many ordered paths as number of switches in the last stage.

Figure 5.20 illustrates the link utilization due to priority requests when using a single ordered path (*Single Root*) or when using several ordered paths (*Roots Address* and *Roots Directory*). The links are ordered according to their use. As shown in the figure, when using a single ordered path, we can classify the use of the links in three types. First, there are a lot of links that are not



(a) 32-processor system (b) 64-processor system

Figure 5.20: Link utilization due to priority requests.

used and, therefore, their utilization is 0. Second, there are a few links that their use is not 0 but they are not very used. These links correspond to the uplinks which make up the path from system components to the root switch. Finally, the third set consists of the downlinks (links from the switch root to the recipients) which have the maximum utilization. Thus, when using a single ordered path, links are mainly either not used at all or continually used. This situation changes when using several ordered paths. In this case, there are few links which are almost not used and the use of the links increases little by little. In the figure, we can observe some steps. These are due to links belonging to switches from the same stage. Switches from lower stages are more used than switches from upper stages. We can also see in the figure that the use of links is more spread and that the number of switches with the maximum utilization decreases. Beside, the maximum utilization decreases in comparison to the approximation with a single ordered path.

Figure 5.21 shows the normalized runtime of the applications according to the system size when using a single ordered path (*Single Root*) or when using several of them. As you can observe, for small systems, the overall performance of the protocol is not affected by the use of a single ordered path or several ordered paths. Thus, in 4-processor, 8-processor, and 16-processor systems, the runtime of the applications is more or less constant. However, when the number of processors increases and, therefore, the number

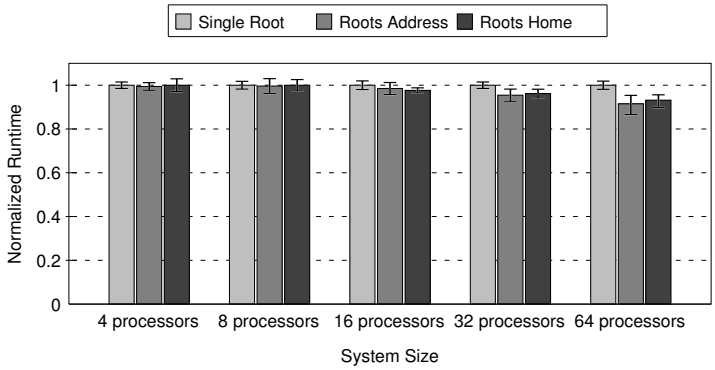


Figure 5.21: Normalized runtime depending on the system size when using several ordered paths.

of processors involved in a starvation situation increases, a single ordered path may become a bottleneck. This is solved by using several ordered paths. Hence, in a 32-processor system we can see that the average runtime improves about 7%. In a 64-processor system, the runtime reduction doubles.

5.14 Conclusions

In this chapter we have proposed an efficient starvation prevention mechanism. Unlike previous proposals, priority requests rely on a total order provided by the routing algorithm to solve protocol races. This simplifies the starvation prevention process a lot and provides a major advantage: the efficient performance policy can be applied every time, regardless of the appearance of protocol races.

The results shown in this chapter show that priority requests substantially reduce the interconnect traffic and the time required to resolve the generated starvation situations, which in turn contributes to reduce the runtime of applications when it is threatened by the occurrence of protocol races. Besides, we have also shown that the performance of priority requests scales much better than that of persistent requests and this is a key factor in current and future systems due to the trend to increase the number of processors and cores.

Chapter 6

Bounding Storage Requirements

Starvation prevention mechanisms require that nodes maintain a list of the starved requests in tables. The size of these tables is proportional to both the system size and the maximum number of simultaneous starved requests allowed per processor. Therefore, the tables are not scalable, entailing a serious problem in medium and large-sized systems. In this chapter we propose an effective strategy to limit the table size at the expense of a slight performance degradation.

6.1 Introduction

The starvation prevention mechanisms based on centralized/banked arbiters are too inefficient since the communications between processors are performed through the arbiters, which causes indirection and increases the latency of cache misses. On the other hand, the mechanisms based on distributed arbiters or priority requests perform much better thanks to a direct communication between processors. According to those approximations, the outstanding starved requests must be remembered in the system components themselves (instead of in the arbiters). To this end, each component maintains a table where the outstanding requests are stored at their arrival. In order to be able to store all the outstanding starved requests, the size of the tables must be

proportional to both the number of processors in the system and the number of simultaneous outstanding requests allowed per processor. Although correct, this approach is not suitable for medium and large systems because, as the system size increases, the space that the nodes must book to store all the outstanding requests will be larger and larger. As result, those proposals lack scalability and they are limited just to systems with a moderate number of processors and only few simultaneous outstanding requests per processor. Besides, since the tables require associative search, they become slower as the system size grows, which in turn slows down the access to the tables and the latency of cache misses.

In this chapter, we propose an effective strategy to limit the storage requirements of the tables used by the starvation prevention mechanisms applied by Token Coherence. The proposed strategy does not replace the applied starvation prevention mechanisms. Rather it is designed as a complementary element. In the description, we link the proposed strategy to the priority request mechanism because, due to its characteristic features (flexibility, ordering, efficiency), it complements perfectly to the strategy. In fact, when we use the priority request mechanism together with the proposed strategy, in spite of being able to reduce the table size to a minimum of one entry, the priority request mechanism only suffers a slight performance degradation, continuing to perform significantly better than the persistent request mechanism with non-scalable tables. However, the applicability of this strategy is not restricted to priority requests, since it could readily adapted to persistent requests. Nevertheless, we do not describe such an implementation option in this work and its development is intended to be future work. Anyway, we think that the application of the proposed strategy to the persistent request mechanism would significantly increase the complexity of the protocol and it would suffer a greater performance degradation due to the inflexibility and inefficiency of persistent requests and the great amount of control traffic that it requires.

By making the table size independent of the systems size, it may occur that an outstanding priority request can not be stored at its arrival because the tables are full of priority requests that have not been completed yet and, therefore, none of them can be replaced. In such a situation, the received

priority request will be rejected (not stored) and its issuer will have to wait for the completion of one of the stored priority requests. When such a completion occurs, it will be informed and, in that moment, it will resend its priority request. Unlike the first sending of a priority request, the proposed strategy guarantees that the resent priority requests will always be accepted (stored). Thus, although in the worst case a priority request may have to be sent twice, its storage and, therefore, its completion is still guaranteed. Following, the whole strategy is described in detail.

6.2 Data Structures

As commented, each system component holds a table where the outstanding priority requests are stored. However, as the tables that we assume now may not have enough entries to simultaneously store all the received priority requests, some requests will not be rejected at their arrival. To ensure that all priority requests will eventually be stored in the tables, we take advantage of the fact that they are ordered messages. As the stored requests are completed, the rejected ones will be resent and stored according to their original reception order. Thus, if several priority requests are rejected, when a stored one completes, the first rejected request will be the next one at being stored. Making so, all the priority requests are guaranteed to be stored.

To implement that strategy, processors must know the order in which rejected priority requests were received. However, it is not necessary that all processors know the global order of all the rejected priority requests. Rather, the issuer of a priority request only needs to know the issuer of the rejected priority request that will replace its own request when it completes. Therefore, the global order of priority requests can be stored in a distributed way. To determine and store the order, each processor will require two registers. In short, each system component (processors and memories) has a priority request table where the outstanding requests are stored and, besides, each processor has two registers to determine and remember the order of requests. Following, this structures are described in more detail.

- **Priority Request Table.** According to our strategy this table has N entries where N can be lower than the maximum number of simul-

taneous outstanding priority requests in the system. Each entry stores the required information about certain priority request and consists of the following fields: *valid* bit, *issuer*, *address*, *identifier*, *operation*, *state*, and *used* bit. These fields are similar to those of the tables used by the baseline priority request mechanism described in Section 5.3. The single difference is the *used* bit. This bit only makes sense for priority requests marked as completed. It indicates whether the identifier held by that entry can be used (that is, included in the *completed PR* field of an issued priority request) or, on the contrary, the identifier has already been used (we will see it in more detail later). The entry size is 10 bytes. Thus, for example, a system where 64 storable requests are allowed would require only a 640-byte table at each component.

- **Ack Register.** Besides the table, each processor requires an *Ack* register. This register holds the identifier of a processor which sent a priority request that was rejected and which has not been able to be stored yet. A processor can only hold a valid value in its *Ack* register if it has recently issued a priority request. When its own request is stored and completed, it will be the only processor responsible for informing the [*Ack*] processor about the completion.

- **Counter Register.** Its value ranges between 0 and N - 1. This register is used to estimate the corresponding value of the *Ack* register.

While the number of table entries does not depend on the maximum number of simultaneous outstanding priority requests per processor, the number of required *Ack* registers and the number of required *Counters* will depend on it, requiring as many *Ack* and *Counter* registers as the maximum number of simultaneous outstanding priority requests allowed per processor. However, note that, in case of reduced tables, increasing the number of outstanding priority requests would not be as harmful as in case of using tables proportional to the system size. For example, assuming a 64-processor system with tables proportional to the system size, if the number of outstanding requests increases from 1 to 8, the size of each priority request table will increase in a factor of 8. However, assuming a 64-processor system with two-entry tables, if the number of outstanding requests increases from 1 to 8, the size of the

required support structures will only increase in a factor of 1,6. This is because, when tables are proportional to the system size, the size of the required structures will depend on the product of the system size and the number of outstanding requests (num_out_PR):

$$structure\_size = num\_proc * entry\_size * num\_out\_PR$$

On the contrary, with reduced tables, the size of the required structures will only depend on the addition of the table size and the number of simultaneous outstanding requests:

$$structure\_size \cong num\_table\_entries * entry\_size + num\_out\_PR$$

6.3 General Working Scheme

As commented previously, a processor only uses the starvation prevention mechanism when it detects it may be starving. In such a situation, the following steps are carried out:

1. The starved processor composes and broadcasts a priority request message. To set the *completed PR* field, the issuer searches in its table a priority request which (1) was issued by itself, (2) is marked as completed, and (3) is not marked as used. If it finds such a request, its identifier is included in the *completed PR* field and the table entry is marked as used. If the table does not contain such a request (because the entry holding the required information has already been removed), the issuer will set the *completed PR* field to Nill.

2. When a processor receives a priority request message, it checks the *completed PR* field. If it is set to Nill and the table is full, the recipient rejects the priority request and updates its *Ack* and *Counter* registers accordingly (as we see later in more detail). If *completed PR* is different from Nill, the received priority request replaces the stored request appointed by the *completed PR* field.

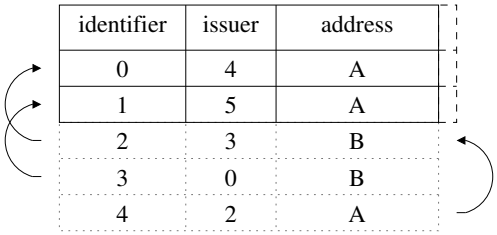


Figure 6.1: Policy followed to ensure the storage of all priority requests. Two table entries are assumed (N = 2).

3. If the storage of a priority request succeeds, the processor will serve it (if required, according to the performance policy), marking it as completed in case of supplying all the requested tokens.

4. Once the issuer of a stored priority request receives all the requested tokens and performs the corresponding memory operation, it marks its priority request as completed. Next, if there exists any priority request that could not be stored (information obtained from its *Ack* register), it will send to the issuer of such a request a resending notification indicating in such a message the identifier of the stored priority request that has just been completed. In this case, the sender marks the stored priority request as used (to prevent the identifier from being included in another request).

5. When a processor whose priority request was rejected receives a resending notification informing about the completion of a stored request, it immediately resends its request, attaching in the *completed PR* field the identifier received in the notification, which will ensure its acceptance.

6.4 Ensuring the Priority Request Storage

Consider an M-processor system and N-entry tables (where $M > N$). If each processor in the system sends a priority request at the same time, only N priority requests will be able to be stored, whereas the remaining M - N requests will not find a free entry and, therefore, will be rejected. In order

to ensure that the rejected priority requests will eventually be stored, the following strategy is carried out. The idea is to replace (when completed) the priority request received in the X^{th} place by the priority request received and rejected in the $(X + N)^{th}$ place (as shown in Figure 6.1). Therefore, when the priority request received and stored in the X^{th} position is completed, its issuer will report about its completion to the issuer of the priority request received and rejected in the $(X + N)^{th}$ position. This report will include the identifier of the completed priority request. Thus, when a processor receives the report, it will resend its priority request, including in the *completed PR* field the received identifier. Since each identifier can only be included once in a *completed PR* field (thanks to the *used* bit), the storage of the resent priority requests is always guaranteed.

6.5 Notifying the Priority Request Completion

The only way to remove a completed priority request from the tables is by using the *completed PR* field of a received priority request. A processor that issues a priority request can get the information required for that field by two different means. The first way is by directly searching in its table. In this case, the issuer searches for a valid priority request issued by itself which is marked as completed, but not marked as used. If it finds such a request, it includes its identifier in the *completed PR* field and marks the table entry as used, thereby preventing that identifier from being included in other requests. Alternatively, the information required by the *completed PR* field can be obtained from a received resending notification too. These notifications are point-to-point messages and they indicate to the receiver that its rejected priority request can be resent with total acceptance guarantee. In addition, these notifications include the identifier of the stored priority request that can be replaced. Therefore, that identifier can be attached in the *completed PR* field.

If the identifier required by the *completed PR* field of the request that is going to be sent is not found in tables, the issuer sets that field to Nill. When the field is set to Nill, the storage of the priority request is not guaranteed and, therefore, neither its completion. Therefore, the priority request will have to

be sent again, but only when its storage can be guaranteed. To this end, the issuer waits for a resending notification, which will inform it about the stored request that has been completed.

In order to generate the resending notifications, each processor which sent a priority request (received in the X^{th} place) must remember (by its *Ack* register) the processor which the notification must be sent to (once its own priority request completes). That processor corresponds to the one whose issued priority request was received in the $(X + N)^{th}$ place. The proper value of the *Ack* register (according to the strategy defined in the previous section) can be estimated easily by means of the *Counter* register. Initially, *Counter* is disabled. While it remains disabled, the arrival of priority requests will not change its state. However, when the a processor receive its own priority request with the *completed PR* field set to *Nill*[1], *Counter* will be initialized to 0, indicating that the count has begun. From this moment, the value of *Counter* is increased in 1 every time a new priority request with the *completed PR* field set to *Nill* is received. When the arrival of a priority request causes the value of *Counter* to overflow the maximum count (*N - 1*), the issuer of such a request will be stored in the *Ack* register and *Counter* will be disabled, indicating the count is over. Making so, *Ack* will point to the processor that issued a priority request (which was rejected) N positions after the priority request issued by the processor where *Ack* is placed.

As soon as a processor completes its priority request and its *Ack* register holds a valid value, the processor sends a notification to the [*Ack*] processor. When the notification is sent, the value of *Ack* is reset and the entry in the table holding the information included in the notification message is marked as used. Marking it as used will prevent such information from being included in a priority request issued locally later. Notice that the first time a processor sends a priority request, it looks in its table for the information required by the *completed PR* field. If it finds such information, it will include it in its priority request (marking it as used). However, if a processor marks one of its priority requests as completed and that information could be included either in a resending notification or in a local priority request, the processor always prioritizes the notification. This prevents the local priority requests

[1]Such a priority request will be able to be stored only if the table is not full yet.

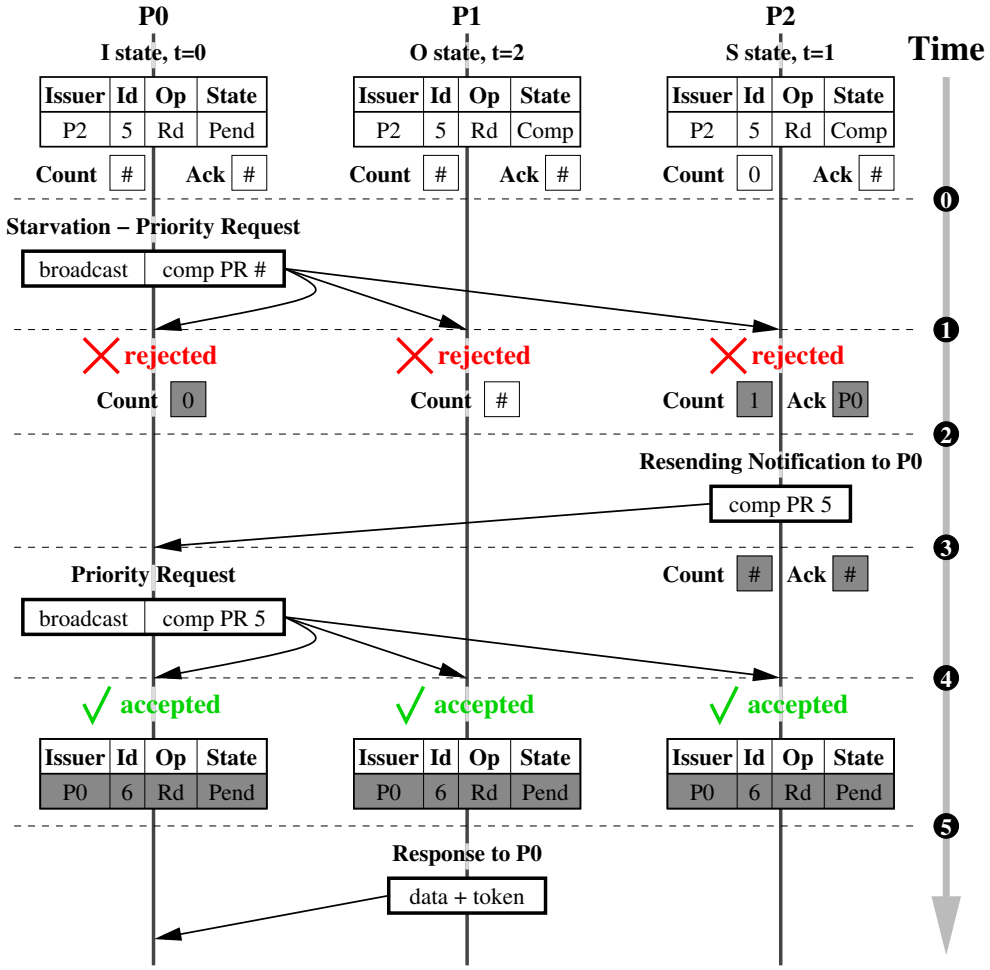


Figure 6.2: Example of a starvation situation. To simplify the example, the table entries only consist of four fields. *Id* stands for the priority level or identifier and *Op* indicates the requested memory operation (*Rd* is Read). *Comp* refers to completed and *Pend* refers to pending. Besides the value # stands for *Nill*.

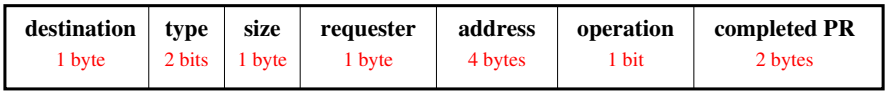
from starving out the rejected requests.

