Switch-based Packing Technique to Reduce Traffic and Latency in Token Coherence

Blas Cuesta

Department of Computer Engineering, Universidad Politécnica de Valencia, Camino de Vera, s/n, 46021, Valencia, Spain. Email address: blacuesa@gap.upv.es Telephone: +34963877007x75745 Fax: +34963877579

Antonio Robles

Department of Computer Engineering, Universidad Politécnica de Valencia, Camino de Vera, s/n, 46021, Valencia, Spain. Email address: arobles@gap.upv.es Telephone: +34963877007x72111 Fax: +34963877579

José Duato

Department of Computer Engineering, Universidad Politécnica de Valencia, Camino de Vera, s/n, 46021, Valencia, Spain. Email address: jduato@gap.upv.es Telephone: +34963877007x79705 Fax: +34963877579

Abstract

Token Coherence is a cache coherence protocol able to simultaneously capture the best attributes of traditional protocols: low latency and scalability. However, it may lose these desired features when (1) several nodes contend for the same memory block and (2) nodes write highly-shared blocks. The first situation leads to the issue of simultaneous broadcast requests which threaten the protocol scalability. The second situation results in a burst of token responses directed to the writer, which turn it into a bottleneck and increase the latency. To address these problems, we propose a switch-based packing technique able to encapsulate several messages (while in transit) into just one. Its application to the simultaneous broadcasts significantly reduces their bandwidth requirements (up to 45%). Its application to token responses lowers their transmission latency (by 70%). Thus, the packing technique decreases both the latency and coherence traffic, thereby improving system performance (about 15% of reduction in runtime).

Keywords: Cache coherence protocol, token coherence, combining switches, message packing, traffic reduction, scalability

1. Introduction

Currently, shared-memory multiprocessors [1, 2, 3] are widely used in modern commercial and scientific computing infrastructures. These systems comprise several processors which share a global memory. Since each processor holds a private cache, they require a cache coherence protocol to consistently view the global memory. The performance and scalability of multiprocessors depend to a large extent on the performance and scalability of the cache coherence protocol. Therefore, in order to provide fast and scalable multiprocessors, cache coherence protocols must incur low latency and have high scalability. However, the protocols based on the traditional snooping and directory approaches cannot simultaneously provide these desirable features. On the one hand, snooping-based protocols [4] commonly provide low latency when they rely on bus-like interconnects, which do not scale. Alternatively, some snooping approaches can also be applied to scalable non-ordered interconnects at the expense of causing indirection or using a greedy algorithm [5], which considerably increases the protocol latency. On the other hand, directory-based protocols [6] use low-latency and scalable interconnects, but in this case the communication between processors is performed through the directory, which introduces indirection and increases protocol latency.

To simultaneously provide low latency and scalability, an approximation based on tokens (Token Coherence [7]) has been proposed. Unlike protocols based on traditional approaches, Token Coherence is able to exploit low-latency interconnects while providing direct communication among processors. This is possible thanks to the use of fast transient requests, which usually succeed in resolving cache misses. The main weakness of this class of request is that, due to their lack of order, when several of them contend for the same memory block, they may generate protocol races and fail to resolve the corresponding cache misses. When this situation happens, Token Coherence must use a starvation prevention mechanism that is able to guarantee the resolution of all cache misses. To date, different proposals of starvation prevention mechanisms have been made, but they present serious drawbacks: persistent requests [8] are broadcast-based and inefficient; token tenure [9] is not based on broadcast, but like persistent requests it is quite inefficient; priority requests [10] efficiently manage tokens, but they are broadcast-based. Thus, although one of the mechanisms (priority requests) is efficient, it is based on broadcast. The bandwidth requirements of this class of message increase quadratically with the system size. Consequently, although in small systems their use may not entail a problem, in medium and large systems they will require much more bandwidth than that provided by the interconnection network. As a result, the network will be congested, which may increase the protocol latency excessively.

Given that the proposed starvation prevention mechanisms are inefficient or require the use of broadcast messages, Token Coherence is not suitable for medium/large systems (where protocol races may be common) and its use is restricted just to systems with a low number of nodes. To improve this aspect, we propose a switch-based strategy which can increase the scalability of broadcast messages when injected in off-chip networks. Since these networks are slower than, for instance, on-chip networks, the injection of near-simultaneous broadcast messages will probably congest the interconnect and, as a result, they are accumulated in the buffers of the traversed switches. These accumulated messages are likely to convey almost the same information because they were generated to resolve certain protocol race (contention over the same block). Therefore, the switch can take advantage of the time that they remain in its buffers to merge several broadcast messages into just one, being able to additionally discard the redundant information. Since most of the information held by them is redundant, the packing of those messages will drastically reduce the quantity of information that will have to be transmitted through the network. As a result, the bandwidth required by the injected broadcast messages diminishes significantly, thereby increasing the scalability of broadcast messages. When applying this technique to the priority request mechanism, Token Coherence is provided with an efficient and scalable starvation prevention mechanism, which extends its applicability to larger systems.

Although this technique is effective in off-chip networks, it may be barely effective in on-chip networks because they are faster and have smaller buffers, which reduces the probability that several broadcast messages coincide in the same switch buffers. Thus, our technique mainly aims to improve off-chip networks, which are frequently used in multicomputers and to interconnect CMPs. Notice that, despite the trend to include an increasingly number of cores in CMPs, the need of larger systems will continue to require connecting several CMPs through off-chip networks.

Another aspect of Token Coherence that can be improved thanks to the packing strategy that we propose is the latency of write misses upon highly-shared blocks. Before modifying a block, a processor must collect all the block's tokens. To this end, it issues a write request and waits for a token response from each of the block's sharers. If a lot of nodes are sharing the block, the write request will result in a burst of token responses directed to the same node (the writer). Although these responses do not require a lot of bandwidth (they are point-to-point messages and they do not convey the memory block), they may collapse the recipient, which will make the latency of write misses considerably large. Given that all the token responses are directed to the same node and they hold almost the same information, switches can take advantage of the packing technique to concentrate (when possible) the tokens carried by several responses in just one. Thus, instead of having to wait for a response from each sharer, the writer will only have to wait for a few responses, which can contribute to reduce the congestion in switches and, therefore, to speed up the protocol.

The rest of this paper is organized as follows. In Section 2, we present some background about Token Coherence that is necessary to better understand the rest of the paper. Section 3 analyzes the problems of Token Coherence, which motivate this work. In Section 4, we describe the switch-based packing technique focused on broadcast messages and in Section 5 we analyse how the packing technique can be extended to improve other aspects of Token Coherence or other protocols. Section 6 discusses the contributions of the proposals made in this paper. Finally, in Section 7, we summarize the main conclusions.

2. Background and Related Work

2.1. The Token Coherence Protocol

Token Coherence [7] is a framework for producing coherence protocols. It captures the best aspects of both snooping-based and directory-based protocols by decoupling the correctness substrate (which provides coherence) from the performance policy (which provides efficiency). Thus, Token Coherence ensures coherence without relying on bus-like interconnects or directories.

Token Coherence uses token counting to enforce the invariant of a single writer or multiple readers. At system initialization, the system assigns each block of the shared memory T

Table 1: Mapping of MOESI states to token counts

State	**Tokens**
Invalid	0 tokens
Shared	At least 1 token, but not the *owner* token
Owned	At least the *owner* token
Exclusive	All tokens (dirty bit unset)
Modified	All tokens (dirty bit set)

tokens. One of the tokens is designated as the *owner token* that can be marked as either clean or dirty. Initially, the block's home memory module holds all the tokens for a block. Tokens are allowed to move between system nodes as long as the system maintains the following rules, which can correspond to the MOESI coherence states [11], as shown in Table 1:

1. *Conservation of tokens.* After system initialization, tokens cannot be created or destroyed. One token for each block is the owner token.
2. *Write Rule.* A node can write a block only if it holds all the T tokens for that block and has valid data. After writing the block, the writer sets the owner token to dirty.
3. *Read Rule.* A node can read a block only if it holds at least one token for that block and has valid data.
4. *Data Transfer Rule.* If a coherence message contains a dirty owner token, it must contain data.
5. *Valid-Data Bit Rule.* A node sets its valid-data bit for a block when a message arrives with data and at least one token. A node clears the valid-data bit when it no longer holds any tokens.

By means of the performance policy, nodes decide when and to whom the system should send coherence messages (requests and responses). Initially, on a cache miss, processors issue a transient request. Transient requests are simple requests that are not guaranteed to succeed, but (in the common case) they usually succeed in resolving cache misses. Transient requests are sent following one of these policies:

- *TokenB (Token Broadcast).* Transient requests are directly broadcast to all nodes and the home memory module.

- *TokenD (Token Directory).* Transient requests are first sent to the home memory module, where a directory decides to which nodes (if any) it should forward the request.
- *TokenM (Token Multicast).* Transient requests are directly sent to a predicted destination set of nodes based on the observation of past events.

Nodes respond to transient requests as they would do with a traditional MOESI policy. Although transient requests will usually succeed in getting all the requested tokens, sometimes they will fail because neither the counting rules nor the performance policy ensures forward progress. For example, tokens can be delayed arbitrarily in transit, tokens can be "ping-pong" back and forth between nodes, or many nodes may wish to simultaneously access the same block. In order to detect such situations, nodes use a timeout. Thus, if after twice the node's average miss latency the cache miss has not been completed, Token Coherence assumes that the transient request has failed and uses a starvation prevention mechanism to resolve the miss. To this end, initially, the *persistent request mechanism* [8] together with several optimizations [5] were proposed. This mechanism always succeed in resolving cache misses, but it is extremely inefficient mainly because it overrides the MOESI policy and starvation situations are always resolved by a strict and inefficient strategy. An alternative starvation prevention mechanism named *priority requests* [10] tackles most of its problems. According to this mechanism, when a possible starvation situation is detected, a priority request is broadcast through an *ordered path.* As a result, all the nodes receive all the priority requests in the same order (which prevents the generation of new races). Besides, priority requests are remembered in tables at least until being completed. Due to the fact that priority requests are served using the MOESI policy and also due to the lack of explicit acknowledgments, the priority request mechanism prevents starvation by a more elegant, flexible, and efficient strategy.

2.2. Related Work

A wide variety of proposals are focused on reducing the traffic and latency of cache coherence systems. Since these proposals can be applied at different levels (coherence policy, network, and protocol among others), most of them are complementary and can be used simultaneously. Here, we briefly comment on some of the most relevant ones.

One way to reduce the traffic required to ensure coherence is by taking into account the data access patterns in the coherence policies. To this end, many different policies have been proposed, such as MSI [12], MOSI [11], MESI [13], or MOESI [11], as well as optimizations for migratory data [14].

A common solution that addresses the traffic problems caused by broadcast messages in snooping protocols is the snoop filter. Snoop filters can be classified as destination, source, and in-network filters. Destination filters [15, 16] hold filtering information at the destination nodes. They save cache-tag look-up power, but their main drawback is that they do not save interconnect bandwidth. On the other hand, source filters [17, 18] maintain filtering information at source nodes. In this case, they do save interconnect bandwidth and power. However, their main drawback is that they only filter snoops over non-shared data and they do not act over shared ones. In-network filters [19] place filtering information in the network switches or routers. Those filters are more efficient, but they make the switch design more complex. Besides, they introduce significant control traffic between switches, which may lead to an important increase of the broadcast latency. In addition, this technique could not be applied to the starvation prevention mechanisms required by some protocols (such as Token Coherence) because they cannot guarantee that the filtering information stored in switches is totally trustful. To make it totally reliable, they need to apply additional mechanisms that would introduce considerable delays, mainly in medium and large systems.

N. D. Enright et. al [20] propose a coherence protocol named VTC which keeps track of sharers of a coarse-grained region and, consequently, its requests only have to be multicast to region sharers (instead of broadcast). Multicasts are supported by virtual circuit tree multicasting [21], which is a proposal orthogonal to ours and both techniques could be jointly used. VTC is not suitable for supporting a starvation prevention mechanism for Token Coherence because it requires a lot of resources and adds excessive overhead (in comparison with other proposed mechanisms). Nevertheless, VTC could be used to propose an alternative performance policy in Token Coherence different to TokenB, TokenD, or TokenM. Such a policy would not require the use of the starvation prevention mechanism because it can guarantee the resolution of all cache misses. Furthermore, in that case, VTC could take advantage of the packing technique that we propose to reduce the bandwidth requirements of the injected traffic.

In [22, 23] the authors propose techniques that predict the destination set of requests to avoid having to broadcast them. However, these two proposals are only applicable to transient requests because, since Token Coherence does not guarantee in-order receipt of requests, predictors cannot determine with total guarantee the set of sharers. As a result, they could fail in resolving some cache misses and the starvation prevention mechanism must ensure their completion.