Figure 6.2 walks through an example of how a starvation situation is solved. In this example, there are three processors (*P0*, *P1*, and *P2*) with single-entry tables. Initially, *P0*'s table is full and its *Counter* and *Ack* registers are set to *Nill* (#). *P1* is in Owner state (i.e., it is in charge of providing the requested

memory block), its table is full, and its *Counter* and *Ack* registers are also set to *Nill*. *P2*'s table is full too. However, as the stored priority request belongs to it, its *Counter* register is initialized to *0*. At time 0, *P0* detects it may be starving and broadcasts a priority request. The priority request message does not hold any information about the completed priority request to replace (*completed PR* field is set to *Nill*) because the *P0*'s table does not contain any information about its own priority requests. Therefore, at time 1, the *P0*'s priority request is rejected at its arrival as tables are full. Besides, as *P0* rejects its own message, it initializes its *Counter* to zero. *P1* does not modify its *Counter* because it is disabled. *P2* increases the value of its *Counter* (because it is already enabled) and, as it reaches the maximum count, it stores in its *Ack* register the issuer of the rejected request. At time 2, *P2*'s priority request is completed. Besides, since *P2* holds a valid value in *Ack*, it sends a resending notification message to *P0* notifying it that the stored priority request with id 5 was completed. At time 3, *P0* receives the notification and it immediately proceeds to resend its priority request, attaching in the *completed PR* field the identifier received in the notification message. Thus, when the priority request is received at time 4, all the processors accept it and store it because it can replace a stored priority request (already completed). As *P1* is in Owner state, at time 5, it sends a response message to *P0* and marks that priority request as completed (not shown in the figure). When *P0* receives the response, it will perform the memory operation and will mark its priority request as completed too.

6.6 Reducing the Control Traffic

According to the described strategy, each starved request will require, at worst, three control messages: two priority requests, which are broadcast messages, and one acknowledgment, which is a point-to-point message. Comparing all this starvation control traffic against that generated by the baseline priority request mechanism, we realize that the control traffic will increase significantly. Although in small systems it will not entail a serious problem, in medium/large systems it will. Besides, given that the aim of this strategy is to improve the scalability of the starvation prevention mechanisms, it would be desirable a



(a) normal

(b) size-reduced

Figure 6.3: Format of normal and size-reduced priority request messages.

reduction of the control traffic. To this end, in this section, we propose several implementation options. Each option addresses one particular problem. Besides, they are exclusive, since the implementation of one of those options automatically discards the application of other.

6.6.1 Size of Rejected Priority Requests

Before sending a priority request, the issuer knows whether the request will be accepted or, on the contrary, it will be rejected, thanks to the *completed PR* field. If the field is set to Nill, the request will likely be rejected[2]. Otherwise, it will be accepted. In spite of the fact that processors know when the priority request is going to be rejected, they broadcast a common priority request, which includes certain information (such as the requested memory block) that is not going to be used by any of the recipients. That information unnecessarily increases the length of priority requests and, taking into account that they are broadcast messages, that information is going to flood the interconnect.

In order to reduce the damage caused by the broadcast of such information, in this section we propose an alternative implementation. Thus, when a processor has to broadcast a priority request and it does not find the required information for the *completed PR* field (which indicates that it will probably be rejected), instead of broadcasting a common priority request, a shorter kind of message is used. Given that the only information that the recipients will need to know is the issuer, this new class of message will not need to

[2]A priority request whose *completed PR* field is set to *Nill* will only be accepted if the table is not full. This only happens during the initialization phase.

include the requested memory block or the request type. It will only consist of a source field (processor which issues the request), a type field (message type), and the destination field (broadcast). Figure 6.3 depicts the differences between a common priority request message and the reduced version[3]. As shown, the size-reduced priority request excludes the *address*, *operation*, and *completed PR* fields. Thus, the size of those requests is 4 bytes in contrast to the 10 bytes in the common case, which allows to save 60% of traffic. With this simple solution, the overload that entails the priority requests could be softened.

6.6.2 Removing Acknowledgments

When a processor A sends an acknowledgment to a processor B, A is certain of the acceptance of the priority request that B will resend. Therefore, instead of that the processor A sends an acknowledgment to the processor B and B resends the priority request, the processor A could directly resend the priority request in behalf of the processor B. To do this, the processor A will need to know all the information concerning to the priority request to resend, that is, the issuer processor, the requested memory block, and the priority request type (read or write). To this end, when the rejection of a priority request causes the *Counter* structure to overflow, in the same way the issuer processor is stored in the *Ack* register, processors will also store the requested memory block (in an *Address* register) and the request type (in an *Operation* register). Thus, with only these two additional registers, processors will have all the information that they need to resend a priority request in behalf of another processor. In this way, when a processor completes its own priority request, if *Ack*, *Address*, and *Operation* registers have a valid value, it will resend a priority request in behalf of the [*Ack*] processor. This priority request will include the information (identifier[4]) about the priority request that has just been completed. The main advantage of this implementation option is the elimination of the acknowledgments, which reduces the control traffic and alleviates the interconnection network. Besides, as processors do not need to

[3]The size of priority requests, responses messages, or any other type of message is in accordance to that assumed in [76].

[4]Note that this identifier was that included in the acknowledgment messages.

wait for acknowledgments, the waiting time for resending the priority requests decreases, which will improve the average latency of serving starved requests.

6.6.3 Handling Rejected Priority Requests as Transient Requests

Although the reduction in the size of some priority request messages and the removal of acknowledgments tackle negatives aspects of the technique proposed in this chapter, its worst aspect continues to be the large amount of broadcast messages that have to use. As commented before, if the storage of a priority request does not succeed, the priority request will not be served (although the processor that rejects it holds sufficient tokens to do it). Therefore, the majority of the starved requests will require two priority requests to solve the starvation situation. Though this is correct, this strategy is quite inefficient as priority requests are broadcast messages and its use entails to flood the network.

To address this problem, in this section we propose a strategy that takes advantage of the rejected priority requests. The priority requests that are rejected could be considered as if they were transient requests (since they are not stored in tables). Thus, if a processor rejects a priority request but it holds all the requested tokens, then it will proceed to serve it. Hence, a starved request may be completed by a single priority request without having to resend it and wait for its storage. Note that, although only one priority request is used, its issuer will continue to receive the corresponding resending notification because the processor in charge of sending such a notification will not be aware of the priority request completion. However, when the resending notification is received and the starved request has already been completed, the processor will not resend its priority request. It simply ignores it. Thus, this simple strategy lets the interconnection network traffic be alleviated due to the reduction of broadcast messages. Besides, the average latency of resolving starved requests will decrease (as responses may be received sooner), which in turn will contribute to improve the overall performance.

As commented, according to this implementation option, it may occur that when a processor receives a resending notification it has already completed its priority request and, since it is completed, it will not need to resend

it. Therefore, the identifier of the completed request received in the resending notification will not be broadcast and, if it is not stored, it will be useless. Notice that the processor that sent the resending notification marked such a request as used and it will not be able to include that information in a priority request. Therefore, if the receiver of a resending notification does not remember nor broadcast that information, the completed priority request will not be able to be removed from the tables and a failure will be caused. To prevent this situation from happing, the recipient of a resending notification will have to keep the information received in the resending notification when that information is not immediately broadcast. To this end, a dedicated register can be used. However, this would increase the storage requirements at each node. Therefore, an alternative solution is to store the received information in the own table. Thus, the processor that receives a resending notification and does not immediately broadcast the received identifier searches in its table for the entry holding the request with the received identifier and it marks that entry as completed. Besides, the *issuer* field of that entry is set to the local processor. Making so, when the processor has to use that information (because of a local priority request or a resending notification) and it searches in the table, it will be able to find the required information.

6.7 Evaluation

In what follows, we compare Token Coherence using the persistent request mechanism (*Persistent*) against Token Coherence using the priority request mechanism with table size proportional to the system size, that is, with 32-entry tables (*Priority R*), and using 16-entry tables (*Priority-16*), 8-entry tables (*Priority-8*), 4-entry tables (*Priority-4*), 2-entry tables (*Priority-2*), and 1-entry tables (*Priority-1*).

6.7.1 Target System and Parameters

We simulate a 32-processor Sparc v9 system with the parameters commented in Chapter 4. We evaluate the referred processor system assuming that processors are connected through an MIN network with the perfect-shuffle permutation. The results for the 2D mesh are not shown because they are similar

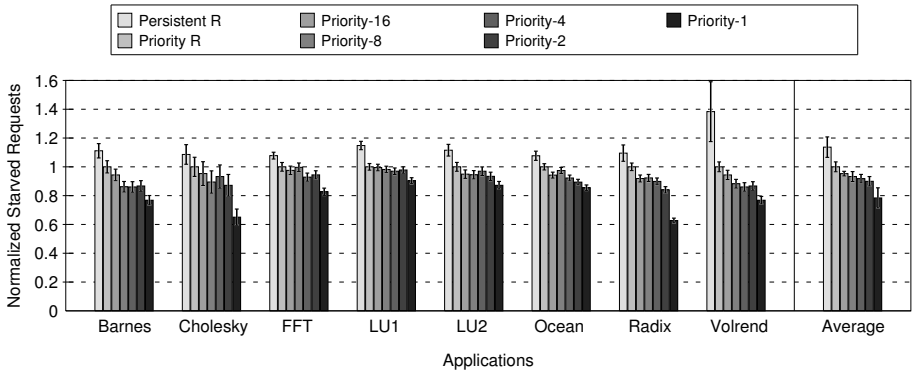


Figure 6.4: Normalized starved requests.

to those obtained for the MIN.

The simulated starvation prevention mechanisms assume only one simultaneous outstanding persistent/priority request per processor. Given that we aim to improve the scalability, the optimization described in 6.6.3 is assumed since it reduces the number of broadcast messages, which are a threaten to the protocol scalability. The assumption of this optimization automatically discards the other ones.

All the data shown in the figures of this section are normalized to those data obtained while using the priority request mechanism.

6.7.2 Starved Requests

Figure 6.4 illustrates the normalized number of requests suffering starvation. As shown, when the size of the priority request tables lowers, the number of starved requests diminishes, reaching 25% of reduction in average in case of single-entry tables. This is due to two main reasons. First, as the tables are reduced, the average latency of completing starved requests increases (as we see next), which, in turn, automatically increases the timer used to detect starvation, causing only the request that actually suffer starvation to be served by the starvation prevention mechanism. Second, since the service of the starved requests is slower, tokens remain longer in caches, avoiding forwarding them too soon and, therefore, preventing new starvation situations from being generated.

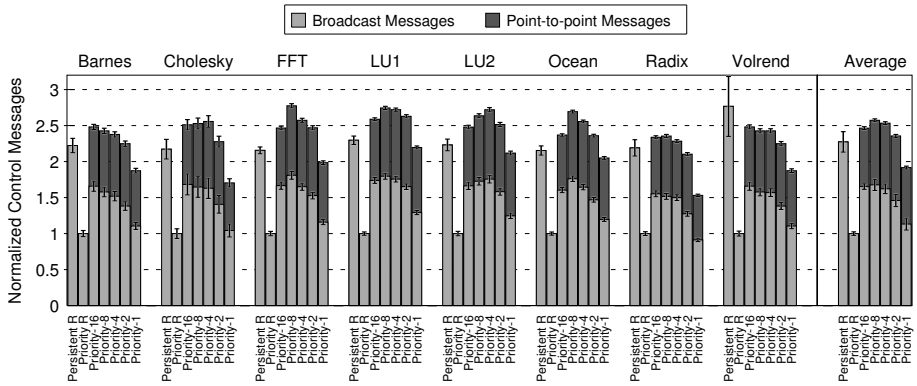


Figure 6.5: Normalized starvation control messages.

6.7.3 Starvation Control

Figure 6.5 shows the normalized number of control messages used to manage the generated starvation situations. These messages include priority requests in *Priority R* and priority requests and resending notifications when tables with reduced size are used. As depicted, for *Priority-16*, *Priority-8*, *Priority-4*, and *Priority-2*, despite the fact that the number of generated starved requests is smaller than that generated by *Priority R*, the total number of control messages is higher. This happens because each starved request will require, at worst, three control messages (two priority requests and one resending notification), which increases the total number of control messages between 10% and 20% in average with respect to *Priority R*. In case of *Priority-1*, as the number of starved requests is so low, the total number of control messages is 5% smaller than that generated by *Priority R*. Besides, in this last case, the number of broadcast messages is 45% lower than that generated by *Persistent R*.

6.7.4 Network Traffic

The normalized total traffic (in packets) generated by Token Coherence is depicted in Figure 6.6. *Control Response* stands for data-less response messages and *Starvation Control* stands for persistent/priority requests. As shown, *Priority R* slightly reduces the number of *Transient Request* messages due to the

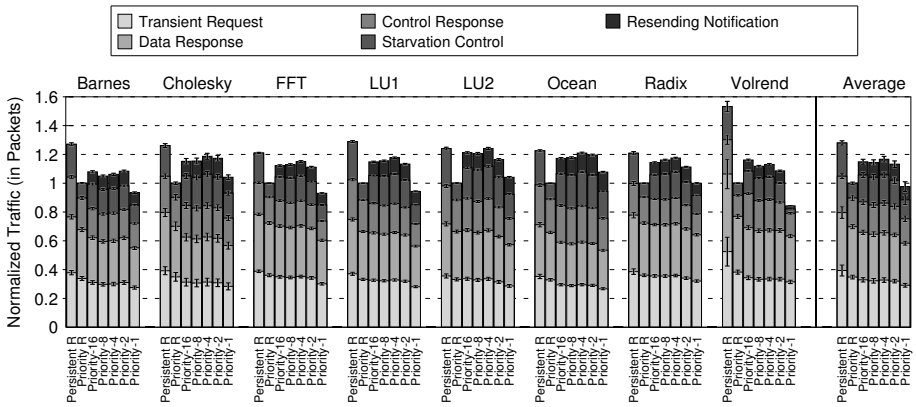


Figure 6.6: Normalized network traffic.

fact that less cache misses are generated because of the use of an efficient performance policy all the time. In turn, the reduction of cache misses causes the number of *Data Response* and *Control Response* messages to lower. Besides, as we commented previously, *Priority R* requires less *Starvation Control* messages than *Persistent R*. Thus, Token Coherence using *Priority R* reduces about 20% the total traffic generated with respect to *Persistent R* mainly because *Priority R* efficiently manages tokens and memory blocks (as efficiently as in absence of races). For *Priority-16*, *Priority-8*, *Priority-4*, and *Priority-2*, although the overall traffic is still smaller than that generated when using *Persistent R*, it slightly increases with respect to that generated by *Priority R* mainly because of the increase in the *Starvation Control* and *Resending Notification* messages. For *Priority-1*, thanks to the great reduction of starved requests, the overall traffic is more or less similar to that generated by *Priority R*.

6.7.5 Starvation Latency

Figure 6.7 shows the normalized average latency of completing starved requests. It includes the elapsed time from a starved request is detected up to the service of that request is completed. According to Figure 6.7, *Priority R* reduces about 25 % the average latency of completing a starved request with respect to *Persistent R*. This reduction is due to the fact that, unlike *Per-*

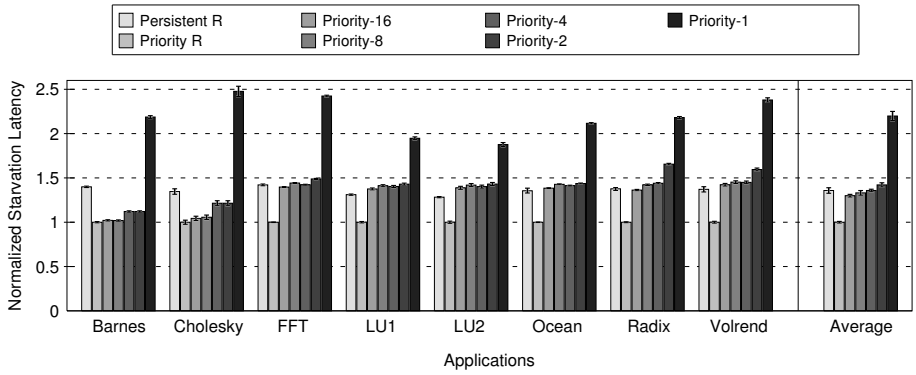


Figure 6.7: Normalized starvation latency.

sistent R, *Priority R* serves the starved requests, without having to wait any acknowledgment (or deactivation message). When using *Priority-16*, *Priority-8*, *Priority-4*, or *Priority-2* the average latency increases slightly, being more or less similar to that of *Persistent R*, as the priority requests may require to be sent twice, having to wait for a point-to-point acknowledgment (the resending notification). Finally, when using single-entry tables (*Priority-1*), the average latency increases by a factor of about 1.5. This increase is due to the fact that some starved requests may be served one by one (only one priority request can be stored in the tables). Despite that, the latency is not extremely large because some priority requests can be served without being stored (according to the optimization proposed in Section 6.6.3). This is possible thanks to the fact that priority requests are ordered messages, which usually suffices to solve most of the protocol races.

6.7.6 Runtime

Finally, Figure 6.8 illustrates the normalized runtime of the applications. As shown, *Priority R* reduces the runtime between 8 % and 20 % (more than 10 % in average). In spite of reducing the table size, the runtime of the applications when using our approach is only slightly higher than that of *Priority R* as the increase in both the overall traffic and the average latency of starvation is offset by the reduction of the number of starved requests. Thus, despite reducing the table to a single entry, the runtime is even lower than that of the most

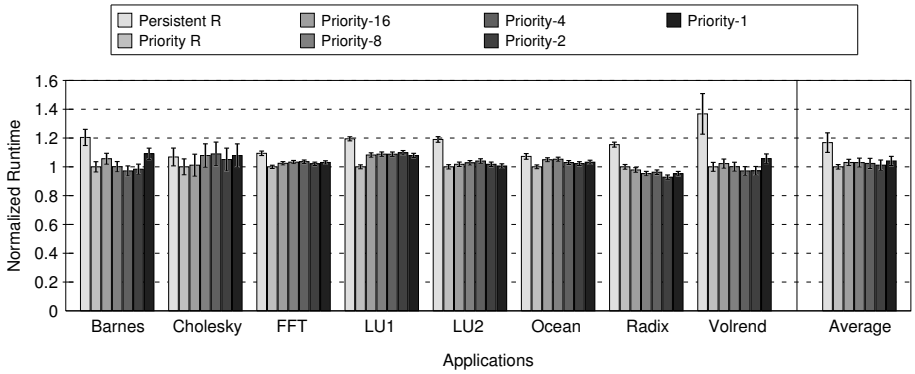


Figure 6.8: Normalized runtime.

efficient implementation of the persistent request mechanism (*Persistent R*). This is possible thanks to the considerable decrease of the starvation situations and the use of an efficient performance policy all the time, which affects all the processors in the system (independently of whether they are involved in a race or they are not).

6.8 Conclusions

In this chapter we address the scalability problem related to the storage requirements in Token Coherence. In particular, we have proposed an effective strategy that improves the scalability of Token Coherence by limiting the storage requirements of the starvation prevention mechanisms proposed until now. This is achieved at the expense of a slight degradation of the overall performance as long as the required table size is reduced. However, despite the fact that the table size can be reduced until a minimum of one entry, the execution time of the analyzed applications is never higher than that of the most efficient implementation of the persistent request mechanism. Thus, our proposal improves the scalability of the Token Coherence protocol by decoupling the storage requirements from the system size, while still maintaining the overall system performance.