A. Raghavan et. al propose in [9, 24] a starvation prevention mechanism that is not based on broadcast. Tokens are associated a timer at their arrival at nodes. If the timer associated with the tokens finishes and an acknowledgment is not received from the directory, a possible starvation situation is assumed, having to forward the tokens to the directory. The directory then activates a single request and sends all the tokens it receives until completing it. Although correct, this scheme has several problems. In particular, each cache miss requires an acknowledgment, which increases the network traffic in the common case. In addition, each node will require as many timers as the maximum number of cache misses that can be served before receiving the acknowledgments from the directory. Besides, the starvation prevention mechanism is specially inefficient in case of highly-contended blocks, since tokens are tenured one by one (sequentially), which is a problem in medium/large systems. Furthermore, a scheme based on acknowledgments between the nodes and the directory is used for resolve contention, which may increase the cache miss latency considerably. Like persistent requests, it resolves races overriding the performance policy which is in charge of providing efficiency. Thus, many racing requests to the same block may cause tokens to inefficiently flow. Another serious problem is that if an acknowledgment is delayed (e.g., due to congestion), tokens are written back to memory, which will entail an increase in the cache miss rate. Finally, due to the fact that processors with untenured tokens ignore direct requests, the delay of acknowledgments may cause a lot of direct requests not to be quickly served. Let see some of these problems by an example. Imagine that a node A wants to write certain memory block and it issues a request to all sharing nodes and the directory.

All nodes sharing the block invalidate the copy they hold in their caches and send all tokens to A. Besides, the home activates A's request by sending to it an acknowledgment. Next, let me assume that A collects all block's tokens but it has not received the acknowledgment yet (because it is delayed in the interconnect). During this time, although A's request is already completed, it will not be able to serve requests issued from other processors (for that block). In addition, after a timeout interval, A will invalidate its block copy and will send all tokens to the home. As a result, if A wants to access the block again, a new cache miss will happen only because an acknowledgment is delayed.

Other proposals reduce traffic by avoiding the issue of broadcast messages when it is unnecessary, keeping them just for the cases when it is totally necessary, imitating to what directory protocols do. Thus, for example, the recently launched Magny-Cours processors [25] assume a snooping-like protocol and, to reduce the number of broadcasts, small directory caches are used for tracking the most recently accessed blocks. However, this class of solutions increase the storage requirements (directory) and requests must be sent first to the directory cache and later to nodes (if proceeds), which increases their latency. Additionally, Kim et al. [26] use the operating system to detect private blocks, which do not require snoops and, consequently, requests for them are not broadcast. Although this proposal is efficient, it cannot be applied to the starvation prevention mechanism since this mechanism is intended mainly for contended blocks which obviously are always non-private blocks.

We proposed in [10, 27] an alternative starvation prevention mechanism for Token Coherence called the priority request mechanism. Although it is based on broadcast, it is able to efficiently resolve races by applying the performance policy, which lowers the latency of cache misses under high contention. In [28] we proposed a preliminary version of the packing mechanism over priority requests. Besides, this mechanism is complementary to that proposed in [29], which allows to simultaneously serve several transient/starved requests by using a single multicast response.

3. Motivation

To avoid the drawbacks of totally ordered interconnects and directories, Token Coherence uses unordered requests (i.e., transient requests). They are fast and usually succeed in

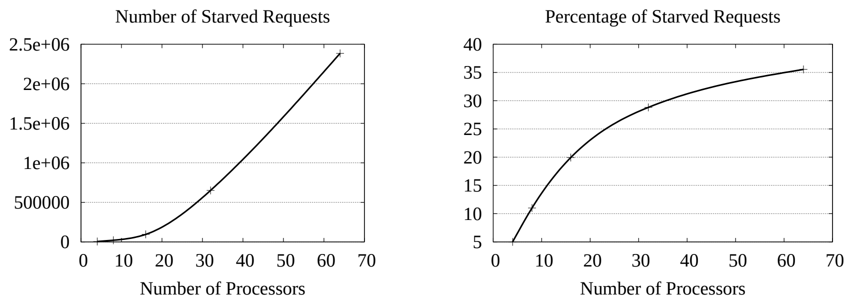


Figure 1: Absolute number and percentage of requests suffering starvation due to protocol races according to the system size. These values are the average ones for all applications cited in Section 6 (Table 2)

solving cache misses. However, due to their lack of order, they may fail. When this happens, Token Coherence requires a starvation prevention mechanism that can ensure forward progress. In very small systems, due to the low latency of misses and the low number of nodes, few transient requests will fail and, consequently, the starvation prevention mechanism will be rarely used, not having a significant impact on global performance. On the contrary, in medium/large systems, the latency of cache misses increases and the number of nodes is also larger. As a result, the probability that a transient request fails (because of a protocol race) grows considerably, as shown in Figure 1. This figure illustrates the number and percentage of requests that must be resolved by the starvation prevention mechanism. Hence, in a 4-processor system only 5% of the misses require the use of the starvation prevention mechanism. This value increases up to about 35% in a 64-processor. Therefore, in medium/large systems, the starvation prevention mechanism will be used more frequently and its impact on global performance will be much more significant. Given that current multiprocessor systems include an increasingly number of nodes and cores, it is not unreasonable to expect the starvation prevention mechanism to be more and more used. Consequently, to avoid degrading protocol performance, it should be scalable and efficient.

Up to now, different starvation prevention mechanisms have been proposed. Both persistent requests and token tenured mechanisms do not resolve races by using the performance policy, which efficiently move tokens between system nodes. As a result, those mechanisms are not suitable in scenarios where the starvation prevention mechanism is often used. On the contrary, the priority request mechanism is able to solve races while using the performance policy. Unfortunately, priority requests are based on broadcast messages, which lack of scalability. Thus, in large systems, when a protocol race triggers the mechanism, a lot of priority requests (for the same memory block) will be broadcast simultaneously, which will flood and saturate the network. Therefore, the latency of both the cache misses involved in the race and those misses outstanding at that time will be quite high. Hence, although priority requests efficiently manage tokens, their broadcast nature poses a serious problem to implement Token Coherence in medium and large systems.

To improve this aspect, we propose an effective network switch-based packing technique. In this sense, when the available bandwidth provided by the network is not enough to quickly deliver all the injected priority requests, they will begin to accumulate in the buffers of the traversed switches. Given that those messages convey almost the same information, switches can take advantage of the time they are waiting in the buffers to merge several of them into a single one, disposing of the redundant information. As a result, the bandwidth requirements of all the priority requests diminish dramatically.