Chapter 7

Switch-based Packing Technique

Starvation prevention mechanisms are based on broadcast messages, which are neither scalable nor suitable for medium and large systems. Besides, as we have already commented, token-based protocols use non-silent invalidations, which increases the protocol latency. To address both problems, in this chapter we propose a switch-based technique which allows to pack several messages into just one. By doing this, we can considerably reduce the traffic related to (1) the broadcast messages used by the starvation prevention mechanism and (2) the number of non-silent invalidations. Other packing techniques were previously proposed [90], however they are implemented in the directory instead of in the network switches and, besides, they aim to improve the cache latency rather than generated traffic.

7.1 Introduction

The implementation of requests by means of broadcast messages in small systems is a good solution because they provide direct and fast communication between processors. Thus, on a cache miss occurrence, processors can quickly find where the memory block they require is by using broadcast messages. However, this class of message has a serious disadvantage: the scalability. While the bandwidth provided by the interconnection network usually only

increases linearly to the system size, the bandwidth requirements of broadcast messages increase quadratically to the system size. Therefore, although in small systems the network can provide enough bandwidth for broadcast messages, in medium and large systems they will require much more bandwidth than that provided by the interconnection network. Thus, due to their lack of scalability, broadcast messages are not a good choice for systems with a considerable number of processors.

The starvation prevention mechanisms proposed so far are based on broadcast messages. This makes the Token Coherence protocol only suitable for small systems, but not for medium or large systems. To address this problem, in this chapter we propose a switch-based technique which allows to pack several broadcast messages in just one message. By doing this, the network traffic can be reduced drastically. Indeed, when the reduction is really high, the bandwidth required by broadcast messages becomes similar to that required by several point-to-point messages, which are scalable. Furthermore, the traffic reduction also affects other messages transmitted through the network, which can help to reduce their latency since the network will be less congested. Besides, as the packing rate increases proportionally to the system size, this technique can provide broadcast messages with certain scalability. Thus, the starvation prevention mechanisms can benefit from this technique, making Token Coherence suitable even for medium/large-sized systems.

The packing technique that we describe next can be applied to whatever of the starvation prevention mechanisms proposed for Token Coherence. However, in this chapter we focus its description on the priority request mechanism because of several reasons, leaving as a future work its application to persistent requests. The main reasons for applying this technique to priority requests are (1) due to the routing algorithm used by priority requests, it is likely several priority requests to coincide along the same path (visited switches), being more feasible the packing of several messages; (2) unlike persistent requests, several priority requests can be served at the same time (as they do not require acknowledgments). Therefore, the priority request packing can accelerate the priority request reception and, in turn, their completion; and (3) we aim at improving the most efficient starvation prevention mechanism and, as we saw in previous chapters, the priority request mechanism outperforms the persistent

request one.

Notice that the starvation situations usually happen because several processors wish to simultaneously access to the same memory block. When such a situation occurs, the processors practically realize at the same time. Therefore, the priority requests that they broadcast to solve it will be sent near simultaneously. Besides, these priority requests for the same memory block must be sent through the same ordered path to ensure they are received in order. Thus, the priority request messages for the same memory block are likely to coincide in the buffers of the network switches placed along the route to their destinations. In addition, the priority requests for the same memory block have in common the majority of their fields. Therefore, when several of those messages are broadcast, a lot of redundant information will flood the interconnect, which unnecessarily consumes a significant part of the whole bandwidth.

To tackle this drawback, we take advantage of the fact that (1) the priority requests for the same memory block usually coincide in the input buffers of the network switches and (2) those requests practically contain the same information to pack (and compress) several messages in a single message (priority request pack). The components in charge of performing the packing are the network switches, which will remove a lot of redundant information, thereby saving a considerable quantity of endpoint traffic[1]. Let us show this by an example. Imagine a 32-processor system where 10 processors each broadcast a priority request. The endpoint traffic due to the generated priority requests will be:

$$10 * 8 * 32 = 2560 \text{ bytes}$$

10 messages, 8 bytes per message, and 32 destinations. However, if we are able to pack the 10 priority requests in just one pack, the generated endpoint traffic will be:

$$1 * 18 * 32 = 576 \text{ bytes}$$

[1]We refer to endpoint traffic as the traffic received by components, as defined in [76].

destination	type	size	requester	address	operation	completed PR
1 byte	2 bits	1 byte	1 byte	4 bytes	1 bit	2 bytes

Figure 7.1: Priority request format. The fields common to all the priority requests upon the same memory block are shown in shadow.

1 message, 18 bytes per message[2], and 32 destinations. Therefore, in this case, we could have saved up to 77,5% of the generated endpoint traffic. This reduction increases as the number of priority requests packed in a single message increases. Thus, this solution may provide scalability to broadcast messages, as the more priority requests the nodes inject, the more packing the switches will be able to carry out, preventing the endpoint traffic from increasing quadratically with the system size. Additionally, the performance may be improved when it is jeopardized by the use of priority requests, which is common is medium and large systems.

In what follows, this technique is explained in more detail.

7.2 Format of Priority Request Packs

Priority request messages are mainly composed of the fields shown in Figure 7.1, being their total size 10 bytes. The priority requests for the same memory block hold almost the same information. In fact, they share most of their fields which are marked in gray in Figure 7.1. The only fields that differ in a set of priority requests for the same memory block are the *requester* field, the *operation* field, and the *completed PR* field. Thus, only 3 out of 10 bytes may differ in a set of priority requests for the same memory block. Therefore, when a starvation situation happens, although several priority requests are broadcast flooding the interconnection with a lot of information, only about the 30% of that information will be useful, as the remaining 70% is replicated and, consequently, could be saved. Taking into account this fact, the packing of several priority requests into a single pack will let us remove the redundant information.

[2]In this case, as will be shown the pack size is larger than the size of individual messages because the pack has to code several issuers.

address	destination	type	size	operation	requester set	completed PR set
4 bytes	1 byte	2 bits	1 byte	1 bit	n bytes	2n bytes

Figure 7.2: Format of priority request packs. The fields in shadow are those that change with respect to the format of priority requests.

From here on, we will use the term *priority request pack* or just *pack* to refer to several priority requests packed in a single message. In fact, we are going to use the term pack in case of a single priority request, since it can be considered as a pack with only one priority request. Figure 7.2 illustrates the format of packs. As shown, the pack format is slightly different from the priority request format. Although the number of fields does not change, the *requester* field becomes the *requester set* field and the *completed PR* field becomes the *completed PR set* field.

The *requester set* field is a list of the processors that request the memory block identified by the *address* field. *Requester set* must be implemented as a list and not a bit vector because, although several priority requests are packed in the same message, it is still necessary to maintain their global order. Notice that a list of processors can keep the order of the messages, meanwhile a bit vector can not keep it. The size of the *requester set* field will depend on the number of packed messages. Hence, if a pack contains n priority requests, its size will be n bytes. Note that its size will be independent of the system size, which does not compromise the scalability of the messages.

Like the *requester set* field, the *completed PR set* field is the list of the identifiers held by the different priority requests included in the pack. The identifiers must be placed in the same order as that occupied by its corresponding requester in the *requester set* field. The size of this field will also depend on the number of packed messages. Hence, if a pack contains n priority requests, its size will be $2n$ bytes. Note that, if we assume one outstanding priority request per processor, it is not necessary that the packs include the *completed PR set* field, since a request from certain processor will always replace the stored request from the same processor.

As shown in Figure 7.2, the *operation* field is implemented as a single bit, not as a list containing all the operations. We do this because we only

pack priority requests for the same memory block and which have the same operation type, that is, either read requests or write requests. This is because we have observed that most of the requests are read requests.

Notice that, besides modifying these two fields of the priority request messages, we have also changed their position. As we will see in more detail later, this is done to ease the packing process and avoid introducing an additional delay in the critical path of priority requests.

7.3 General Packing Process

Switches carry out the following strategy to implement the packing of priority requests:

1. When a switch receives a pack for a certain memory block, it checks whether that memory block matches the memory block requested by the pack stored in the last position of the buffer where the received message should be stored.

2. If a match occurs, the requester/s of the received pack (*requester set* field) and the information about the completed priority requests to replace (*completed PR set* field) will be added to the corresponding fields of the stored pack. After this, the remaining fields of the received pack can be discarded.

3. If a match does not occur, the priority request pack is queued in the last position of the corresponding buffer.

4. When a pack reaches the head of the buffer, it will be no longer possible to add new requests to the pack, as its transmission could begin at any time.

Note that, according to the described strategy, priority requests do not have to wait for other messages to form a pack (which would delay their transmission) since they are routed as soon as the requested output link becomes free. Indeed, what we propose is to take advantage of the elapsed time that a certain priority request message has to be waiting for being routed to

compound a single message (pack) by packing the priority request messages received during that time. Thus, if the interconnection network is congested due to the high quantity of injected broadcast messages, several priority requests will coincide in the buffers. Therefore, the packing of those messages into a single one will contribute to alleviate the network traffic. On the contrary, if priority requests do not coincide in the buffers, it will mean that the interconnection network can support and efficiently handle that traffic and, therefore, the broadcast messages do not entail a problem for the network at that time. Thus, the packing technique would no be used because it would not be necessary.

In the following two sections, we describe in detail how the matching and packing of priority requests is carried out.

7.4 Checking Message Matching

When a switch receives a priority request pack, it has to check whether the incoming pack and the last pack stored in the queue are similar enough (that is, they correspond to the same memory block and request the same operation) to compose a single message. This checking must be quite simple and it should not require a significant hardware complexity. To this end, we make the following assumptions:

- Priority requests use a dedicated virtual channel. This simplifies the process as the checking is reduced just to the messages routed through that virtual channel.

- A received message only can be packed into the last message stored in the buffer. Besides, that message can not be placed at the head of the buffer because it means that the sending of the message could begin at any moment.

- Each queue has two reading ports: one port to read the head of the queue (that is, the message that is being transmitted) and another port to read the tail of the queue (for carrying out the checking between the last stored pack and the incoming pack).

7.5 Packing Priority Requests

When a switch receives a pack which requests a memory block that matches the one requested by the last stored pack, a packing is performed. The final pack is generated by doing three operations.

- First, the *requester set* field of the incoming pack is concatenated with the *requester set* field of the stored pack. This concatenation lets packs maintain the global order of priority requests.
- Second, the *completed PR set* field of the incoming pack is concatenated with the *completed PR set* field of the stored pack. Like in the previous case, the concatenation allows to maintain the global order.
- Third, the other fields of the incoming pack (*address*, *destination*, *type*, ...) are discarded, since they are already present in the stored pack.

To dynamically compose the packs (the *requester set* and *completed PR set* fields) without affecting the latency of messages through the switches, the organization of the fields that compose a pack has been slightly modified. As shown in Figure 7.2, the first field of packs is the *address* field. Placing that field in the first place allows to quickly perform the comparison between the incoming and the stored pack because, as soon as the *address* field of the incoming pack is received, the comparison can be performed (it would not be necessary to wait for receiving the whole message). While the comparison is being performed, the rest of fields of the incoming pack will complete their reception.

To ease the packing, a decoupling buffer is placed together with each queue dedicated to priority requests. This decoupling buffer holds the *completed PR set* field of the last pack in the queue. On a match occurrence, the *completed PR set* field of the incoming pack is copied to the end of the decoupling buffer, thereby performing easily the concatenation. Note that, if an incoming pack does not match the last stored pack, the current contents of the decoupling buffer will have to be copied to the tail of the stored pack[3] before queuing the

[3]This copy will hardly delay the queuing of the incoming message because as soon as the comparison finishes, the copy could be performed.

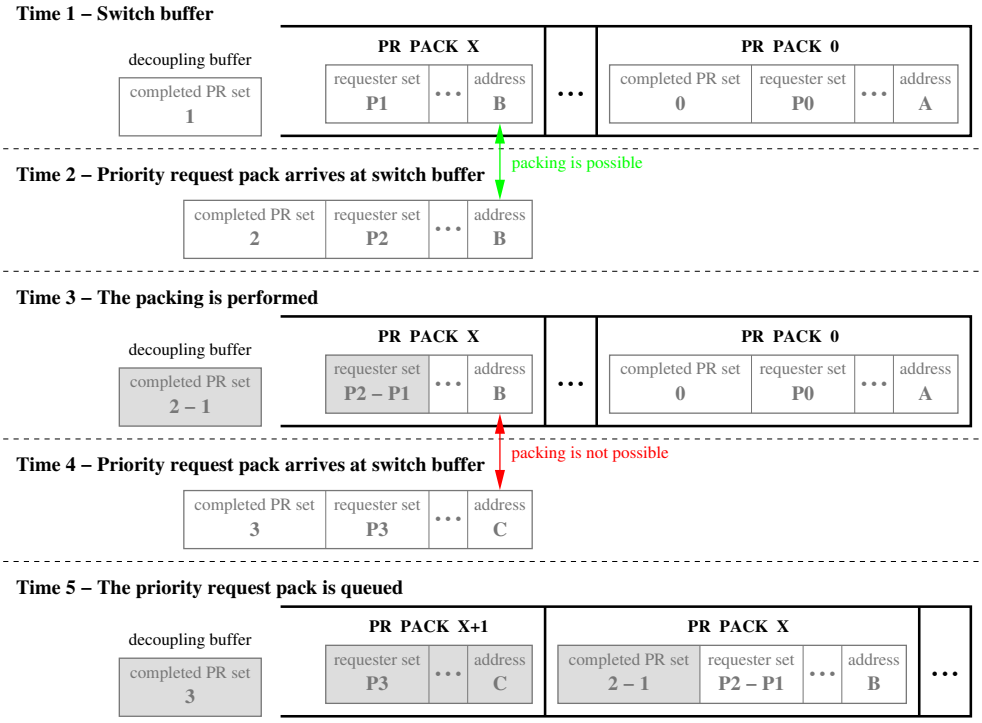


Figure 7.3: Example of message packing.

incoming pack. In this way, the final pack is completed by adding to its end the *completed PR set* field that has been composed in the decoupling buffer. After this, the incoming pack can be queued in the buffer and the decoupling buffer can be updated with the *completed PR set* field of the incoming pack.

On a match occurrence, besides the concatenation of the *completed PR set* field, the *requester set* field of the incoming pack will be copied to the end of the corresponding buffer. This is done to compose the *requester set* field of the final pack. Note that it is possible to perform the concatenation of the *requester set* fields in such an easy way because the last field of the last pack in the queue is the *requester set* field, given that its *completed PR set* field is in the decoupling buffer.

Figure 7.3 shows an example of how switches perform the packing. The figure shows the state of a queue of packs. Initially, the queue contains several packs. The last pack in the queue only contains a *P1*'s request for the memory block B. Note that, the information of its *completed PR set* field is temporarily

stored in a decoupling buffer. At time 2, the queue receives a new pack. The incoming pack contains a priority request for the memory block B. As this memory block matches the block requested by the last stored pack, a packing can be performed. This consists in putting the *requester set* field of the incoming pack (P2) at the end of the queue and the *completed PR set* field at the end of the decoupling buffer. The rest of the fields of the incoming pack are discarded. At time 4, the queue receives another pack. In this case, the *address* field of the incoming pack does not match the *address* field of the stored pack. Therefore, a packing is not possible and the incoming pack must be queued. However, before queuing it, the information stored in the decoupling buffer is reassociated to its pack by copying it at the end of the queue. This operation will not entail an increase in the latency because the address comparison is performed before having received the whole packet completely. Once the pack is received completely, it is placed at the end of the queue, storing the *completed PR set* field in the decoupling buffer.

7.6 Increasing Packing Opportunities

The main advantage of the proposed technique is its simplicity. However, there are some situations that can prevent the technique from taking advantage of its whole capabilities. In this section we analyze these situations and we propose several alternative implementation options that increase the number of opportunities to maximize the number of generated packs.

7.6.1 Increasing the Number of Ordered Paths

We assume that there is a single ordered path for the transmission of all the priority requests. Therefore, the probability to form packs will be lower than when several ordered paths are assumed. This is because priority requests for different memory blocks follow the same path. As a result, most of the priority requests for the same memory block do not arrive consecutively to switches, preventing the formation of a single pack.

The impact of this situation can be alleviated by using different ordered paths. As a result, the priority requests for the same memory block are likely

address	destination	type	size	operation set	requester set	completed PR set
4 bytes	1 byte	2 bits	1 byte	n bits	n bytes	2n bytes

Figure 7.4: Pack with priority requests for different memory operations.

to be received consecutively by switches, making more probable the formation of a single pack.

7.6.2 Allowing Different Request Types

Another factor that can affect to the packing is the request type (read or write). If a starvation situation is caused because several processors contend for the same memory block and some of them want to read it and others want to write it, processors will broadcast both priority read and write requests. However, as long as the priority read requests and the priority write requests are received interleaved, the generation of a single pack will not be possible.

To address this problem, we could allow a single pack to include priority requests for different memory operations. To this end, the pack format should be slightly modified as shown in Figure 7.4. In this case, packs include a list of the memory operations requested by each requester (*operation set* field). Like the *completed PR set* field, the order in which the memory operations are stored in the *operation set* field must be in accordance to the order in which the requesters are stored in the *requester set* field. As shown in the figure, the size of packs may increase because of the *operation set* field. However, this increase will not be significant because only 1 bit is required to code each memory operation. Thus, if we pack n priority requests into a single message, the size of the *operation set* field will be n bits.

To be able to continue easily packing several priority requests, besides using a decoupling buffer to concatenate the *completed PR set* fields, we will also require a decoupling buffer to concatenate the *operation set* fields. The use of this buffer would be similar to that for the *completed PR set* fields.

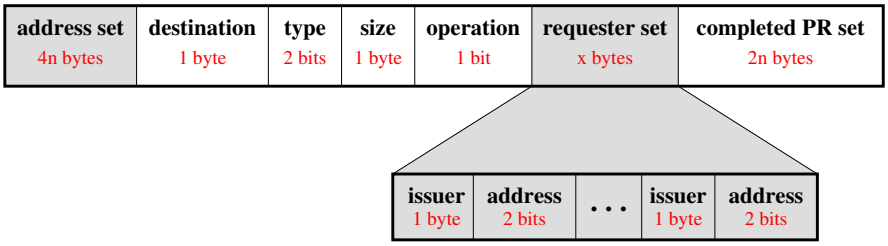


Figure 7.5: Pack of priority requests upon different memory blocks. A maximum of 4 different addresses can be contained in a pack.

7.6.3 Allowing Different Memory Addresses

When several *non-consecutive* priority requests for the same memory block coincide in the buffers, the proposed strategy (simplified to ease its implementation) will not detect the match between the packs, thereby preventing the generation of a single pack. To solve this problem, in this section we propose to allow to pack priority requests for different memory blocks. Note that the reduction (in bytes) of packing several priority requests for different memory blocks is very small. However, this solution is indeed intended to pack the priority requests for the same memory block that are not received consecutively.