Additionally, we can apply the packing technique to solve the problem caused by the use of non-silent invalidations. When a block held by certain node is invalidated (due to an eviction or an incoming invalidation message), the tokens that it holds will have to be forwarded to either the home memory module or another node since tokens cannot be destroyed. This may lead to serious inefficiencies in case of writes over highly-shared blocks because, before writing the block, the writer must get an exclusive copy of the block (collect all the block's tokens). To this end, it sends a write request informing about the intention of modifying the block. On the write request arrival, the nodes that hold any of the block's tokens reply to the requester with a token response, which conveys all the tokens held by that node. Once the intended writer has received all the tokens, it is allowed to modify it. If the block to modify is shared across a lot of nodes, the writer will have to wait for a lot of token responses. Since all those token responses are generated almost simultaneously and they are directed to the same node, the writer becomes a bottleneck and the latency of the token responses increases significantly. As a result, write misses will be slow to resolve.

The packing technique can help to address that problem too. When a write request results into a burst of token responses directed to the same node, the responses will accumulate in switches. Therefore, the idea is to concentrate (when possible) the tokens carried

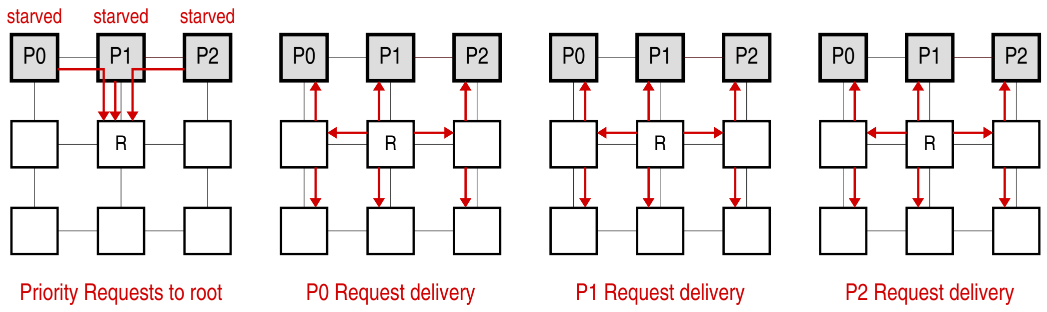


Figure 2: Routing of priority requests through ordered paths. R is the root switch of the ordered path

by several responses on just one. Thus, instead of having to wait for a response from each sharer, only few responses will be received. This will reduce the write latency and will alleviate the congestion of the buffers, mainly in the writers.

4. Switch-based Packing Technique

In this section we describe a switch-based technique that, applied to broadcast messages, is able to widen their scalability. We link its description to the priority request mechanism because, due to its main features, it can make the most of it. However, this technique is not restricted to priority requests, since it can also be applied to other kind of messages, such as token responses, or even to other protocols, as commented on Section 5.

Protocol races happen when several nodes simultaneously contend for the same memory block. In Token Coherence, those nodes detect the race by a timeout and each one of them broadcasts a priority request. Therefore, it is common that, to solve a race, several priority requests for the same block are issued near-simultaneously. Furthermore, all the priority requests for the same block are firstly routed through the same switch (root switch) and then delivered to the destinations (such as Figure 2 illustrates). When the interconnection network has enough bandwidth to quickly transmit all priority requests, they do not coincide in switch buffers. However, if the bandwidth is not enough, several priority requests conveying almost the same information coincide in the buffers of the visited switches along their path to the destinations. This redundant information that floods the interconnect consumes a significant part of the available bandwidth and congests the network, which makes all network messages remain a long time in switch buffers.

We propose that switches take advantage of the long time that priority requests remain

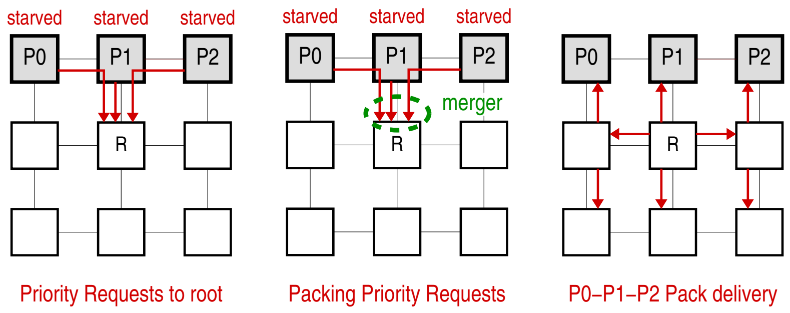


Figure 3: Packing several priority requests into just one on coinciding at switch R

in switch buffers to merge several of them into just one (such as Figure 3 depicts), disposing of the redundant and unnecessary information. Thus, the bandwidth required by several priority requests for the same memory block will be considerably reduced and an appreciable quantity of endpoint traffic (i.e., traffic received by nodes [8]) will be saved. 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 priority requests will be 3200 bytes (10 messages $\times$ 10 bytes per message $\times$ 32 destinations). However, if the 10 priority requests are packed into just one, the total endpoint traffic will be 576 bytes (1 message $\times$ 18 bytes per message $\times$ 32 destinations). Therefore, in this particular example, up to 82% of the endpoint traffic can be saved.

If we only allow the merger of priority requests for the same block, when a switch does not receive them *consecutively*, they will not be able to be merged. This may be frequent in systems where nodes can simultaneously issue more than one request. Thus, to avoid losing packing opportunities, we propose to allow the merger of all priority requests, regardless of the requested memory blocks. Although the packing of priority requests for different blocks does not lead to a traffic reduction (because they do not hold so much redundant information), switches will be able to merge non-consecutive (but close) priority requests for the same block. For instance, imagine a 32-processor system where 5 processors each broadcast a priority request for a block A and other 5 processors broadcast a priority request for a block B. If we did not allow the merger of priority requests for different blocks and the intermediate switches received them interleaved (A, B, A, B ..), they could not be merged, being the total endpoint traffic 3200 bytes (10 messages $\times$ 10 bytes per message $\times$ 32 destinations). However, by allowing their packing, the 10 priority requests can be merged