To exploit the capabilities of this implementation option without loosing efficiency, the format of packs, shown in Figure 7.5, is slightly different from that of the previous proposals. As shown, the different memory blocks are coded in the *address set* field as a list. However, in order to avoid that a memory block address appears several times in the *address set* field (which would unnecessarily increase the pack size), we limit to one the number of times that a requested block can appear in the cited field. Therefore, to know which memory block is requested by each processor, each processor in the *requester set* field is coded together with an identifier indicating the position that the memory block requested by it occupies in the *address set* list. For this purpose, we trust on limiting the number of different memory blocks per pack. In this sense, limiting it to, for example, 4 different memory blocks per pack, each processor in the *requester set* field would require only 2 additional bits for indicating its requested memory block (among all of those included in

Table 7.1: Possible packing situations.

number of different addresses in the incoming pack	number of different addresses in the stored pack			
	1	2	3	4
1	X	X		
2	X	X		
3	X			
4	X			

the *address set* field). Let us show the benefits of using this approximation by an example. Imagine a 32-processor system where 5 processors each broadcast a priority request for a memory block A and other 5 processors broadcast a priority request for a memory block B. According to the technique proposed in previous sections, if the intermediate switches receive the priority requests interleaved (A, B, A, B ...), a pack will not be able to be generated, being the total endpoint traffic $10 * 8 * 32 = 2560$ bytes (10 messages, 8 bytes per message, and 32 destinations). However, if we apply the changes proposed in this section, the 10 priority requests could be packed into one pack. In this case, the generated endpoint traffic would be $1 * 25 * 32 = 800$ bytes (1 message, 25 bytes per message, and 32 destinations). Therefore, in this case, we could save up to 70% of the generated endpoint traffic. Note that, in this case, the size of the pack is bigger than that assumed in previous proposals for the same number of included priority requests (25 bytes against 18 bytes). However, the packing is now possible.

To implement this option without affecting the critical path of messages, the number of different addresses that can be included in a pack is bounded to four. Table 7.1 shows the situations where it is possible to perform a packing. The row indicates the number of different addresses included in the stored pack and the column indicates the number of different addresses included in the incoming pack. Only in the cases marked as X it is possible to pack two

messages (regardless of the address matching). Otherwise, the incoming pack will be queued at the end of the buffer, since a new pack will not be able to be generated. This approach does not affect the critical path of messages because it allows the overlapping between the address comparison and the message reception. On a pack reception, to easily know if we are in presence of any of the situations shown in the table, each priority request pack has a *counter* field (two bits) which indicates the number of different addresses that it holds. At a pack arrival, the *counter* fields of both the incoming pack and the stored pack are used to know if the packing can be performed or not.

The process of packing is similar to that described in previous sections. When the first address of the incoming message is available, it is compared with the addresses of the stored pack[4]. If a math occurs, it is not necessary to add that address in the stored pack. Otherwise, the address is inserted in the *address set* field of the stored pack. To easily concatenate the *address set* fields, the switch allocates enough space in the buffer to insert, at most, four different addresses. Any way, the address identifiers of the *requester set* field of the incoming pack will have to be updated accordingly, since the order of the addresses in the *address set* field of the new pack may change. To this end, the position that each address will have in the new pack is stored in a buffer (2 bits per address). Thus, when a *requester set* field is received, its identifier is updated with the offset stored in the corresponding buffer before being stored in the input queue. Like in the previous proposal, the *completed PR set* field of the last message is stored in a decoupling buffer.

The technique described in this section has the aforementioned advantage of packing non-consecutive priority requests for the same memory block. However, depending on the number of different addresses allowed per pack (limited to four in this dissertation), its complexity might increase considerably.

[4]Note that, according to Table 7.1, at worse, an address of an incoming message will have to be compared with 2 addresses. Therefore, we will assume that the time required to perform 2 comparisons is shorter than the time required to finish the next address (or the whole pack) reception.

7.7 Adjusting the Starvation Detection Timeout

As previously commented, processors use a timeout to estimate whether their transient requests are suffering from starvation. According to the performance policy defined in Section 5.8, the timeout is set to twice the processor's average miss latency. Setting the timeout to that value prevents a slightly delayed response from causing starvation, but it also detects starvation quick enough as to avoid a large performance penalty when a protocol race occurs. Besides, this policy adapts to different interconnect topologies and traffic patterns.

Since the value of the timeout depends on the cache latency, several factors will influence when calculating it. Thus, for example, one of the factors that can cause a variation in the timeout is the size of messages. If we use two different systems which are exactly the same, but we use messages of different size, the timeout intervals estimated in each case will differ because the size of messages affects the cache miss latency and, therefore, it also affects the timeout. Consequently, if the value of the timeout is estimated under certain circumstances, its value will only be suitable when the system moves under those circumstances. Note that this can entail a problem when we use the packing technique because the timeout intervals are mainly calculated in absence of priority request packs. However, when several priority requests are packed into a single pack, we are using messages whose size can be considerably different from that of, for example, transient requests. Therefore, the timeout interval calculated using transient requests may not be suitable when we use priority request packs. To improve this aspect, we propose several alternatives that can address this problem and that can make the system perform even better:

- Given that the size of packs is larger than the size of transient requests, the messages coinciding with packs through the network may have a higher latency, which may increase in some cases the average latency of cache misses. Therefore, to avoid assuming starvation when the message is simply delayed in the interconnect, the simplest solution is to use a higher timeout interval. For instance, setting the timeout to *three* times the processor's average miss latency. Although correct, this solution has the disadvantage of increasing the time required to detect starvation,

which can cause a large performance penalty in some scenarios.

- Another choice would be to use different timeouts based on the elapsed time from the reception of the last pack. Thus, for example, if a pack has not been received lately, we will assume that the network will not contain any pack and, therefore, we can use the typical timeout (twice the processor's average miss latency). However, if a pack has been received lately, we assume that other packs may be in transit and, therefore, a higher timeout can be required. In this case, we could set the timeout to *three* times the processor's average miss latency. Hence, we use two different dynamics timeouts depending on the possible existence of packs in transit.

- The last proposal we make is to adjust the timeout as long as new packs are received. Thus, when a pack is received we slightly increment the value of the timeout because it means that that pack may have delayed the reception of a response. For example, if the timeout is set to 200 ms and a new pack is received, the timeout is increased in a certain percentage, let us assume 5%, setting it to 210 ms. As consecutive packs are received, the value of the timeout will increase. However, when processors do not receive more packs (because there are not more starved requests), the value of the timeout naturally will recover little by little its original value (previous to the increase) since the following transient requests will decrease it (if succeed).

7.8 Packing Non-Silent Invalidations

The technique proposed in this section has been associated to the packing of broadcast messages, specifically to priority requests. However, this technique can be applied in other situations. For example, it can be used to alleviate the problems caused by the non-silent invalidations. Thus, in this section we explain how the technique can be used in that case.

When a processor wants to write a memory block, it needs to get an exclusive copy of the block. Therefore, before performing the write, it will have to invalidate all the copies held by other components. The invalidation is

performed by sending a write request. This invalidation is non-silent because the issuer of a write request will have to wait for an acknowledgment from each node that maintains a copy of the block. Those acknowledgments are data-less token messages. When the issuer receives all those messages, it knows that the copies of the memory block have been invalidated and, therefore, it holds an exclusive copy. Only when it is sure that it holds an exclusive copy, it will be able to write the block. This will entail a problem when a lot of processors share the same memory block because, although the data-less response messages are not very large, the writer will have to wait for receiving a message from each sharer. Besides, the more processors the system has, the more sharers there may be and, therefore, the worse the situation may become. Consequently, as data-less responses are in the critical path of write requests, it is desirable that they are received as soon as possible. Furthermore, the writer processor may become temporarily a bottleneck since a write request will result in a burst of packets directed to a single node (the writer).

To improve this aspect of token-based protocols in this section we suggest to apply the proposed packing technique to data-less responses. Notice that in this case the message packing could be carried out without increasing the size of the resulting message and without modifying their format. By packing several messages into just one, we can reduce the number of acknowledgments that the writer has to receive. By reducing the number of acknowledgments, the writer will be able to perform the operation sooner, which can contribute to reduce the latency of write requests.

Taking into account that all the data-less responses are directed to the writer, they are likely to coincide along their way to the destination. In fact, they all will have to go through the switch which the writer is connected to. This situation is shown in Figure 7.6.

The process to pack several data-less responses is similar to that described in previous sections:

1. When a switch receives a response message (which uses a dedicated virtual channel), it checks if both the incoming response and the last stored response in the switch buffer are data-less responses for the same memory block.

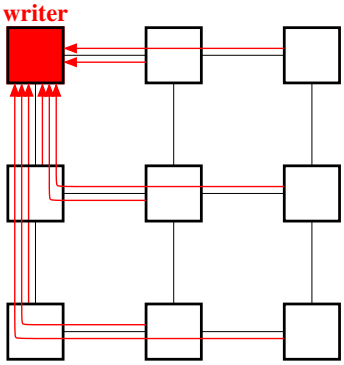


Figure 7.6: Example of data-less responses. The node marked in red wants to write a memory block and it has to wait for receiving all the block's tokens.

2. If the tokens belong to the same memory block and both messages go to the same destination, a packing is performed. The packing only consists in (1) adding the tokens held by the incoming message to the stored one and (2) updating the *completed PR* field.

3. If the tokens does not belong to the same memory block or they go to different destinations, the incoming response is queued at the end of the buffer.

4. When a response message reaches the head of the queue, no more tokens can be included in it.

Note that in this case the packing process is even simpler that in case of priority requests. In fact, the format of the messages does not change. The packing process consists in:

- The number of tokens held by the incoming response are added to the number of tokens held by the stored response.

- If the value of the *completed PR* field of the incoming response is higher than that of the stored response, then it replaces the *completed PR* field of the stored message. Otherwise, it is discarded. By doing this, we maintain the value of the last completed request.

7.9 Evaluation

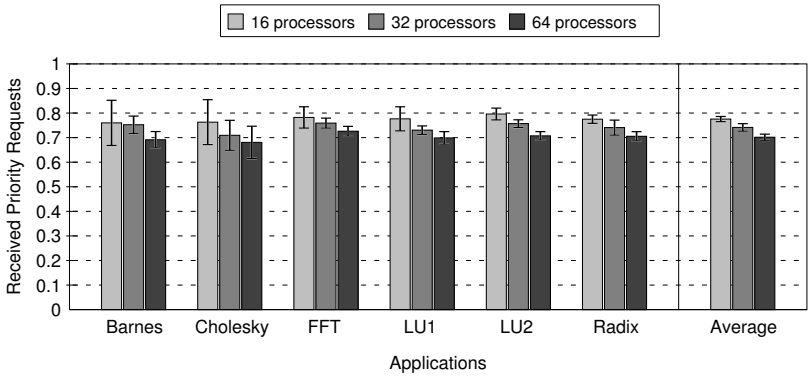
In this section, we analyze the contribution of the proposed packing technique. To this end, we first apply the packing technique to priority requests and, later, we apply the packing technique just to data-less responses.

7.9.1 Target System and Parameters

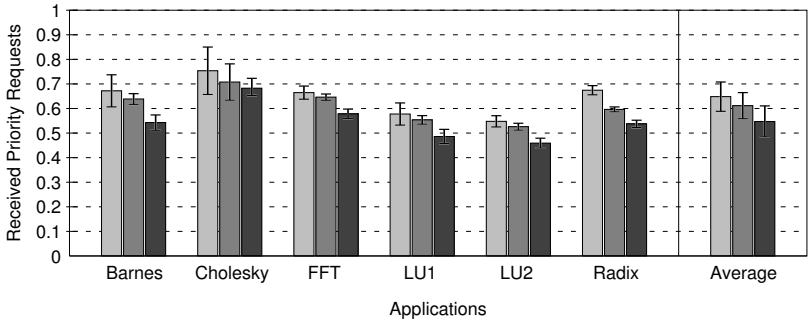
We simulate three target systems: 16, 32, and 64-processor Sparc v9 systems using the parameters described in Chapter 4. We assume that processors are only connected through an MIN interconnect with the perfect shuffle permutation because the results for a 2D mesh are similar.

First, we show the results of applying our proposal to the priority request mechanism. The evaluated proposal allows the packing of requests upon different memory blocks as explained in Section 7.6.3. However, note that we only assume a single ordered path for all the priority requests and we only allow to pack priority read requests, which simplifies the implementation. We also show the comparison between the packing technique using the typical timeout (twice the processor's average miss latency) and the packing technique using a more suitable timeout (three times the processor's average miss latency) as described in 7.7. The figures labeled as *t2* will refer to the approximation with the typical timeout, while the figures labeled as *t3* will refer to the approximation with the more suitable timeout. In this case, the packing technique is evaluated in terms of received priority requests, endpoint traffic, average latency of completing starved requests, link utilization, and runtime of the applications. *16p*, *32p*, and *64p* refer to the results obtained in a 16-processor, 32-processor, and 64-processor system, respectively. All the results are normalized with respect to the data obtained for the priority request mechanism without using packing.

Second, we apply the packing technique to control responses and we analyze the obtained results. In this case, we do not need to use a different timeout because the packing technique does not generate messages with different sizes. In fact, the packed messages keep their original size. Like in the previous case, all the results are normalized with respect to those obtained for Token Coherence without using the packing technique.



(a) typical timeout (t2)

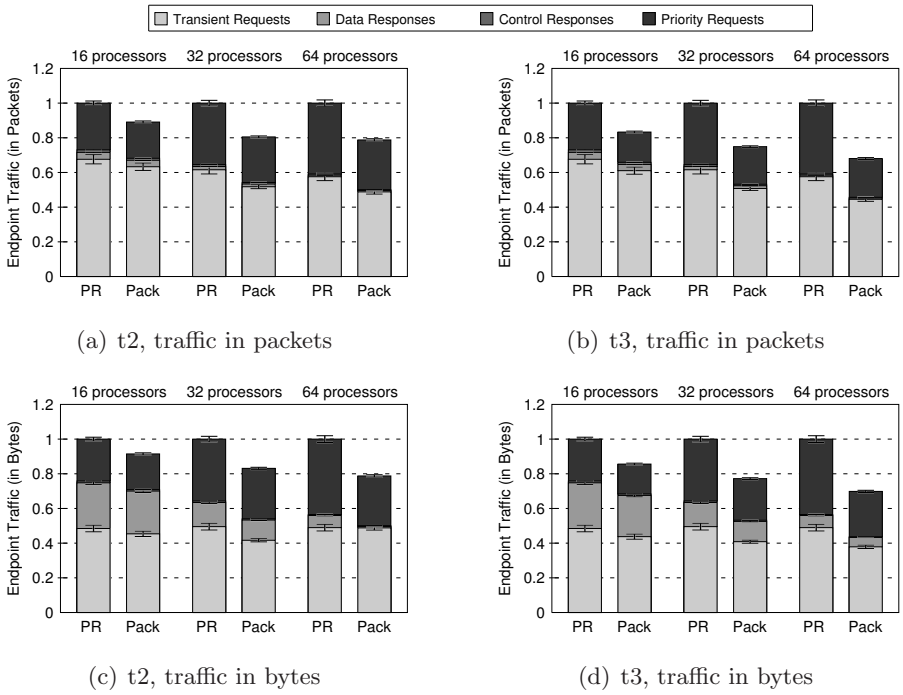


(b) suitable timeout (t3)

Figure 7.7: Normalized endpoint traffic due to priority requests when using packing and different timeouts.

7.9.2 Priority Request Endpoint Traffic

Figure 7.7 shows the traffic due to the reception of priority requests when the packing technique is applied. This results are normalized to the traffic received when the packing technique is not applied. As shown in Figure 7.7(a), the packing technique leads to a significant reduction in the number of received priority requests. This is due to two main reasons: (1) several priority requests are packed into a single message, which reduces the number of received requests, and (2) the packing lowers the number of starved requests (as we will see later). Besides, the reduction is higher as the system size increases, reducing in about 30% (in average) the number of received priority requests without using the packing in a 64-processor system. Figure 7.7(b)



(a) t2, traffic in packets

(b) t3, traffic in packets

(c) t2, traffic in bytes

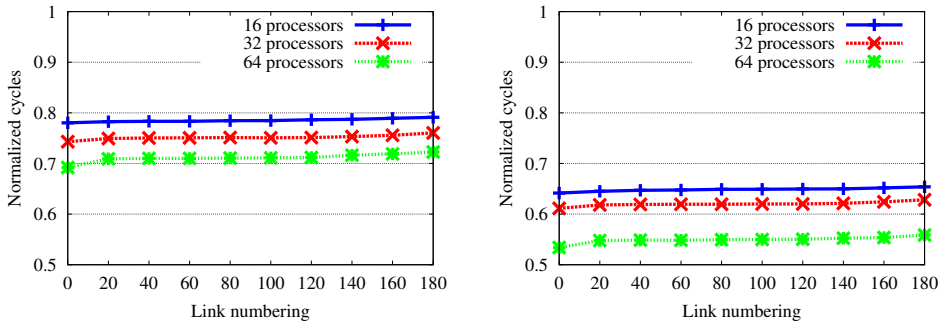
(d) t3, traffic in bytes

Figure 7.8: Normalized total endpoint traffic. *PR* stands for Token Coherence using only normal priority requests and *Pack* stands for Token Coherence using the packing technique.

shows that this reduction can be larger when, besides the packing technique, a more suitable timeout is used, reaching a reduction of about 47% in average in a 64-processor system.

7.9.3 Overall Endpoint Traffic

Figure 7.8 illustrates the normalized endpoint traffic generated during the execution of the applications. *PR* refers to the protocol using only priority requests and *Pack* refers to the same protocol using the packing technique. We only show the average values in these figures to reduce the number of figures. As shown Figure 7.8(a), the use of the packing technique lowers the number of cache misses, which, in turn, reduces the number of *Transient Requests* mainly because, since the broadcast messages cause less congestion in the network, tokens and memory blocks are transmitted faster and, therefore,



(a) typical timeout (t2)

(b) suitable timeout (t3)

Figure 7.9: Link utilization normalized to the link utilization without using the packing technique.

they stay longer in nodes and shorter time traveling through the network. The reduction of *Transient Requests* causes, in turn, the number of responses (*Data Responses* and *Control Responses*) to lower. Besides, the number of requests suffering starvation decreases because the network is less congested and it is used in a more efficient way. As a result, the total endpoint traffic decreases, being this reduction more significant as the system size increases. In particular, in a 64-processor system, the packing technique reduces about 20% the total endpoint traffic (in packets and in bytes) when it is used in a 64-processor system. When the packing technique is used together with a suitable timeout, the reduction of endpoint traffic grows even more, reaching a reduction of about 35% in average, which is shown in Figure 7.8(b) and Figure 7.8(d).

7.9.4 Link Utilization

Figure 7.9 shows the number of cycles that the links are busy due to priority request traffic when the packing technique is used. The data are normalized to the number of cycles that links are busy due to priority requests when the packing technique is not applied. These figures only show the utilization of the links used for going from the root switch to the destinations, since the utilization of the links used for going from the source node to the root switch is much less significant. This is because while a priority request is going

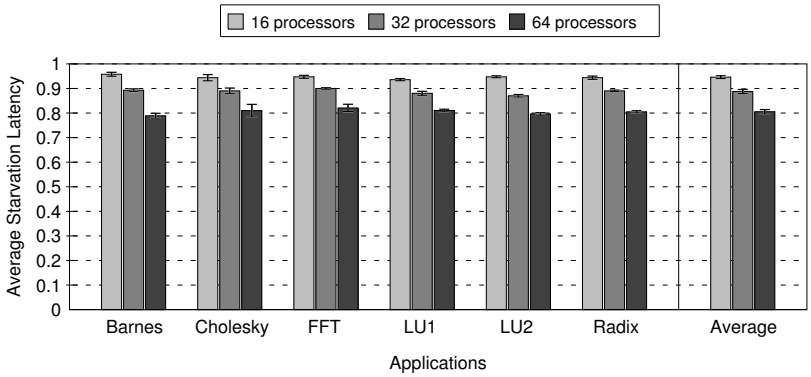
to the root switch, it only uses one uplink (it behaves like a point-to-point message). However, when the request is going from the root switch to the destinations, it uses all the downlinks and it is at that time when the network begins to be flooded. Like in the previous section, we only show the average results to decrease the number of figures. As shown in Figure 7.9(a), the packing technique reduces the overall traffic that goes through the network. This reduction varies between about 20% in a 16-processor system and 30% in a 64-processor system. Therefore, the reduction grows with the system size, mainly because the more processors the system has, the more congested the network is and, as a result, a greater number of packs will be able to be generated. Figures 7.9(b) depicts that the packing technique together with a suitable timeout can significantly increase the reduction, decreasing in about 45% the traffic crossing the network due to priority requests.

7.9.5 Starvation Latency

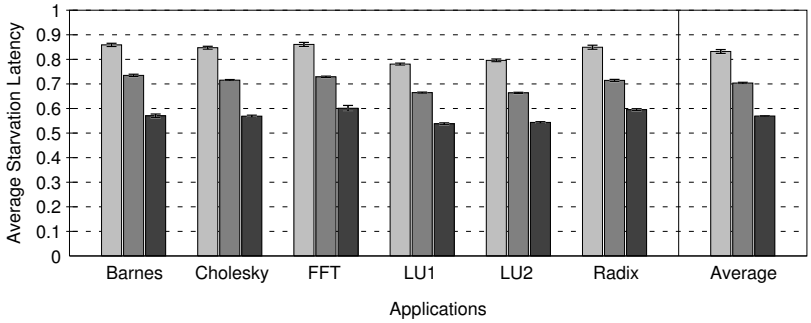
Figure 7.10 depicts the normalized average latency of completing a starved request. It includes the elapsed time from a processor detects possible starvation up to that processor receives all the requested tokens. As shown in Figure 7.10(a), the packing of priority requests decrements the latency of completing starved requests. This decrement is higher as the system size grows. This reduction happens because, when priority requests are packed, they do not reach the destinations sequentially. Rather, several priority requests can arrive simultaneously at several destinations allowing them to served them more quickly. Besides, since the traffic in network lowers, the rest of messages (responses) suffers shorter delays, which contributes to reduce the average latency of completing starved requests too. This latency can be improved even more by using a suitable timeout, such as Figure 7.10(b), where the latency is reduced more than 40% in average in a 64-processor system.

7.9.6 Runtime

Figure 7.11 shows the normalized runtime of the applications. As shown, the reduction in both traffic and latency hardly has effect in the runtime of the applications in 16-processor and 32-processor systems. This is because, in



(a) typical timeout (t2)

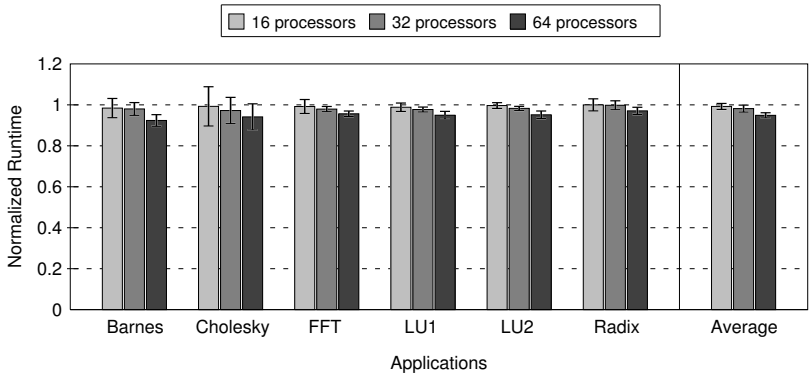


(b) suitable timeout (t3)

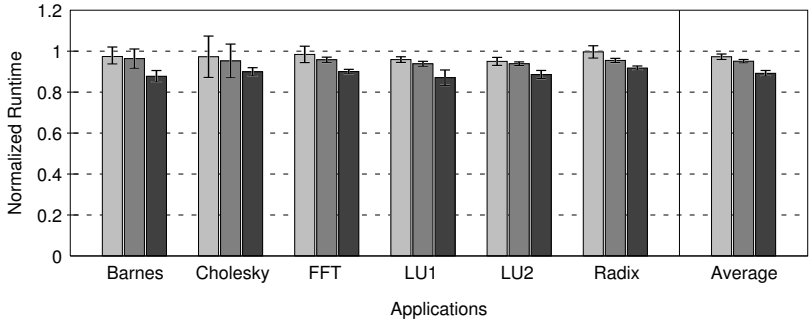
Figure 7.10: Normalized latency of completing starved requests when using packing.

those systems, the use of broadcast messages is not still a big problem for the performance[5]. Therefore, in general terms, the packing of priority requests in 16-processor and 32-processor systems does not contribute to reduce the runtime of the applications. However, as the system size increases, the effect of broadcast messages over the overall performance is more significant. Hence, in 64-processor systems or larger systems, the use of packing techniques will not only contribute to reduce the traffic, but to reduce the runtime of applications too. This will be more significant when, besides the packing technique, a suitable timeout is used. Thus, according to Figure 7.11(b) the packing

[5]As commented in other works [76], the broadcast-based solutions are the most efficient options for systems with a moderate number of nodes.



(a) typical timeout (t2)



(b) suitable timeout (t3)

Figure 7.11: Normalized runtime when using packing.

technique could reduce in about 10% the runtime of applications (in average) in a 64-processor system.

7.9.7 Packing Control Responses

Following we show the effects of applying the packing technique just to control response messages to alleviate the problem caused by the non-silent invalidations, as explained in Section 7.8. Figure 7.12 illustrates the rate of received/sent control response messages. Obviously, when the packing technique is not used, the rate is 1 since processors receive the same number of control responses as they sent. However, when several control responses are packed while in transit, processors receive less control responses than they injected. As shown in Figure 7.12, the rate of received/sent control responses

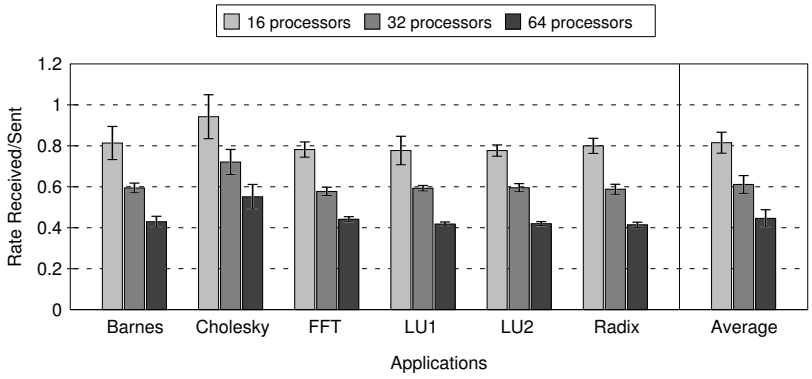


Figure 7.12: Received/Sent control response rate when using packing over control response messages.

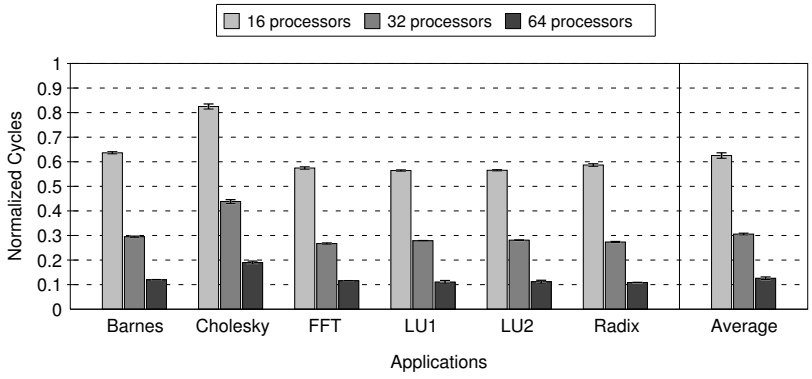


Figure 7.13: Reduction in the latency of control response messages when using the packing technique.

decreases significantly according to the system size, reaching a reduction of almost 60% in average in case of a 64-processor system. Note that this reduction will not lead to a significant reduction in the traffic transmitted through the network because the packing is when messages are near the destination, but the reduction can considerably alleviate the input buffers of the NIC or the node.

Besides reducing the congestion in the input buffers of the NIC or the node, applying the packing technique over the control responses can help us to decrease their average latency, which is depicted by Figure 7.13. As shown, the packing technique has a huge effect over the average latency of control

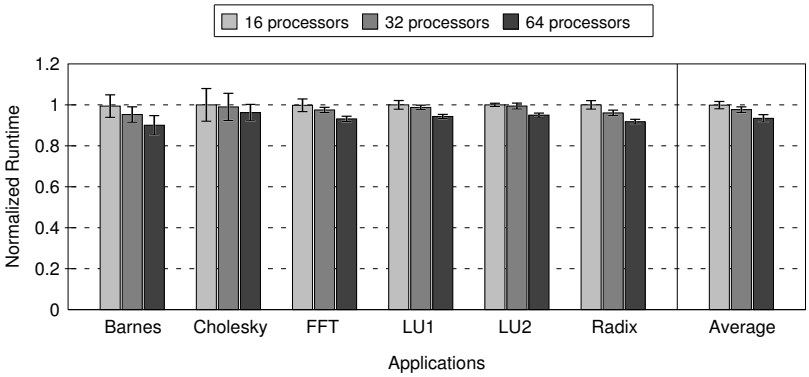


Figure 7.14: Normalized runtime of applications when applying the packing technique to control responses.

responses, reducing the their average latency in approximately 90% in average in a 64-processor system. In fact, when the packing technique is applied, the average latency of control responses decreases as the system size increases.

Figure 7.14 shows the runtime (when the packing technique is used) normalized to the runtime without using the packing technique. As shown, in 16-processor and 32-processor systems, the packing technique does not have a great influence on the runtime of applications because in those systems the non-silent invalidations do not entail a serious problem. However, as the system size increases, their effects will be more significant and, therefore, the use of the packing technique can help to reduce the runtime of applications. Thus, in a 64-processor system the runtime of applications can be reduced in about 10% in average. Note that, when we apply the packing technique, we are only acting during a starvation situation. However, when the packing technique is applied to data-less responses, it is being used all the time and not only during starvation situations.

7.10 Conclusions

It is known that the main problem of broadcast messages is their lack of scalability. In spite of that fact, Token Coherence must use them to ensure the resolution of all the cache misses by the starvation prevention mechanism. In this chapter we have addressed this problem. In particular, we have proposed

an effective strategy to alleviate the damage caused by broadcast messages. This technique is implemented in the network switches and it is only used when the bandwidth required by broadcast messages exceeds the bandwidth provided by the network. In that case, broadcast messages begin to be accumulated in the input buffers of the network switches. Hence, switches can take advantage of the fact that those broadcast messages are similar enough to compose a single message. In this way, the traffic due to broadcast messages can be drastically reduced. This reduction increases as the number of broadcast messages increases, which can provide certain scalability to the starvation prevention mechanism. As shown in the results, the reduction of the traffic due to broadcast messages affects other class of messages since the overall traffic network is significantly alleviated. As a result, the latency of cache misses is reduced, which in turn can reduce the runtime of applications.

The switch-based packing technique can also be extended to the case of non-silent invalidations. In making so, the number of received data-less responses and their latency reduce, which in turn contributes to improve the latency of cache-to-cache misses.

Chapter 8

Multicast Responses

The owner processor is the one in charge of providing copies of a block to the processors that request it. In case of highly-contended blocks, the owner processor may become a bottleneck. This situation worsens in case of starvation since requests are received near-simultaneously. To address this problem, in this chapter we propose a mechanism that lets the owner processor simultaneously serve several requests by using a single multicast message, instead of multiple individual messages.

8.1 Introduction

Cache coherence protocols are implemented on a specific coherence model such as MOESI or a derived one. Coherence models establish an efficient way to move memory blocks between system components. These models usually designate at each time a single processor which is the only one in charge of providing the value of certain memory block to the processors that request it. This strategy is (1) efficient, because only one processor responds with data to requests, and (2) fast, because the responses from processors are faster than the responses from memory. From now on, we will use the term *owner processor* (or just *owner*) to refer to the processor in charge of providing certain memory block to requesters.

The strategy of using a single owner processor at each time to serve all the requests is efficient and fast when the interval between the reception of the

requests is enough to completely inject the responses in the network. However, when the owner processor has to serve several near-simultaneous requests, it may become a bottleneck. Since the response messages sent by the owner processor include a copy of the requested memory block, they are the largest messages in the system. Therefore, the time required to inject a data response is much higher than the time required to receive a request. Thus, when the owner processor receives several near-simultaneous requests, the data response messages that it sends will accumulate in the output buffers. For example, assuming that a processor can inject/receive 1 byte per cycle, the size of requests are 10 bytes, and the size of data responses are 74 bytes, the owner processor could receive 1 request every 10 cycles, while it could only send 1 data response every 74 cycles. Consequently, the owner processor can receive about 7 requests while it is injecting a single response. This can also cause additional congestion in the network interface controller since the competition for the input buffers of switches can add significant delay to packets. In addition to the performance inefficiencies, the multiple unicast data responses consumes additional power due to redundant information in packets traversing the network since the multiple data responses are likely to hold the same memory block.

The situation described above frequently happens during protocol races. A protocol race is caused when several requests contend for the same memory block and at least one of them requires exclusive access to the memory block, that is, it is a write request[1]. In that situation, several requests will not be completely served:

- The write request fails at getting all the block's tokens.
- Alternatively, some of the other requests that contend for the same block may also fail. These requests are likely to be read requests as they are much more common than write requests.
- The processors that have responded to the write request by sending all the block's tokens and by invalidating the copy of the memory block that

[1]Note that if all processors require shared access (read requests), a protocol race will not occur. However, when one of them requires exclusive access, then a protocol race is likely to happen.

they held in their cache are likely to issue new (read) requests for that block because, according to the temporal locality exploited by caches, it is likely that in a short time those processors want to access to the block and, since they do not have a valid copy in its cache, they will issue a request. However, those requests will probably fail because the system requires a time until detecting and solving the starvation situation.

From this analysis, we infer that a starvation situation will usually entail one write request (which is the initiator of the situation) and several requests (probably read requests). Thus, after completing the write request, the owner processor will have to serve several read requests. Taking into account that processors detect starvation near-simultaneously, the owner processor will receive a lot of consecutive read requests or, even, it can already have stored in its table several read requests. As the owner is in charge of completing all of them, their service will turn the owner processor into a bottleneck. Besides, the data response messages will suffer a high delay, since they will have to wait long until being injected into the network.

To lower the delay suffered by the injected data responses during a starvation situation and take full advantage of the facilities offered by the priority request mechanism, in this chapter we propose the use of *Multicast Data Responses* (MDRs). An MDR is a single data response message which can be sent to simultaneously serve several priority requests. However, note that MDRs are not suitable to serve several persistent requests because at any time only one persistent request can be active and, therefore, only that persistent request can be served. On the contrary, several priority requests can be served because they do not need to be activated and deactivated and, therefore, several priority requests can be served simultaneously. Note also that only the priority read requests (which are much more common than priority write requests) can be served by MDRs. Several priority write requests can not be served at the same time because, after serving the highest priority write request, the node will lack of tokens to continue to serve the next one. On the other hand, the data messages in response to priority **read** requests are all sent by the owner processor and, in addition, all of them include the same information (one token and the memory block), which makes them suitable to form MDRs.

In order a node to generate an MDR, it is necessary that a certain number

destination	source	type	size	address	data	tokens	completed PR
1 byte	1 byte	2 bits	1 byte	4 bytes	64 bytes	10 bits	2 bytes

(a) unicast data response

destination set	source	type	size	address	data	tokens	completed PR
n bytes	1 byte	2 bits	1 byte	4 bytes	64 bytes	10 bits	2 bytes

(b) multicast data response

Figure 8.1: Differences between the formats of (a) unicast data responses (UDRs) and (b) multicast data responses (MDRs).

of consecutive priority read requests are previously stored in the node's table. However, taking into account that nodes proceed to serve read requests as soon as the requested tokens are available, priority read requests may hardly coincide in tables, preventing MDRs from being formed. However, note that the coherence messages serving several consecutive priority read requests will coincide in the NIC output buffers of the owner processor. This is because their transmission is serialized and they are much larger than priority requests. Therefore, the node can take advantage to join all these unicast responses into a single MDR. Mainly, two advantages are derived from the use of MDRs. First, the average latency of responses decreases and, second, the network traffic related to data responses is reduced, which in turn may contribute to alleviate the network overhead.

8.2 MDR Packet Format

Since the responses to consecutive priority read requests are likely to coincide in the NIC output buffers of the owner processors and, in addition, the information held by those messages is very similar, the generation of a single multicast data response from several responses is feasible. To be able to pack several responses into just a single MDR packet, the format of data responses changes slightly. Figure 8.1 illustrates the differences between a unicast data response and a multicast data response. As shown, the unique difference is that MDRs include the *destination set* field instead of the *destination* field.

However, the rest of fields are exactly the same.

In unicast messages, the *destination* field indicates the component that must receive the message. In multicast messages, the *destination set* field refers to the set of components that must receive the message. To implement this field, several options can be chosen:

- The *destination set* field can be implemented as a bit vector with as many bits as number of processors in the system. If so, the size of this field will depend on the system size, but it will be independent of the number of destinations that have to receive the message. Therefore, a bit vector will be a good solution when MDRs include a lot of destinations.

- Alternatively, the *destination set* field can be implemented as a list of processors. In this case, the size of this field will depend on the number of destinations, but it will not depend on the system size. Besides, to identify the number of destinations, the header of the message has to include a field which indicates how many destinations there are in the list. This is a good solution when the number of destinations per message is usually small.

- A third option is to combine the two previous proposals. In particular, if the next expression is true:

$$x * log_2 M > M$$

  (where x represents the number of destinations in the MDR message and M represents the number of processors in the system), then a bit vector should be used. Otherwise, the *destination set* field is implemented as a list of processors. To implement this option, the message requires an additional field (just one bit) to indicate the format that is being used at each time.

In terms of efficiency and size of the messages, the best proposed option is the last one, since it chooses the most suitable format for each situation. However, this option is the most complex to implement. On the other hand, the main advantage of implementing the *destination set* field as a bit vector

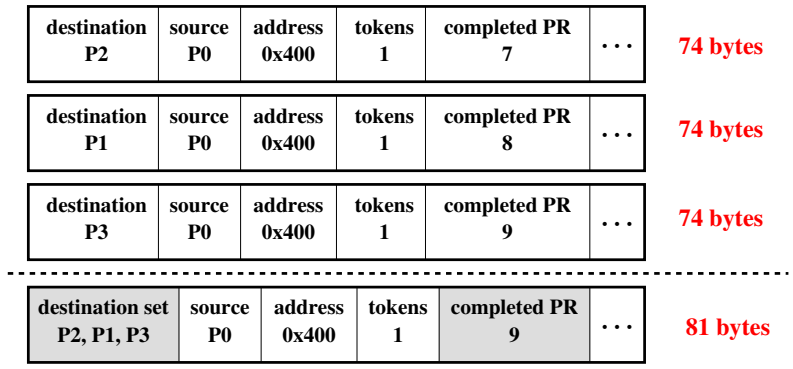


Figure 8.2: Example where three different unicast messages compose a single MDR.

is its simplicity and that is easy to implement. Besides, the size of MDRs is independent of the number of destinations. Due to these reasons, in this work we assume the vector bit implementation.