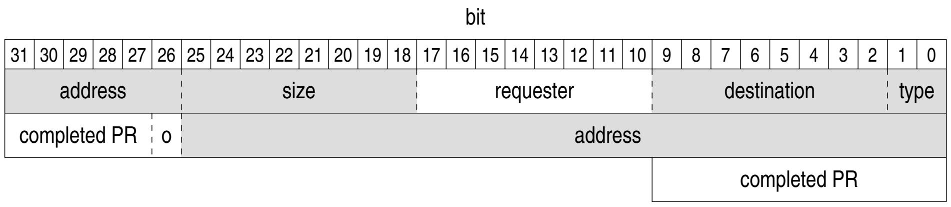


Figure 4: Priority request format. *Type* is the packet type (transient request, response, or priority request). *Size* is the size of the payload (*address*, *o*, and *completed PR* fields). *O* indicates the requested operation (load or store). *Completed PR* is an identifier necessary to inform about the completion of priority requests (for greater detail see [27, 10])

in one. As a result, the generated endpoint traffic would be 800 bytes (1 message × 25 bytes per message × 32 destinations). Therefore, in this case, we could save up to 75% of the generated endpoint traffic. Notice that this time the message size is larger than that assumed in the previous example (25 bytes against 18 bytes) because merged requests do not hold so much redundant information.

4.1. Packing Overview

In general, the packing works as follows. While a switch is receiving a priority request, it checks if that incoming request can be merged with the last priority request stored in the switch input queue. In particular, a merger is only allowed when (1) the last stored priority request is not placed in the queue head and (2) the resulting request does not exceed the maximum number of addresses and requesters allowed per message. Thus, if a switch determines that a merger is possible, it performs it in parallel with the receipt of the request. On the contrary, if the merger is not possible, the incoming priority request is placed in the end of the input queue. Notice that, according to this strategy, priority requests do not wait actively for other requests to compose a single message (which would increase their transmission latency). Indeed, the idea is to take advantage of the time that certain priority request spends in a switch buffer to merge it with the priority requests that may arrive during that time.

4.2. Format of Priority Request Packs

Priority requests comprise the fields shown in Figure 4. Fields in shadow are those common in priority requests for the same memory block. As shown, those requests have

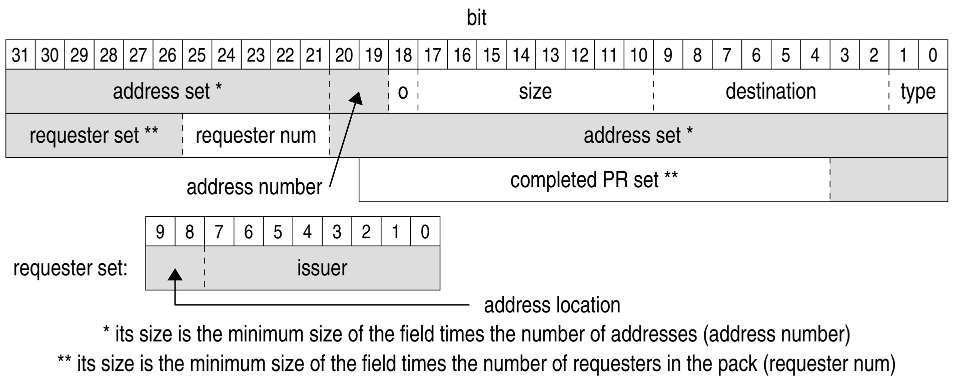


Figure 5: Format of priority request packs for different blocks. A maximum of 4 addresses are allowed

nearly the same information and only 3 out of 10 bytes will not be redundant. Therefore, although several priority requests are broadcast to solve one race, only about 30% of that information would be necessary, as the remaining 70% is redundant and could be saved. The packing technique that we propose aims to remove that redundant information.

From here on, we use the term (priority request) *pack* to refer to several priority requests compressed or packed into a single message. In fact, we will also use the term pack to refer to just a single priority request. The format of packs is shown in Figure 5. Packs comprise most of the fields of priority request messages. In particular, the *address*/*requester*/*completed PR* fields become the *address set*/*requester set*/*completed PR set* fields, respectively. Besides, two new fields, *address number* and *requester number*, are added.

Address set is the list of blocks requested by the requester set. Each block is coded just once in the *address set* field. We assume a maximum of 4 addresses per pack. *Address number* indicates the number of different memory blocks requested in a pack (2 bits).

Requester set is the list of nodes that request any of the memory blocks coded in *address set*. *Requester set* must be implemented as a list and it is not possible its implementation as a bit vector. This is because the packs have to maintain the order in which the included priority requests would have been received if they had not been packed. To indicate the block requested by each node within the *requester set* field, each requester (*issuer*) is coded together with an identifier (*address location*) that indicates the position that its requested block occupies within the *address set* field. The size of *requester set* will depend on the number of messages included in the pack. Hence, if a pack contains n priority requests, its

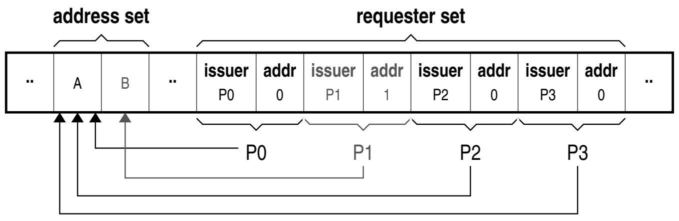


Figure 6: Coding priority requests for different blocks

size will be $n \times (8 + 2)$ bits. *Requester num* indicates the number of requesters in *requester set*. Since it is 5 bits, a maximum of 32 requesters can be included in a single pack.

Completed PR set contains the list of identifiers held by the individual priority requests included in the pack. The identifiers must be placed in the same order as that occupied by their corresponding requester within the *requester set* field. The size of this field will also depend on the number of included requests. Hence, if a pack contains n priority requests, its size will be *2n* bytes.

As shown in Figure 5, the operation field (o) is implemented as a single bit. Therefore, we assume that only the priority requests which request the same operation (i.e., either read or write) will be able to be merged in a pack. We do this because we have observed that most of the priority requests are loads.

Figure 6 illustrates how the coding of addresses and requesters works. As shown, *P0*, *P2*, and *P3* request the memory block that occupies the location 0 within the *address set* list (i.e., block A), whereas *P1* requests the block occupying location 1 (i.e., block B).