Besides composing the *destination set* field, when an MDR is generated, the *completed PR* field also receives a special treatment. Note that the several unicast messages included in an MDR will have different values in their *completed PR* field. Therefore, we have to choose which is the proper value for the final MDR message. However, in this case, the value of this field can be easily determined. The *completed PR* field of data responses indicates that all the priority requests (for the held block) with an identifier equal to or smaller than it were completely served. Since priority requests are completely served in order, the value of the *completed PR* field of the last included data response will be the highest value and it will implicitly include the other values. Therefore, the value of the *completed PR* field of an MDR message is the value of the *completed PR* field belonging to last data response included in it.

Figure 8.2 walks through an example where a single MDR is composed from three consecutive data responses. As shown in the figure, *P0* sends three data responses to complete the *P2*'s priority request, the *P1*'s request, and the *P3*'s request. These messages are similar and only the *destination* and *completed PR* fields differ. To compose an MDR message from these three unicast messages, the *destination set* field codes the values of the corresponding *destination* fields and the *completed PR* field includes the value of that field belonging to the last

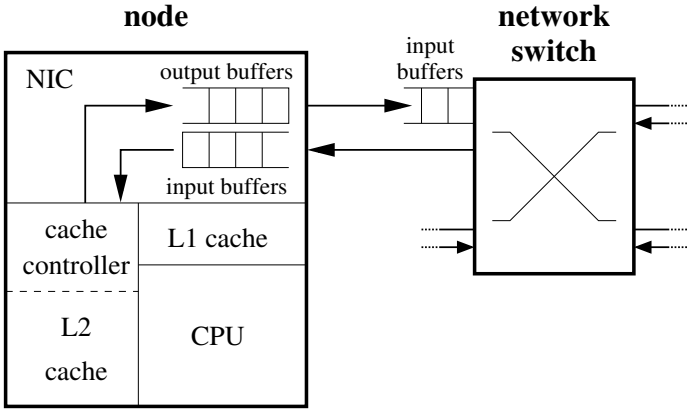


Figure 8.3: Node structure.

included message. By doing so, it is like if *P0* only had sent a single message whose total size is 81 bytes[2] instead of sending three different messages with 222 bytes of total size.

8.3 MDR Generation

As commented before, to take full advantage of priority requests, the output buffer of nodes can be instrumented to easily pack several data responses into a single MDR. Unlike the proposal in Chapter 7, in this case the packing is performed in the nodes themselves, instead of in the network switches. To implement this technique, we assume an SMP environment where each node is modeled as a CPU and input and output buffers are connected to a network switch, such as that shown in Figure 8.3. When a processor sends a message, the outgoing message is stored in the output buffer of the NIC. The messages stored in the output buffers are sent one by one applying a FIFO scheme to the input buffer of the switch which the node is connected to. Thus, if a processor generates many data responses in a short time, they will accumulate in the output buffers of the node. Therefore, those buffers are the natural place to compose MDRs.

To be able to form MDRs, processors implement the following strategy:

[2]In this example, we assume that the *destination set* field is implemented as a bit vector.

1. When a processor has to serve a priority read request, before generating a new data response, it checks whether its output buffer contains a data response where the new response could be included.

2. If there exists such a message, the new destination processor is added to the *destination set* field of the stored message and the information about the completed priority requests (*completed PR* field) is updated accordingly.

3. If the processor does not find a suitable message to compose a single MDR, a new MDR with only one destination is generated and inserted into the output buffer.

4. When an MDR reaches the head of the output buffer and the input buffer of the switch have enough room to store the message, the message is injected into the network by copying it in the switch buffer.

Note that, according to the defined scheme, data responses do not have to wait for forming an MDR (which would delay their transmissions) since they are transmitted through the network as soon as possible. Indeed, what we propose is to take advantage of the elapsed time that a certain data response has to wait for being transmitted to compound a single data response by adding to it the destinations of the data responses that would have been generated during that time.

To extract the maximum benefit from this strategy, we need that the data responses coincide in the output buffers of processors. In fact, the more data responses coincide, the more advantage we will be able to take of it. As shown in Figure 8.3, each processor is connected to a network switch. Thus, when a message reaches the head of the output buffer and the input buffers of the switch have enough room to store that message, it is transmitted from the output buffers of the processor to the input buffer of the switch and its transmission through the interconnect begins. Note that, if the input buffers of the switches can hold a lot of messages, then the data responses are likely to coincide in the switches, which may prevent the generation of certain MDRs. We are not in favor of implementing this technique in the switches because it would entail to make much more complex the logic of the network switches

and switches do not have as much computational power as a processor or a NIC [30, 85]. Therefore, the number of messages coinciding at the output buffers should be maximized. To this end, we can reduce the storage space of the input buffers of switches to just two MDRs. Making so, we force MDRs to coincide in the output buffers of the switches, which increases the opportunities to generate a single MDR. Besides, by limiting the space to just two MDRs, we ensure that the injection of the MDR into the network will not be delayed by the fact of reducing the size of the input buffers. Note that the assumption of two MDRs is in accordance with parameters and the flow control assumed in this work to prevent buffer inefficiency and bubbles.

8.4 Searching for MDRs at NIC Output Buffers

The search performed by processors to check if there exists an MDR where a new response can be included must be simple. Otherwise, it could affect the critical path of data responses, delaying their generation and slowing down the system. To this end, we assume the following considerations:

- Responses use a dedicated virtual channel. Therefore, the search is only performed in the output buffer associated to that virtual channel, instead of through all the output buffers of the link.

- A *present* register (1 bit) can be used to indicate if the output buffer contains an MDR where a new response could be added or not. This register can be used to avoid the search when the output buffer does not contain any MDR or it only contains an MDR placed at the head of the buffer[3]. The bit will be set the first time an MDR is inserted in the output buffers and it will be unset when the last inserted MDR is removed.

When the owner processor has to send a data response, it checks if the *present* register is set. If so, the processor begins the search by checking the type of the message stored in second place in the output buffer. If the message

[3]Note that the message in the head of the buffer can not be used to compose an MDR because its injection into the network could occur while the packing is carried out.

type is a data response, then it checks if the memory address of the held data matches the memory address of the data that it has to send. In such a case, the new destination processor is added to the *destination set* field of the stored message and the *completed PR* field is updated accordingly. If the data do not match, then it continues the search with the third message. It continues until finding a suitable MDR or checking all the messages in the buffer dedicated to responses.

Note that, according to the defined strategy, we do not check if the first message in the output buffers is suitable to compose an MDR. This is because the first message can be injected in the network at any time and, if we are packing the message, its injection could be delayed. Thus, if we avoid packing the first message, the critical path of data responses is not affected.

8.5 Packing Process

When the owner processor has to send a new data response to a requester and it finds a suitable MDR in its output buffer, it proceeds to pack the two messages. The packing consists in adding the new destination to the *destination set* field and updating the *completed PR* with the latest value. The process of adding a new element to the *destination set* field can be performed in different ways which depend on how the *destination set* field is implemented (see Section 8.2). Since we will assume in the evaluation that the *destination set* field is implemented as a bit vector, we only describe the packing process for this implementation option. One of the advantages of implementing the *destination set* field like a bit vector is that it is easy to add a new element. The vector will have as many bits as number of possible destinations. The bit X within the bit vector is set to 0 if the processor X does not have to receive the message. The bit X within the bit vector is set to 1 when the processor X does have to receive the message. Therefore, to add a new element to the vector simply an OR operation is performed:

$$destinationset = destinationset\, OR\, (1 << X)$$

where X is the destination processor, OR is the logical OR operation, and $<<$ is the displacement operation.

8.6 Adjusting the Starvation Detection Timeout

As we commented in Section 7.7, if the value of the timeout is estimated under certain circumstances, its value will only be suitable when the system moves under those circumstances. Therefore, a problem can occur when we use MDRs because the timeout intervals are mainly calculated assuming unicast data responses. Notice that, when a protocol race happens and we use MDRs, we are using messages whose size can be considerably different from that of unicast data responses. As a result, the timeout interval calculated using unicast responses (or MDRs with a single destination) may not be suitable when we use MDRs with several destinations. To improve this aspect, one of the alternatives previously proposed in Section 7.7 should be used:

- The simplest solution is to use a higher timeout interval in all cases, for example, *three* times the processor's average miss latency. This is the solution that we assume in the evaluation due to its simplicity and efficiency.

- Another choice is to use different timeouts based on the elapsed time from the reception of the last priority request.

- The last option is to adjust the timeout as long as new priority requests are received.

8.7 Evaluation

In what follows, we analyze how the use of MDRs contributes to improve the performance of priority requests and, in turn, the performance of the whole protocol. *UDRs* stands for the protocol using only unicast data responses. *MDRs t2* stands for the protocol using both unicast and multicast data responses. Besides, like *UDRs*, *MDRs t2* uses a timeout set to twice the average miss latency. *MDRs t3* stands for the protocol using both unicast and multicast data responses and a timeout set to three times the average miss latency.

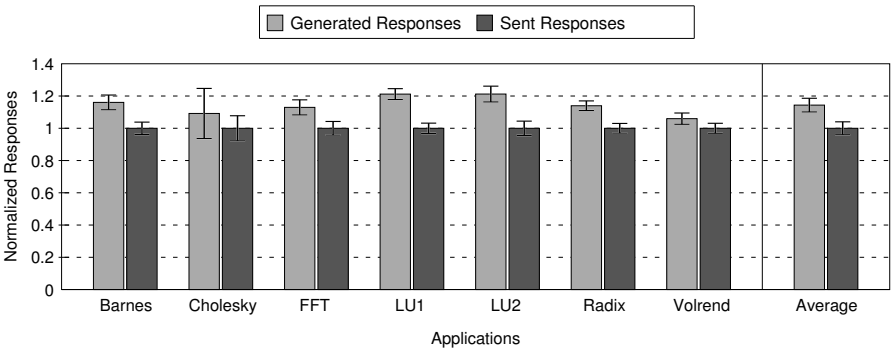


Figure 8.4: Normalized number of data responses: generated responses against injected responses (*MDRs t2*).

8.7.1 Target System and Parameters

We simulate a 32-processor Sparc v9 system with the parameters commented in Chapter 4. We evaluate the referred processor system assuming that processors are connected through an MIN network with the perfect-shuffle permutation. The results for the 2D mesh are not shown because they are similar to those obtained for the MIN. The simulated starvation prevention mechanism assume only one simultaneous outstanding priority request per processor.

All the results shown in the following figures are normalized to the value obtained for *MDRs t3*.

8.7.2 Generated vs Injected Data Responses

Figure 8.4 shows the comparison between the number of responses generated by components and the number of responses transmitted through the network (both in term of packets). Due to the fact that a single MDR can include several responses to various components, the number of sent responses is lower than the number of generated responses. In a 32-processor system and for the applications used in this case, we observe that the use of MDRs can entail a reduction of about 15% in average in the number of data responses. This result is important because the data responses messages are the biggest messages in the system, being their size noticeably larger than that of the other kind of messages (74 bytes against 10 bytes). This reduction can contribute to allevi-

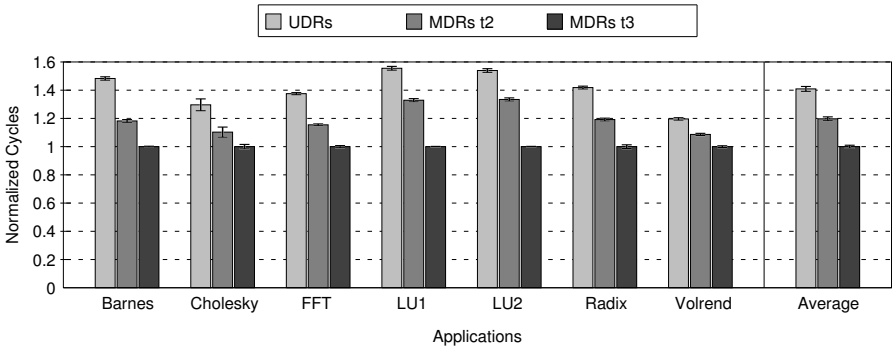


Figure 8.5: Normalized average latency (in cycles) of UDRs and MDRs.

ate both the input buffer of the system components and also the interconnect traffic.

8.7.3 Latency of Data Responses

Figure 8.5 illustrates the latency of data responses. This latency includes the elapsed time from a response is enqueued in the output buffers until that response reaches its destination (or the last destination in case of an MDR). As shown, the latency of data responses decreases about 20% in average when using MDRs together with a bad timeout. However, this reduction can become more significant when using MDRs with a suitable timeout in which case a reduction of 40% can be reached. This reduction is obtained thanks to two main reasons. First, when MDRs are used, the number of messages stored in the output buffers reduces, making them less collapsed. This contributes to reduce the time required to inject a message in the interconnection network. Second, as the traffic going throughout the interconnect decreases (which will be shown later), messages suffer less contention and their latency reduces.

As shown in the figure, the fact of using an appropriate timeout also affects the latency of data responses. This is because a suitable timeout reduces the generated traffic. This reduction of network traffic alleviates the congestion in the network, which in turn contributes to reduce the latency of all kind of messages.

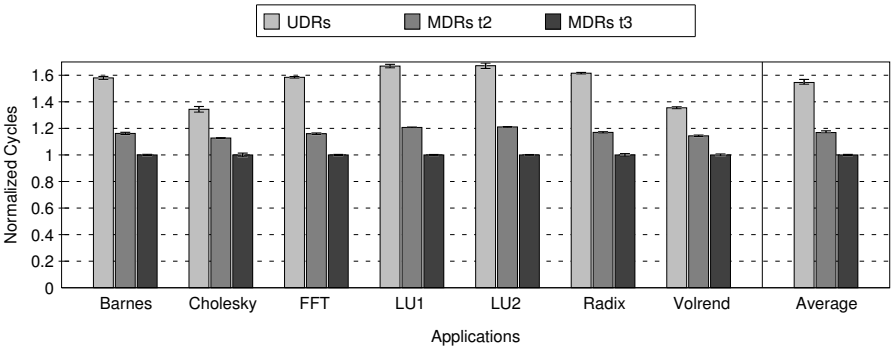


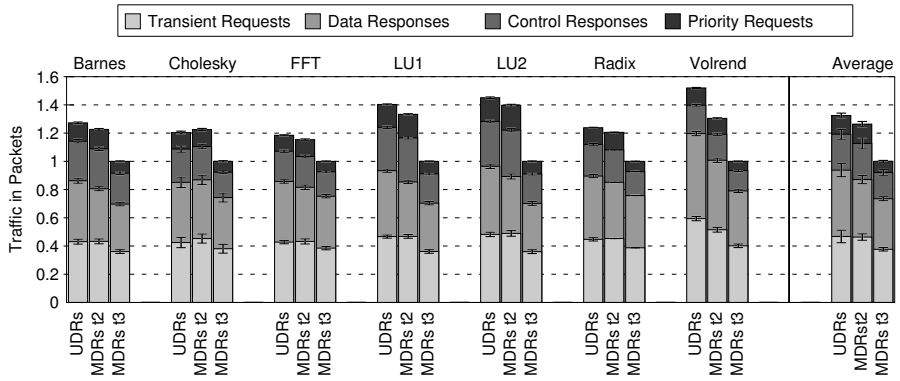
Figure 8.6: Normalized latency (in cycles) of solving starvation when using UDRs and MDRs.

8.7.4 Starvation Latency

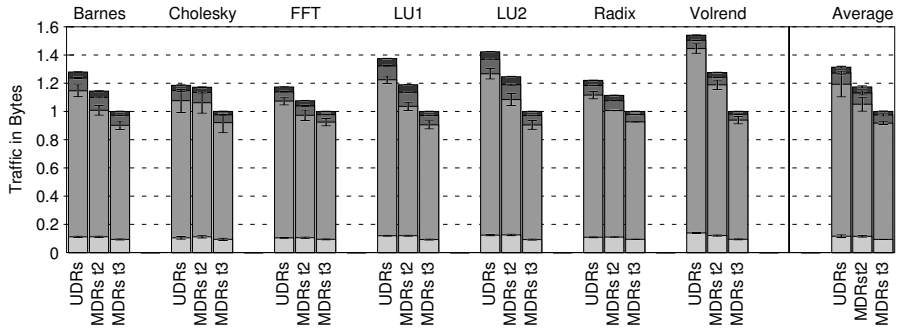
Figure 8.6 depicts the average time of solving starvation situations. Due to the fact that MDRs cause the latency of data responses to reduce and, additionally, the network traffic due to data responses reduces, the average latency required to solve the starvation situations lowers. This is shown in the figure, where *MDRs t2* gets a significant reduction of the starvation latency. In case of *MDRs t3*, the latency reduces even more, reaching more than 50% of reduction with respect to *UDRs*.

8.7.5 Network Traffic

Figure 8.7 shows the generated network traffic. As we can see, when using MDRs with a timeout which is not totally suitable, the total network traffic generated does not differ a lot from that generated when using UDRs. This happens because, although the number of data responses decreases, the number of *Transient Req*, *Control Resp*, and *Starvation Ctrl* messages rises slightly. This increase is caused by the influence of a bad timeout. A non-suitable timeout entails that some nodes detect false starvation. For example, a message may be delayed in the network due to congestion or other reasons. This false starvation causes the generation of *Starvation Ctrl* messages, which in turn will require the generation of *Data Resp* and *Control Resp*. Therefore, the memory blocks stay longer in the interconnect (since components forward them in response to priority requests to solve false starvation). As a result,



(a) Traffic in Packets



(b) Traffic in Bytes

Figure 8.7: Normalized traffic (a) in packets and (b) in bytes when using UDRs and MDRs.

the number of cache misses increases, thereby rising the number of *Transient Req.* All these problems can be solved by using MDRs together with a suitable timeout. Thus, you can observe in the figure that *MDRs t3* noticeably reduces the generated network traffic, decreasing it in more than 30%.

8.7.6 Runtime

Finally, Figure 8.8 shows the runtime of different applications when using UDRs and MDRs. As shown, when using *MDRs t2* the runtime of the applications is reduced about 6% in average, mainly because of the reduction in the latency of data responses. Besides, when using a suitable timeout, the

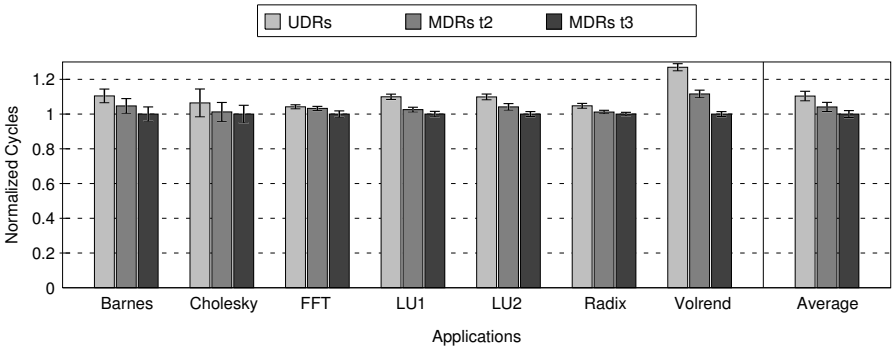


Figure 8.8: Normalized runtime when using UDRs and MDRs.

runtime of applications diminishes even more, reaching 10% of reduction in average. This is because not only the latency of data responses decreases, but the generated traffic also reduces considerably. As a result, applications can be executed more quickly.

8.8 Conclusions

In this chapter we have proposed the use of multicast routing techniques to enhance token-based protocols, focusing on the starvation prevention mechanism. In this case, multicast routing techniques can be applied thanks to the flexibility provided by an efficient starvation prevention mechanism such as priority requests. Since this mechanism lets the processors serve simultaneously several priority requests, during a starvation situation the NIC output buffers of the owner processor can suffer congestion. By using multicast data responses, the number of messages (and also the number of bytes) queued in the output buffers can be drastically reduced. Additionally, the overall network traffic and the average latency of data responses reduces considerably, yielding to a reduction in the runtime of applications.

Chapter 9

Conclusions

Cache coherence protocols are a key component to achieve the goal of providing a performance growth proportional to the system size to current shared-memory multiprocessor systems, which incorporate more and more processors and cores (CMPs). This dissertation makes several contributions in the space of token-based cache coherence protocols applied to a CC-NUMA environment. In this chapter (Section 9.1), we first summarize the contributions. Then in Section 9.2, we offer some conclusions and reflections based on this research. In Section 9.3 we show the publications derived from the works carried out in the dissertation. Finally, in Section 9.4 we discuss about the future work.

9.1 Contributions

This dissertation addresses five different problem areas related to token-based cache coherence protocols with the goal of providing greater scalability. In particular, we have focused on the Token Coherence protocol.

First, we recognized that order can be provided to requests without the need of using totally-ordered interconnects nor directories. To this end, we rely on the routing algorithm, providing a total order when required. In this sense, we contributed a new starvation prevention mechanism named priority requests that exploits the order provided by the routing algorithm. In doing so, we have shown how the new starvation prevention mechanism performs much better and provides higher scalability.

Second, to avoid the utilization of arbiters (which would generate indirection), the starvation prevention mechanism requires a table at each system component which grows proportionally to two factors: the number of processors in the system and the number of simultaneous outstanding requests per processor. Although the number of outstanding requests per processor can be limited at the expense of a performance degradation, those tables were still proportional to the number of processors, which jeopardizes the scalability of Token Coherence in terms of storage requirements. We contributed a simple and effective strategy that decouples both the number of processors and the number of simultaneous outstanding requests per processor from the table size at the expense of a slight performance degradation. This is possible thanks to the required information is appropriately distributed through all nodes instead of replicated. Besides, the table size is totally independent of the number of processors in the system.

Third, although token-based protocols can use different types of requests (broadcast, multicast[1], point-to-point[2]) to quickly resolve cache misses, the starvation prevention mechanisms require to use requests based on broadcast messages. It is known that broadcast messages entail a serious problem in medium and large systems, due to their lack of scalability. We contributed a switch-based technique to pack several broadcast messages into just a single message. Making so, broadcast messages are endowed with certain scalability, easing their use in medium and large systems. The effectiveness of this technique is much higher when it is applied to priority requests because, thanks to the utilization of ordered paths, priority requests are likely to coincide in the buffer of the traversed switches, increasing packing opportunities.

Forth, we contributed the concept of multicast data responses. After writing a memory block, the writer will have to provide a copy to all the processors that want to read it. If it is a highly-contended block, after the write, a lot of processors will want to read it. Providing a copy one by one is too slow and, due to the size of data responses, the writer will easily become a bottleneck. Thus, by using a multicast data response, several copies of a memory block

[1]This type of message requires a prediction which may fail.

[2]The point-to-point messages are used in a directory-like approximation which presents the problem of indirection.

can be simultaneously validated in different processors. This contributes to reduce the cache miss latency of highly-contended memory blocks. This kind of message is only applicable when using priority requests, since persistent requests must be solved one by one, not allowing to simultaneously serve several persistent requests.

Fifth, we detected another serious problem of token-based protocols: the non-silent invalidations. When a memory block is invalidated because of the reception of a write request, the recipient sends all the tokens it holds to the writer. The writer can not perform the memory operation until having received all the tokens, that is, the confirmation from all the sharers indicating they have invalidated their copy. Although these data-less token messages are not very large, in medium and large systems they may flood the network. We contribute a switch-based technique that reduces the problem of non-silent invalidations. By packing several invalidations into just one, the latency of the write cache misses can be reduced. Unlike previous proposals, this technique can be applied independently of the starvation prevention mechanism.

9.2 Conclusions

In this section, we draw some conclusions and also reflect on the dissertation research with the benefit of hindsight and the freedom to make statements of opinion.

Thanks to the global order provided by ordered paths, priority requests can easily resolve starvation without the need of overriding the performance policy and without using explicit acknowledgments. As a result, an efficient performance policy is applied all the time, which causes the total network traffic to lower considerably. This reduction in traffic is more significant as the system size increases, being able to save up to 40% of the total traffic generated when persistent requests are used. Although the traffic reduction does not entail a strong decrease in the applications runtime (30% in average in a 64-processor system), it alleviates the congestion in the interconnect. This fact may be really important because, thanks to it, a greater number of applications could be running simultaneously in the system before reaching the saturation point of the network. The benefits obtained by the applica-

tion of priority requests grow proportionally to the system size, which shows their scalability. These results have been observed regardless of the network topology and the executed applications. Besides, thanks to the flexibility provided by the priority requests, we can apply multicast routing techniques to data responses, which increase the benefits of priority requests. The use of multicast data responses alleviates the congestion at the NIC buffers, which in turn contributes to reduce their latency in about 40%. In addition, the use of multicast data responses causes the priority request mechanism to increase the network traffic reduction, which improves the runtime of application in about 10%. Therefore, thanks to the flexibility of priority requests, multicast data responses can be used, which multiply the benefits provided by priority requests, reaching about 30% of reduction in the runtime of application with respect to persistent requests.

It is important to highlight that all the advantages provided by priority requests are achieved without increasing the hardware requirements of the most efficient implementation of persistent requests. Furthermore, the use of ordered requests can additionally help us to decrease these hardware requirements without losing effectiveness. In particular, the technique proposed to limit the storage requirements of the starvation prevention mechanisms drastically minimizes them while still performing well. Thus, although the priority request tables at nodes can be reduced to the minimum of one entry, the total network traffic generated remains more or less constant. Hence, in spite of using tables 32 times smaller (assuming a 32-processor system), the priority request mechanism still reduces about 25% the total traffic generated when persistent requests are used. On the other hand, the reduction of the table size causes the runtime to slightly increase. However, the runtime continue being significantly smaller (up to 18%) than that obtained when using persistent requests.

Unlike the priority request proposal, the proposed switch-based packing technique does increase the hardware requirements of the system, in particular the hardware requirements of network switches. Since we have not evaluated this rise at VLSI level, the packing technique has been limited just to certain cases to simplify the technique and to avoid a huge increase in complexity. According to the obtained results, it seems that the advantages provided by

the packing technique could offset the additional hardware complexity. We have only applied the packing technique to priority requests because, unlike persistent requests, they offer clear opportunities for the request packing. The packing of priority requests into a single message is an important contribution because it solves the congestion caused by broadcast messages in medium and large system. Thus, the packing technique reduces above 40% the endpoint traffic generated by priority requests. This reduction not only contributes to improve the latency of priority requests, but it also contributes to improve the average latency of all kind of traffic. This may result into a decrement in the runtime of applications, reaching a reduction above 10% in average (in a 64-processor system). Moreover, the packing technique can be also applied to control responses, obtaining similar results. This allows us to address the problem of the non-silent invalidations. Thus, the packing technique can help us to reduce the latency of write misses, which also affects the runtime of applications, reducing it about 10% in average. Note that the packing technique over control responses can be applied independently of the use of persistent or priority requests.

From this analysis, we can see the benefits of using ordered requests. Nevertheless, some works such as [76] argue that the protocols that rely on total order of requests are undesirable due to the higher latency and cost. We think that this is true only if the order is provided by totally-ordered interconnects because, as shown throughout this dissertation, a global order can be alternatively provided without using high-latency or high-cost interconnects. The routing algorithm is enough to provide such a global order. Although both approximations may seem similar, they are not. In fact, we are going to analyze the main arguments against totally-ordered interconnects and we will be able to see how our proposal avoids them:

- *"In systems that rely on a total order of requests, requests (and sometimes responses) for different addresses must be kept in order with respect to each other throughout the system"*. One major advantage of ordering requests by the routing algorithm instead of by the interconnect is that we can decide when we want to use the algorithm to put requests in order and when we do not want. Thus, the system can use only ordered paths when required, thereby avoiding the cited problem. Besides, we

have shown different methods to provide a total order only to requests for the same memory address, not having to put in order requests upon different memory addresses. In addition, responses can use a different routing algorithm and, therefore, they do not need to be ordered.

- *"On a total order of requests, the internal pathways and coherence controllers are perhaps more difficult to bank (in the same manner in which memory and caches are banked for increased bandwidth)"*. Since the order is provided by the routing algorithm and not by the interconnect, the physical location of memory banks and caches is not restricted by the interconnect. Therefore, the use of ordered paths does not complicate the system design.

- *"Cost and latency of the interconnect"*. Since the interconnect is not limited to any particular type, point-to-point interconnects can be used. With regard to the latency, only the requests that require to be ordered will slightly have a higher latency. However, in spite of this higher latency, the average latency of cache misses decreases because the benefits that the ordered requests provide offset the small increase in their latency.

As shown, the approximations to provide global order to requests by totally-ordered interconnects or by ordered paths are completely different. Besides, this approach is also different from those approaches that put requests in order by using directories. On the one hand, on directory-based protocols, requests are sent to the directory. The directory receives them, processes them, and then it decides if they must be forwarded. This is slow because the directory is usually located at the main memory, which makes it slow. On the other hand, priority requests are sent through an ordered path. To this end, they are sent through certain switch selected as a root. This is very different since the root switch does not process or consume the requests. In fact, the single difference of sending a request through an ordered path is that the request does not follow a minimal path, which slightly increases the average latency in some cases. Therefore, unlike directory-based protocols, we can not consider that the ordered paths generate indirection.

We conclude that the routing algorithms based on ordered paths can be applied to obtain a global order being able to avoid the problems caused by totally-ordered interconnects or directories. Thus, the application of such algorithms can help to simplify token-based protocols, provide greater scalability, reduce the overall network traffic, and reduce the runtime of applications.

9.3 Scientific Publications

The work presented in this dissertation has been published in the conferences and papers listed below, having received useful feedback from the reviewers.

- B. Cuesta, A. Robles, and J. Duato. *"Enhancing the Starvation Prevention Mechanism of Token Coherence"*. Submitted to IEEE Transactions on Computers.

- B. Cuesta, A. Robles, and J. Duato. *"Switch-Based Packing Technique for Improving Token Coherence Scalability"*. 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies. Dunedin (New Zealand). December 2008. ISBN 978-0-7695-3443-5.

- B. Cuesta, A. Robles, and J. Duato. *"Switch-Based Packing Technique for Improving Token Coherence Scalability"*. XIX Jornadas de Paralelismo. Castellon (Spain). September 2008. ISBN 978-84-8021-676-0.

- B. Cuesta, A. Robles, and J. Duato. *"Improving Token Coherence by Multicast Coherence Messages"*. 14th Euromicro International Conference on Parallel, Distributed and Network-Based Processing. Toulouse (France). February 2008. ISBN 978-0-7695-3089-5.

- B. Cuesta, A. Robles, and J. Duato. *"Enhancing starvation prevention mechanism for Token Coherence"*. XVIII Jornadas de Paralelismo. Zaragoza (Spain). September 2007. ISBN 978-84-9732-593-6.

- B. Cuesta, A. Robles, and J. Duato. *"An effective starvation avoidance mechanism to enhance the Token Coherence protocol"*. 15th Euromicro

International Conference on Parallel, Distributed and Network-Based Processing. Naples (Italy). February 2007. ISBN 978-0-7695-2784-0.

- B. Cuesta, A. Robles, and J. Duato. *"Enhancing Token Coherence by Using a New Starvation Avoidance Mechanism"*. XVII Jornadas de Paralelismo. Albacete (Spain). September 2006. ISBN 84-690-0551-0.

- B. Cuesta, A. Robles, and J. Duato. *"Enhancing Token Coherence by Using a New Starvation Avoidance Mechanism"*. 2006 Advanced Computer Architectures and Compilation for Embedded Systems. L'Aquila (Italy). June 2006. ISBN 90-382-0981-9.

9.4 Future Work

Although this dissertation has mostly focused on scalability and performance characteristics, we believe that the aforementioned contributions may have other desirable attributes that were not fully explored in this dissertation. Thus, in this section we comment some of the future work lines.

A key aspect of current systems is the power consumption [88, 60, 59]. It is known that systems incorporate more and more processors (or cores in case of CMPs) and, therefore, their power consumption and dissipation should be as small as possible. Although we have not explored these aspects in the dissertation, it would be interesting to do it as a future work. As we have seen throughout the dissertation, the use of the priority requests and the rest of proposals substantially reduces the generated network traffic. This reduction can also entail a reduction of power consumption in the network. However, the reduction of the traffic may not only affect the power consumption of the interconnect, but it may also affect to other aspects. Thus, for example, when a message is sent or received, the components have to access to different structures (tables, caches, tags, ..). By reducing the number of messages, we are also reducing the number of accesses to those structures, which in turn may contribute to reduce the consumption at nodes and memories. Besides, we have proposed a technique to reduce the size of the tables that store the starved requests. Reducing their size also implies a power reduction and, additionally, a reduction in the access latency. Hence, the contributions of this

dissertation may have another desirable attribute that has not been explored here and which should be done in the future.

Apart from power consumption matters, there are other aspects of the different proposals that could make them more attractive and which have not been fully explored in the dissertation. Following we shortly describe some of them:

- *Table reduction.* Throughout the dissertation we have assumed 3 virtual channels for performing a fair comparison with previous proposals. The traffic is assigned to a specific virtual channel depending on its type. Thus, for example, point-to-point messages (which are usually responses) use the virtual channel 0, adaptive broadcast messages (which are transient requests) use the virtual channel 1, and ordered broadcast messages (which are priority requests) use the virtual channel 2. Each virtual channel has certain priority, for example, the virtual channel 2 has higher priority than virtual channel 1, and virtual channel 1 has higher priority than virtual channel 0. This priority scheme and the way of distributing the traffic can affect the latency of messages and, therefore, it can also affect the cache miss latency. In particular, changing the priority scheme or adding new virtual channels (for instance, a dedicated virtual channel for resending notifications) may influence the protocol performance. Thus, it would be interesting to perform a study of how the traffic should be distributed among virtual channels and the most suitable priority schemes to apply. This has special interest in the proposal of the table reduction because this proposal introduces of a new class of message: the resending notifications. Since this kind of message is a point-to-point message, they use the same virtual channel as the responses. However, that virtual channel is the one with the lowest priority and transmitting resending notifications through it may generate inefficiency. Besides, some alternatives have been proposed, such as the removing of the resending notifications, which have not been fully explored and, although they do not perform as well as the studied one, some scenarios could favor them.

- *Switch-based Packing Technique.* The effectiveness of this technique in-

creases when the number of the priority requests that coincide in the network switches increases. Therefore, several alternatives could be implemented to increase the number of priority requests that coincide in switches. For example, for obtaining the results shown in this dissertation we have assumed 2x2 network switches in the MIN interconnect. Thus, each switch has 4 links or ports and each of them supports 3 virtual channels. The arbiters of the switches decide which the message of all the virtual channels is routed at any time. By using different arbitration policies, we can favor the accumulation of certain class of messages. Hence, a thorough study should be done to see how affect the election of a specific policy in the effectiveness of the packing technique.

- *Multicast Data Responses.* According to this proposal a single message can be used to serve several priority requests. Since priority requests are completely served in order, a multicast data response can complete several consecutive priority read requests. If a table contains a lot of priority read requests but they are not consecutive (that is, there are some priority write requests between all the priority read requests), all the priority read requests will not be able to be served by a single message. Thus, to increase the packing opportunities, the policy could be slightly modified, serving first, the priority write requests and, then, all the priority read requests by a single message. To do this, the information attached in responses will have to be modified and the new policy should ensure that the priority write requests will not starve out the priority read requests.

Given that priority requests handle tokens and memory blocks as efficiently as transient requests and they let Token Coherence preserve its flexibility and main characteristics, we think that priority requests could successfully replace transient requests. In making so, the necessity of a starvation prevention mechanism is removed since the completion of priority requests is guaranteed. Besides, in systems where the number of protocol races is considerable, the performance of the protocol could improve considerably. For example, when a transient request fails the penalization in runtime is high, since nodes need to wait for a time (twice the processor's average miss latency) until realizing

of such a situation and then the request is resent (as a priority request). However, if we use a priority request instead of a transient request, the cache miss latency in that case would be much lower. Thus, the timeout used to assume starvation could be eliminated and the resending of requests could be also removed.

Although replacing transient requests by priority requests may have some advantages, it has one minor disadvantage and one major disadvantage. The minor one is that the average latency of requests raises. This is a minor disadvantage because the increase of the latency may be offset by the impossibility of generating protocol races. The major disadvantage is that, whereas the implementations of transient requests can be based on different types of messages (broadcast, multicast, or even point-to-point messages), priority requests must be implemented like broadcast messages. This can entail a serious problem in large and medium systems. Although in Chapter 7 we contributed a mechanism to improve this aspect of broadcast messages, it could be said that this technique is only effective when processors contend for the same memory block, but, in the absence of contention, the effectiveness falls. To address this problem, we have a new solution in mind. The general idea is the following. If we could store some information about the priority requests, due to the fact that their completion is ensured, we could predict with certainty the processors that share certain block. Therefore, instead of broadcasting priority requests, they would only need to be sent to the set of components that share that block. In making so, priority requests could be implemented as multicast messages instead of broadcast messages, which solves the problem of their scalability. To implement this solution without introducing any disadvantages, several aspects should be studied:

- The first question is where to locate the information about the priority requests. The first option would be to put the information in main memory (like a directory), but we would inherit the typical problems from directory-based protocols, such as indirection. Therefore, we discard this possibility. The second option is to place the information in the processor caches. However, in this case, only the local processor could have direct access to that information, whereas the rest of processors would have to access to it through another request, which has the same problem

as directory protocols. Therefore, we discard this possibility too. The third option is to place the information in the network switches. Due to the fact that all the priority requests for the same block are routed through the same root switch, if the information about the sharers is stored in the root switch, it is sure that all the priority requests will be able to use that information to decide which of all the destinations really need to receive the requests and which does not need to. In this way, processors would broadcast priority requests, but when they go through the root switch they could become multicast priority requests.

- The second decision to implement this proposal is to decide what information about priority requests is kept. Since switches are simple components, we can not store the information about all the memory blocks in the network switches because (1) the storage requirements would be too high and (2) the access latency to that table would significantly delay the transmission of each priority request. Therefore, an alternative decision would be to store only the information about some memory blocks, for example, about the most recently accessed ones. In this way, root switches could have like a cache to store the list of processors that share a block. The information in this cache can be used to discard the unnecessary destinations of certain broadcast messages.

- Another important aspect of this proposal is the following. Each priority request is assigned a priority at its arrival at nodes. As all nodes receive all the priority requests and they are received in the same order, each priority request is assigned the same priority. However, according to the new proposal, not all the nodes would receive all the priority requests. Therefore, to continue assigning the same priority to each request, the priority would have to be generated in the root switch. Since this switch knows the order in which they are going to be served, they can assign the priority.