4.3. Merger Checking

While a switch is receiving a new pack, it must check whether a merger is possible. This includes several verifications: (1) the last stored pack is not placed in the head of the input buffer, (2) the requested operations of the incoming and the last stored pack match, and (3) neither the number of different addresses nor the total number of requesters of the final pack will exceed the maximum allowed. If these conditions are met, the merger of the incoming pack will succeed. Otherwise, the merger will not be completed (it will be aborted because the merger is performed in parallel with the pack receipt). To avoid delaying the transmission of packs or incurring additional latency, an incoming pack can only be merged with the last pack stored in the input buffer and, therefore, a search is not required. Furthermore, the stored pack cannot be placed at the head of the buffer because that would mean that its transmission to the next switch could have begun or it could begin at any moment and the merger could delay it.

4.4. Auxiliary Buffers

In order to be able to merge an incoming pack with a stored pack without incurring additional latency, the field organization of packs (shown in Figure 5) is slightly modified with respect to that of priority requests (shown in Figure 4). In particular, the *requester set* field is placed at the second last position and the *completed PR set* field is placed at the last one. Besides, two decoupling buffers are associated with each queue dedicated to priority requests, which use a dedicated virtual channel: *completed PR set buffer* and *requester set buffer*. On the one hand, the *completed PR set buffer* is used to hold the *completed PR set* field of the last pack stored in the queue. On the other hand, the *requester set buffer* keeps the *requester number* and the *requester set* fields of the last stored pack. Doing so leaves the *address set* field as the last field in the input queue. Thus, when two packs have to be merged, as the pack is being received, the switch only has to copy the addresses of the incoming pack to the end of the input queue, the *requester set* field to the end of the *requester set buffer*, and the *completed PR set* field to end of the *completed PR set buffer*.

In addition to these two decoupling buffers, switches also require two temporal buffers: an *offset buffer*, which is used to update the address location of the requester set field of the final pack, and an *incoming message buffer*, which keeps the incoming pack in case the merger process must be aborted.

4.5. Packing Process

The receipt of a pack is performed in several steps, as shown in Figure 7). First, the switch receives the pack header, which is copied to the *incoming message buffer*.

Second, next, it receives the *address number* and the *address set* fields which are also copied to the *incoming message buffer*. In addition, each incoming address that is not present in the *address set* field of the stored pack is inserted to the end of the input queue and the *address number* field of the new pack is updated. In parallel, the *offset buffer* is initialized

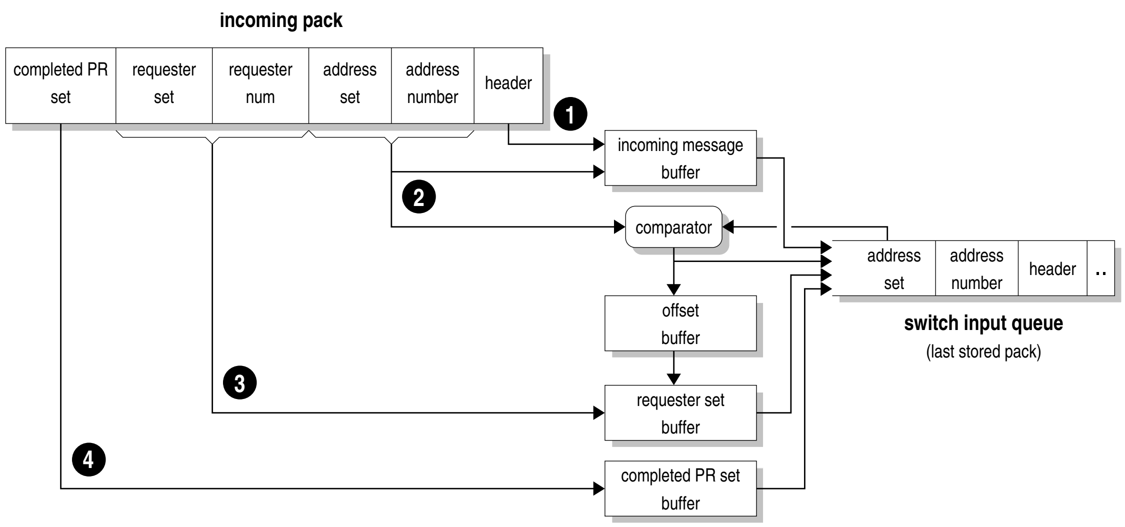


Figure 7: Use of the auxiliary buffers in the packing process

with the location that the incoming address will occupy in the new pack. If the incoming address is already present in the stored pack, it only has to update the *offset buffer*. If the switch realizes that the final pack will exceeds the maximum number of addresses, the packing process is aborted.

Third, after the addresses, the switch receives the *requester num* and *requester set* fields. If it detects that the new pack will exceed the maximum number of requesters, the packing process is aborted. Otherwise, the incoming fields are copied to the end of the *requester set buffer*. During the copy, the *address location* field associated to each issuer of the new pack is updated accordingly by using the *offset buffer*.

And fourth, after the requesters, the switch receives the *completed PR set* field, which is copied to the end of the *completed PR set buffer*.

Figure 8 shows an example of how the packing mechanism works in general terms. In particular, it shows how the offset and decoupling buffers are used to ease the merger. Initially, the last pack of an input queue contains a request for the block B issued by *P2* (*requester set buffer*) and whose *completed PR set* field is 7. A new pack begins to arrive at the switch. The switch receives the first address (A) and, since it is not in the *address set* field of the stored pack, it copies it to the end of the input queue, stores in the *offset buffer* the new location (1), and increases the *address number* field of the new pack (2). The switch then receives the second address (B) and, since it is already present in the *address*

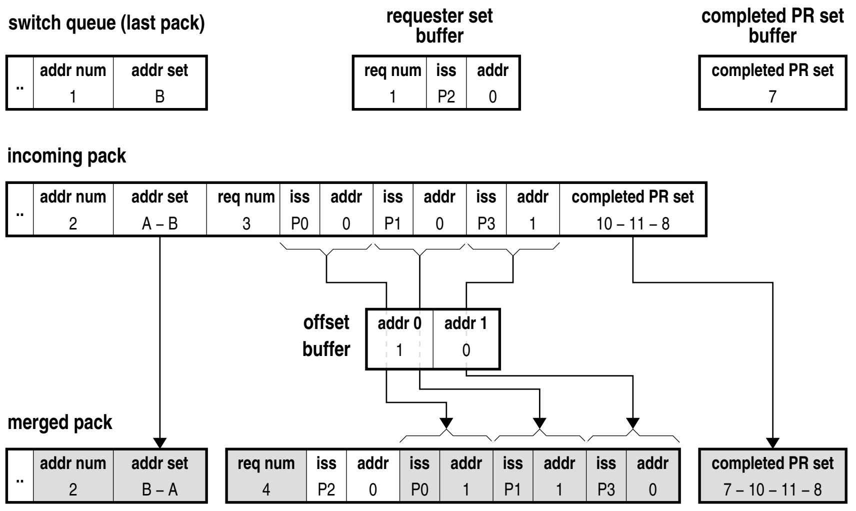


Figure 8: Packing example. *Iss* refers to issuer, *addr* to address location, and *req num* to request number

set field of the stored pack, it only has to store in the offset buffer its location (0). Next, the switch receives the *requester num* field, which is added to that of the stored pack. Since it does not exceed the maximum number allowed, the *requester num* field of the new pack is updated (4) and the packing process continues. Next, the switch receives the *requester set* field, which is copied to the end of the *requester set buffer* updating their address location fields according to the *offset buffer*. Finally, the switch receives the *completed PR set* field which is copied to the *completed PR set buffer*.