This proposal would be feasible because there are other works, such as [43], which have already proposed to put some information in networks switches.

This dissertation has been developed assuming a traditional shared-memory multiprocessor system. Over the last years, a new class of system has come

out: the CMPs. The special features of CMPs cause the traditional cache coherence protocols to suffer some changes. Thus, due to the increasingly use of CMPs, it would be interesting to adapt the contributions detailed through the dissertation to the CMPs environments. Although the general idea and the motivation of each contribution would basically be the same, they may require certain modifications to fulfill the more challenge requirements regarding power consumption and silicon area existing in CMP environments. In this sense, the proposal made in this dissertation can be useful in CMPs due to its capability for reducing significantly storage requirements and the overall network traffic.

Bibliography

[1] http://download.intel.com/design/processor/manuals/253668.pdf.

[2] Technical documentation of pa-7100lc processor. *Hewlett-Packard*, http://ftp.parisc-linux.org/docs/chips/PCXL_ers.pdf.

[3] Virtutech AB. http://www.virtutech.com/.

[4] D. Abts, D. J. Lilja, and S. Scott. So many states, so little time: Verifying memory coherence in the cray x1. *IPDPS 03: 17th International Parallel and Distributed Processing Symposium*, April 2003.

[5] Manuel E. Acacio, José González, José M. García, and José Duato. Owner prediction for accelerating cache-to-cache transfer misses in a cc-numa architecture. *Supercomputing '02: ACM/IEEE conference on Supercomputing*, pages 1–12, 2002.

[6] Manuel E. Acacio, José González, José M. García, and José Duato. The use of prediction for accelerating upgrade misses in cc-numa multiprocessors. *PACT '02: International Conference on Parallel Architectures and Compilation Techniques*, pages 155–164, 2002.

[7] Manuel E. Acacio, Jose Gonzalez, Jose M. Garcia, and Jose Duato. A two-level directory architecture for highly scalable cc-numa multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 16(1):67–79, 2005.

[8] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The mit alewife

machine: Architecture and performance. *22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[9] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. *SIGARCH Computer Architecture News*, 16(2):280–298, 1988.

[10] Niket Agarwal, Li-Shiuan Peh, and Niraj K. Jha. In-network snoop ordering (inso): Snoopy coherence on unordered interconnects. *HPCA '09: IEEE 15th International Symposium on High Performance Computer Architecture*, pages 67–78, February 2009.

[11] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. *ISCA '00: 27th Annual International Symposium on Computer Architecture*, pages 248–259, 2000.

[12] R. Ejaz Ahmed. Energy-aware cache coherence protocol for chip-multiprocessors. *Electrical and Computer Engineering*, pages 82–85, May 2006.

[13] Alaa R. Alameldeen, Milo M. K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Mark D. Hill, David A. Wood, and Daniel J. Sorin. Simulating a $2mcommercialserverona$2k pc. *Computer*, 36(2):50–57, 2003.

[14] Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. *HPCA '03: nineth International Symposium on High-Performance Computer Architecture*, page 7, 2003.

[15] Alaa R. Alameldeen and David A. Wood. Ipc considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, 2006.

[16] J. Archibald and J.L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[17] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John

Shalf, Samuel W. Williams, and Katherine A. Yelick. The landscape of parallel computing research: a view from berkeley. *Electrical Engineering and Computer Sciences, University of California at Berkeley*, (UCB/EECS-2006-183), December 2006.

[18] E. Atoofian and A. Baniasadi. A power-aware prediction-based cache coherence protocol for chip multiprocessors. *IPDPS '07: IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007.

[19] M. Azimi, F. Briggs, M. Cekleov, M. Khare, A. Kumar, and L. P. Looi. Scalability port: A coherent interface for shared memory multiprocessors. *10th Hot Interconnects Symposium*, August 2002.

[20] David H. Bailey. Ffts in external or hierarchical memory. *Fourth SIAM Conference on Parallel Processing for Scientific Computing*, pages 211–224, 1990.

[21] A. Baniasadi and A. Moshovos. A power-aware branch predictor for high-performance processors. *ICCD*, 00:458–465, 2002.

[22] 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. *27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[23] F. Baskett, T. Jermoluk, and D. Solomon. The 4d-mp graphics superworkstation: Computing + graphics = 40 mips + 40 mflops and 100,000 lighted polygons per second. *COMPCON '88: 33rd IEEE Computer Society International Conference*, pages 468–471, February 1988.

[24] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine cm-2. *SPAA '91: third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, 1991.

[25] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. *ISCA '07: 34th Annual International Symposium on Computer Architecture*, pages 81–91, 2007.

[26] E. Bolotin, Z. Guz, I. Cidon, R. Ginosar, and A. Kolodny. The power of priority: Noc based distributed cache coherency. *NOCS '07: First International Symposium on Networks-on-Chip*, pages 117–126, May 2007.

[27] J. Borkenhagen and S. Storino. 4th generation 64-bit powepc-compatible commenrical processor design. *Processor Design. IBM Server Group Whitepaper*, January 1999.

[28] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded powerpc processor for commercial servers. *IBM Journal of Research and Development*, 44(6):885–898, November 2000.

[29] J. M. Borkenhagen, R. D. Hoover, and K. M. Valk. Exa cache/scalability controllers. *In IBM Enterprise X-Architecture Technology: Reaching the Summit*, pages 37–50, 2002.

[30] Jijun Cao, Jinshu Su, and Chunqing Wu. Using nic-based multicast scheme to improve forwarding rate for application layer multicast. *APSCC '07: second IEEE Asia-Pacific Service Computing Conference*, pages 179–186, 2007.

[31] A. Charlesworth. The sun fireplane interconnect. *SC '01*, November.

[32] A. Charlesworth. Starfire: Extending the smp envelope. *IEEE Micro*, 18(1):39–49, January/February 1998.

[33] A. Charlesworth. The sun fireplane smp interconnect in the sun 6800. *Nineth Hot Interconnects Symposium*, August 2001.

[34] A. Charlesworth. The sun fireplane smp interconnect in the sun fire 3800-6800. *HOTI '01: Ninth Symposium on High Performance Interconnects*, page 37, 2001.

[35] G. Chinya, J. Collins, M. Girkar, H. Jiang, G. Lueh, L. Pearce, X. Tian, H. Wang, P. Wang, and S Yakoushkin. Accelerator exoskeleton. *Intel Technology Journal*, August 2007.

[36] S. Coll, J. Duato, F. Petrini, and F.J. Mora. Scalable hardware-based multicast trees. *ACM/IEEE Conference Supercomputing*, pages 54–54, November 2003.

[37] A. L. Cox and R. J. Fowler. Adaptive cache coherency for detecting migratory shared data. *20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[38] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture: a hardware-software approach*. Morgan Kaufmann, San Francisco, 1999.

[39] Z. Cvetanovic. Performance analysis of the alpha 21364-based hp gs1280 multiprocessor. *30th Annual International Symposium on Computer Architecture*, pages 218–229, June 2003.

[40] William J. Dally and John W. Poulton. *Digital systems engineering*. Cambridge University Press, New York, NY, USA, 1998.

[41] J. Duato, S. Yalamachili, and L. Ni. Interconnection networks: An engineering approach. *Morgan Kaufmann*, 2003.

[42] N. Eisley, L. Peh, and L. Shang. In-network cache coherence. *IEEE Computer Architecture Letters*, 5(1), 2006.

[43] Noel Eisley, Li-Shiuan Peh, and Li Shang. In-network cache coherence. *MICRO 39: 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 321–332, 2006.

[44] N.D. Enright Jerger, Li-Shiuan Peh, and M.H. Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence. *MICRO '08: 41st IEEE/ACM International Symposium on Microarchitecture*, pages 35–46, November 2008.

[45] Stanford Parallel Applications for Shared Memory. `http://www-flash.stanford.edu/apps/SPLASH/`.

[46] S. J. Frank. Tightly coupled multiprocessor system speeds memory-access times. *Electronics*, 57(1):164–169, January 1984.

[47] M. Galles and E. Williams. Performance optimizations, implementation and verification of the sgi challenge multiprocessor. *27th Annual Hawaii International Conference Systems Sciences*, pages 134–143, 1996.

[48] K. Gharachorloo, L. A. Barroso, and A. Nowatzyk. Efficient ecc-based directory implementations for scalable multiprocessors. *SBAC-PAD '00: 12th Symposium on Computer Architecture and High-Performance Computing*, October 2000.

[49] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and design of alphaserver gs320. *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, November 2000.

[50] James R. Goodman. Using cache memory to reduce processor-memory traffic. *ISCA '83: 10th Annual International Symposium on Computer Architecture*, pages 124–131, 1983.

[51] Parallel Architecture Group. `http://www.gap.upv.es/`.

[52] Jim Handy. *The cache memory book*. Academic Press Professional, San Diego, CA, USA, 1993.

[53] J. P. Hayes. *Computer Architecture and Organization*. McGraw-Hill International Editions, San Diego, CA, USA, 1998.

[54] T. Horel and G. Lauterbach. Ultrasparc-iii: Designing third generation 64-bit performance. *IEEE Micro*, 19(3):73–85, May/June 1999.

[55] Intel. From a few cores to many: A tera-scale computing research overview. *ftp://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf*, 2006.

[56] N.E. Jerger, Li-Shiuan Peh, and M. Lipasti. Virtual circuit tree multicasting: A case for on-chip hardware multicast support. *ISCA '08: 35th International Symposium on Computer Architecture*, pages 229–240, June 2008.

[57] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, 2005.

[58] 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.

[59] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V.Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 2003.

[60] Nam Sung Kim, Krisztian Flautner, David Blaauw, and Trevor Mudge. Circuit and microarchitectural techniques for reducing cache leakage power. *IEEE Trans. Very Large Scale Integr. Syst.*, 12(2):167–184, 2004.

[61] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[62] G. Konstadinidis, M. Rashid, P.F. Lai, Y. Otaguro, Y. Orginos, S. Parampalli, M. Steigerwald, S. Gundala, R. Pyapali, L. Rarick, I. Elkin, Ge Yuefei, and I. I. Parulkar. Implementation of a third-generation 16-core 32-thread chip-multithreading sparcs processor. *ISSCC '08: IEEE International Solid-State Circuits Conference*, pages 84–597, February 2008.

[63] S. R. Kunkel. Personal communication. April 2000.

[64] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. C. apin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor.

21st Annual International Symposium on Computer Architecture, pages 302–313, April 1994.

[65] An-Chow Lai and Babak Falsafi. Memory sharing predictor: the key to a speculative coherent dsm. *26th Annual International Symposium on Computer Architecture*, pages 172–183, May.

[66] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[67] J. Laudon and D. Lenosk. The sgi origin: A ccnuma highly scalable server. *24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[68] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. Ibm power6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007.

[69] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. *17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[70] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D.Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The stanford dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[71] Owen Liu. Amd technology: power, performance and the future. *HPC '07: ATIP's 3rd Workshop on High Performance Computing*, pages 89–94, 2007.

[72] A. Louri and K. Kodi. An optical interconnection network and a modified snooping protocol for the design of large-scale symmetric multiprocessors (smps). *IEEE Transactions on Parallel and Distributed Systems*, 15(12), December 2004.

[73] T. D. Lovett and R. M. Clapp. Sting: A cc-numa computer system for the commercial marketplace. *23th Annual International Symposium on Computer Architecture*, May 1996.

[74] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 36(2):50–57, February 2003.

[75] M. P. Malumbres, Jose Duato, and Joseph Torrellas. An efficient implementation of tree-based multicast routing for distributed shared-memory multiprocessors. *SPDP '96: eighth IEEE Symposium on Parallel and Distributed Processing*, page 186, 1996.

[76] Milo M. K. Martin. Token coherence. *Doctoral dissertation, Computer Sciences Deptartment, University of Wisconsin*, 2003.

[77] Milo M. K. Martin, Mark D. Hill, and David A. Wood. Token coherence: Decoupling performance and correctness. *ISCA 2003*, pages 182–193.

[78] Milo M. K. Martin, Mark D. Hill, and David A. Wood. Token coherence: A new framework for shared-memory multiprocessors. *IEEE Micro*, 23(6):108–116, 2003.

[79] Milo M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Maurer, M. Plakal, M. D. Hill, and D. A. Wood. Timestamp snooping: An approach for extending smps. *Ninth International Conference Architectural Support for Programming Languages and Operating Systems*, pages 25–36, November 2000.

[80] Milo M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth adaptive snooping. *Eighth International Symposium on High-Performance Computer Architecture*, pages 251–262, February 2002.

[81] 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. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[82] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. K. Martin, and D. A. Wood. Improving multiple-cmp systems using token coherence. *11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 328–339, 2005.

[83] M.R. Marty and M.D. Hill. Coherence ordering for ring-based chip multiprocessors. *MICRO '06: 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 309–320, December 2006.

[84] Aleksandar Milenkovic. Achieving high performance in bus-based shared-memory multiprocessors. *IEEE Concurrency*, 8(3):36–44, July-September 2000.

[85] Adam Moody, Juan Fernandez, Fabrizio Petrini, and Dhabaleswar K. Panda. Scalable nic-based reduction on large-scale clusters. *SC '03: 2003 ACM/IEEE Conference on Supercomputing*, page 59, 2003.

[86] Gordon E. Moore. Cramming more components onto integrated circuits. *Readings in computer architecture*, pages 56–59, 2000.

[87] David Mosberger. Memory consistency models. *SIGOPS Operative Systems Review*, 27(1):18–26, 1993.

[88] Trevor Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, 2001.

[89] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. The alpha 21364 network architecture. *Nineth Hot Interconnects Symposium*, August 2001.

[90] Henk L. Muller, Paul W. A. Stallard, and David H. D. Warren. The data diffusion machine with a scalable point-to-point network. *Department of Computer Science, University of Bristol*, page 16, October 1993.

[91] Jason Nieh and Marc Levoy. Volume rendering on scalable shared-memory mimd architectures. *VVS '92: 1992 Workshop on Volume Visualization*, pages 17–24, 1992.

[92] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, and M. Parkin. The s3.mp scalable shared memory multiprocessor. *International Conference on Parallel Processing*, volume I:1–10, August 1995.

[93] Soo-Cheol Oh, Sang-Hwa Chung, and Hankook Jang. Design and implementation of cc-numa card ii for sci-based pc clustering. *IEEE International Conference on Cluster Computing*, pages 145–151, 2002.

[94] Dhabaleswar K. Panda, Craig, B. Stunkel, Rajeev Sivaram, Rajeev Sivaram, Dhabaleswar K. P, and A Craig B. Stunkel. Efficient broadcast and multicast on multistage interconnection networks using multiport encoding. *Eighth IEEE Symposium on Parallel and Distributed Processing*, pages 36–45, 1996.

[95] M Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors with private cache memories. *11th Annual International Symposium on Computer Architecture*, pages 384–354, June.

[96] David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach.* Morgan Kaufmann, San Francisco, 2007.

[97] Fabrizio Petrini, Juan Fernandez, Eitan Frachtenberg, and Salvador Coll. Scalable collective communication on the asci q machine. *11th Symposium on High Performance Interconnects*, pages 54–59, August 2003.

[98] F. Pong, M. Dubois, and K. Lee. Design and performance of smps with asynchronous caches. *Technical Report HPL-1999-149, Hewlett Packard, HP Laboratories Palo Alto*, November 1999.

[99] X. Qiu and M. Dubois. Moving address translation closer to memory in distributed shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 16(7):612–623, 2005.

[100] AMD Press Release. Amd introduces the world's most advanced x86 processor, designed for the demanding datacenters. *http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~119768,00.html*, September 2007.

[101] Intel Press Release. Dual core era begins, pc makers start selling intel-based pcs. *http://www.intel.com/pressroom/archive/releases/20050418comp.htm*, April 2005.

[102] Edward Eric Rothberg. *Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization*. PhD thesis, Stanford, CA, USA, 1993.

[103] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing signatures for transactional memory. *MICRO '07: 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 123–133, 2007.

[104] M.D. Schroeder, A.D. Birrell, M. Burrows, H. Murray, R.M. Needham, T.L. Rodeheffer, E.H. Satterthwaite, and C.P. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318–1335, October 1991.

[105] Sun SPARC Enterprise M9000 Server. `http://www.sun.com/servers/highend/m9000/specs.xml`.

[106] Jaswinder Pal Singh. *Parallel hierarchical N-body methods and their implications for multiprocessors*. PhD thesis, Stanford, CA, USA, 1993.

[107] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yaun, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvery, E. Hagersten, and B. Liencres. Gigaplane: A high performance bus for large smps. *Forth Hot Interconnects Symposium*, pages 41–52, August 1996.

[108] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.

[109] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, 1982.

[110] P. Stenstrom, M. Brorsson, and L. Sandberg. Adaptive cache coherence protocol optimized for migratory sharing. *20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[111] Karin Strauss, Xiaowei Shen, and Josep Torrellas. Flexible snooping: Adaptive forwarding and filtering of snoops in embedded-ring multiprocessors. *ISCA '06: 33rd Annual International Symposium on Computer Architecture*, pages 327–338, 2006.

[112] Craig B. Stunkel, Jay Herring, Bulent Abali, and Rajeev Sivaram. A new switch chip for ibm rs/6000 sp systems. *Supercomputing '99: 1999 ACM/IEEE Conference on Supercomputing*, page 16, 1999.

[113] Taeweon Suh, Douglas M. Blough, and Hsien-Hsin S. Lee. Supporting cache coherence in heterogeneous multiprocessor systems. *Design, Automation and Test in Europe Conference and Exhibition Volume II*, page 21150, 2004.

[114] Herbert Sullivan and T R Bashkow. A large scale, homogeneous, fully distributed parallel machine, i. *SIGARCH Computer Architecture News*, 5(7):105–117, 1977.

[115] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. *13th Annual International Symposium on Computer Architecture*, pages 414–423, June 1986.

[116] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Server Group Whitepaper*, October 2001.

[117] Jonathan S. Turner and Jonathan S. Turner. An optimal nonblocking multicast virtual circuit switch. *Proceedings of Infocom*, pages 298–305, 1994.

[118] M. K. Vernon and U. Manber. Distributed round-robin and rst-come rst-serve protocols and their applications to multiprocessor bus arbitration. *15th Annual International Symposium on Computer Architecture*, pages 279–289, May 1988.

[119] W.D. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. *Third International Conference on Architectural Support for Programing Languages and Operating Systems*, pages 243–256, April 1989.

[120] Steven C Woo, Jaswinder P Singh, and John L. Hennessy. The performance advantages of integrating message passing in cache-coherent multiprocessors. Technical report, Stanford, CA, USA, 1993.

[121] Steven Cameron Woo and John Hennessy. *The performance advantages of integrating block data transfer in cache-coherent multiprocessors*. PhD thesis, Stanford, CA, USA, 1996.

[122] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. *HPCA '07: 13th International Symposium on High Performance Computer Architecture*, pages 261–272, 2007.

[123] Youtao Zhang, Lan Gao, Jun Yang, Xiangyu Zhang, and Rajiv Gupta. Senss: security enhancement to symmetric shared memory multiprocessors. *HPCA: 11th International Symposium on High-Performance Computer Architecture*, pages 352–362, 2005.