Notice that the switch can detect that the merger is not possible at the latest when the *requester num* field is received. Thus, if a merger is not possible, before queuing the incoming pack, the *requester num*, the *requester set*, and the *completed PR set* fields stored in the decoupling buffers will have to be associated again to their corresponding pack by copying their content to the end of the input buffer. Once the referred copies are completed, the incoming pack (retrieved from the *incoming message buffer*) is stored in the end of the input queue, being from that moment on the new last stored pack.

4.6. Adjusting the Starvation Detection Timeout

Processors use a timeout to estimate whether their transient requests are suffering from starvation. This timeout is set to twice the processor's average miss latency. Setting it 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.

Furthermore, this policy adapts to different interconnect topologies and traffic patterns.

Since the value of the timeout depends on the miss latency, several factors will influence when calculating it, such as the size of the used messages. Thus, if we assume two systems which are exactly the same, but the used messages are different size, the timeout intervals estimated in each system will differ because the message size influences their transmission latency and, as a result, the miss latency. 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 may be a problem when we use the packing technique because the timeout intervals are mainly calculated in absence of packs (only in presence of transient requests). Hence, when a pack with several priority requests is generated, its size will considerably differ from that of transient requests. This will influence the latency to resolve the ongoing misses and, therefore, the calculated timeout interval may not be suitable. Given that the size of packs is larger than the size of transient requests, the messages coinciding with packs through the network may have higher latencies, which may increase the average latency of cache misses. Therefore, to avoid assuming starvation when a message is simply delayed in the interconnect, we propose to use a higher timeout interval. For instance, setting the timeout to *three* times the processor's average miss latency.

4.7. Discussion

For the proposed packing technique to be effective, the messages to merge must meet three conditions: (1) they must be sent near-simultaneously, (2) they must be directed to the same destination, and (3) they must hold almost the same information. Thus, for instance, the packing technique is barely effective when applied to transient requests because they do not meet the cited conditions. On the other hand, since priority requests are only issued when several nodes contend for the same memory block, they meet the required conditions: (1) priority requests are issued almost at the same time, (2) they are broadcast messages, and (3) they request the same block. Therefore, the packing technique will be always effective when applied to priority requests.

Some scenarios may increase the probability of packing. Hence, for instance, when nodes are allowed to issue more than one outstanding request, the chances of packing increase because the amount of traffic in the interconnect will be higher, which will make the network more congested. As a result, the transmission of priority requests will be slower, thereby increasing the probabilities of coinciding at switches. Nevertheless, due to simulation constraints, we have been unable to evaluate processors using more than one simultaneous outstanding request.

The use of different ordered paths for priority request issued for different memory blocks may increase the effectiveness of packing. When all priority requests use the same ordered path, the priority requests for the same block may not be received consecutively. However, if they use different ordered paths, the probabilities that they are received consecutively grows and, consequently, the packing will be more effective.

Finally, we would like to highlight that, such as the packing technique has been proposed, it is only suitable for off-chip networks since similar priority requests in on-chip networks will hardly coincide in switch buffers due to two main reasons: switch buffers are very small and the latencies of messages is quite low. Hence, the proposed technique only aims to improve the latency and traffic in off-chip networks.

4.8. Implementation Aspects

In order to avoid that the packing technique incurs additional latency to the transmission of the packed messages, several implementation aspects must be carefully considered.

We consider that an incoming message can only be merged with the last message of the input queue only if that message is not placed at the head. This avoids that the routing of the message at the head of the queue has to be postponed due to the fact that the message is involved in a merger process.

We also assume that the merger of two packs can be done in parallel with the receipt of the incoming pack. For this to be possible, we need as many address comparators as the maximum number of comparisons must be performed per switch cycle. In particular, in this work, we assume that a switch receives 4 bytes per cycle. Therefore, it can receives one address (*address set* field shown in Figure 5) per cycle. Since we limit the maximum number of addresses to 4, the switch will have to perform at most 4 address comparisons per cycle. As a result, it will need 4 comparators. As well, 4 cycles at most (incoming pack with 4 addresses) will be required. Notice also that, since a 4-cycle routing latency is assumed, the address comparisons could be performed in parallel with the routing of the merged pack. This fact would avoid introducing additional delays in the case the merged pack reaches the queue head after starting the packing.

The address comparisons and the verifications of the maximum sizes allowed (number of addresses and number of requesters) are simultaneously performed with the receipt of the rest of the message. This is possible thanks to the proposed organization of the pack fields (Figure 5).

Assuming other network parameters (e.g., higher bandwidth) could make us change some of the implementation decisions that we assume here. In particular, to reduce the number of comparisons, we can reduce the maximum number of different addresses allowed per pack. In fact, if we assume at most one address per pack, the packing process can be tremendously simplified. Alternatively, we could also opt for increasing the number of cycles dedicated to the routing. A small increase of the routing latency is viable in off-chip networks because the latencies are typically quite high (e.g., in comparison with on-chip networks) and that small increase would hardly have a significant impact on the whole system performance. Besides, it could be offset by the benefits of applying the packing technique.

5. Other Uses of Packing

Up to now, the description of the proposed packing technique has been linked to the priority request mechanism. However, this technique is not limited to that mechanism since it can also be used to improve other aspects of Token Coherence or even other protocols. In this section, we first extend the application of the packing technique to improve the latency of writes over highly-shared blocks in Token Coherence, which is evaluated in Section 6.3. Later, in Section 5.2, we briefly explain other scenarios and protocols where the proposed packing technique could be applied. However, since we only focus on Token Coherence, they are not evaluated in this paper, which is left for future work.

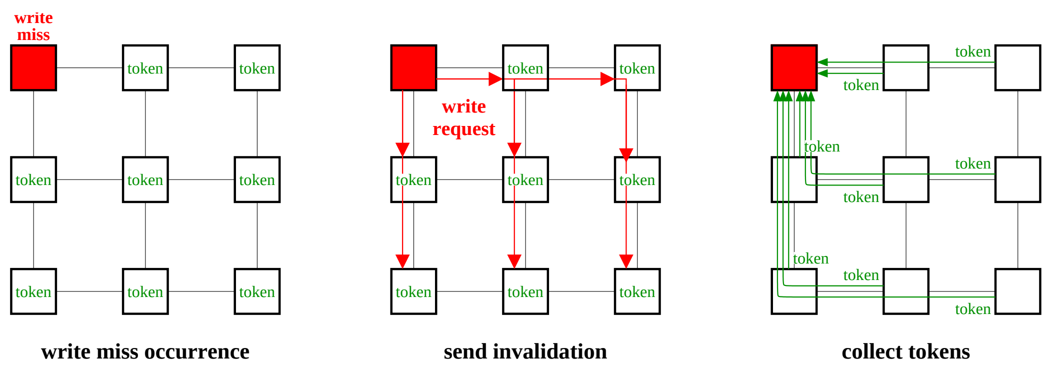


Figure 9: Write over a highly-shared memory block

5.1. Packing Token Responses

In Token Coherence, when a node wants to modify a block shared across a lot of nodes, it sends an invalidation message (write request) and waits for an acknowledgment (token response) from each sharer informing about the invalidation. In this situation, the writer processor may become temporarily a bottleneck since the write request will result in a burst of simultaneous responses directed to it, such as Figure 9 illustrates. This situation worsens as the system size increases because the more processors the system has, the more sharers there may be.

To improve the latency of those writes and to avoid the bottlenecks, in this section we suggest to apply the proposed packing technique to token responses. By packing several responses into just one, the writer will have to wait for less acknowledgments. Thus, the writer will be less congested and the latency of writes over highly-shared blocks diminishes.

Token responses due to write requests are suitable for applying the packing technique because (1) they are generated near-simultaneously, (2) they convey almost the same information, and (3) they are likely to coincide along their way to their destination (see Figure 9).

The packing process is similar to that described in previous sections. First, upon the receipt of a token response, the switch checks whether both the incoming response and the last stored response in the switch buffer hold tokens for the same memory block. Second, if the tokens belong to the same block and both messages are directed to the same destination, the responses are merged. This is done just by (1) adding the tokens held by the incoming message to the stored one and (2) updating the *completed PR* field. Third, if the tokens do not belong to the same block or they are headed for a different destination, the incoming response is queued at the end of the buffer. And fourth, when a response message reaches the head of the queue, it cannot be merged (in that switch).

Unlike priority requests, in this case it is not necessary to modify the format of token responses because the recipient of the message does not need to know the senders of the responses. Rather, it only needs to know the number of received tokens. Consequently, a decoupling buffer is not required. Besides, we do not need to keep the list of completed priority requests (*completed PR* fields) because the *completed PR* field in token responses indicates that the priority request identified by *completed PR* and all the previous ones (those with a lower identifier) are completed. Therefore, the final token response only needs to keep the highest value of all the *completed PR* fields of the packed responses. Given that the message size is not modified when the packing technique is applied to token responses, then it is not necessary to adjust the timeout intervals used to appropriately detect starvation.

Notice that, in this case, the packing process is much simpler than that explained in Section 4 because the message format is not modified and responses only include information (tokens) for just a block.

5.2. Other Protocols

The proposed packing technique can also be applied to address some problems of other protocols. This section only intends to show how the packing technique could be beneficial in other scenarios.

Protocols based on directory caches. Directory caches may suffer frequent replacements (as reported in recent studies [30, 20]), which entails the invalidation of cached blocks. When the directory cache does not keep the full list of sharers (e.g., as it happens in AMD Opteron processors [25]), the invalidation of shared blocks is performed by broadcasts and it must wait for the receipt of several acknowledgments that will have been generated near-simultaneously. Although those invalidations are not in the critical path of cache misses, they may be frequent and they may hurt the performance of other requests due to the bandwidth consumption. Since they are generated near-simultaneously, they are headed for the same destination, and they hold almost the same information, they could be managed by the proposed packing technique.

Protocols based on directory caches or directories. A similar situation to that described in the previous point happens when, instead of invaliding a memory block due to a directory cache replacement, the block is invalidated due to a write and, in addition, the block is highly-contended. In this situation, a burst of similar messages directed to the same destination are generated near-simultaneously.

Protocols that multicast requests to nodes belonging to certain region (broadcast inside a region) [17]. Although in these scenarios broadcast messages are only limited to a region, if they are frequent, the network in such a region may be congested, which will affect to the transmission latency of other messages that go through that part of the network and even to the messages that only move inside that region. Since the used routing algorithm is quite similar to that used by priority requests, those multicast messages are excellent for being merged under the same circumstances than those described for Token Coherence.

6. Experimental Results

In this section, we analyze the contribution of the proposed packing technique in Token Coherence. To this end, we first evaluate the packing technique over priority requests and, later, we evaluate it over token responses. We also see its evaluation when it is simultaneously applied over both priority requests and token responses .

6.1. System Configuration and Benchmarks

We evaluate our proposals with full-system simulation using Virtutech Simics [31] extended with the Wisconsin GEMS toolset [32] which enables detailed simulation of multiprocessor systems. We extended them with a multiprocessor interconnection network simulator developed by the Parallel Architecture Group [33]. We simulate three multicomputer systems: 16, 32, and 64-processor Sparc v9 systems. Each node includes a processor, split L1 caches, unified L2 cache, and coherence protocol controllers. Table 2(a) shows the system parameters, which are similar to those chosen in [7]. Note that the latency of memory (80 cycles) is quite optimistic, but we have just assumed this value to speed up the simulations.

The used workloads consist of the applications from the SPLASH 2 suite shown in Table 2(b). We have chosen only those applications because of the time requirements to

Table 2: (a) System parameters and (b) Application description

(a)

split L1 I&D caches	64 KB, 4-way, 2 cycles
unified L2 caches	4 MB, 4-way, 6 cycles
cache block size	64 bytes
main memory	80 cycles (4 GB)
memory controllers	6 cycles
network link latency	8 cycles
switch cross latency	1 cycle
network routing latency	4 cycles
switch buffer size	5 packets/virtual channel

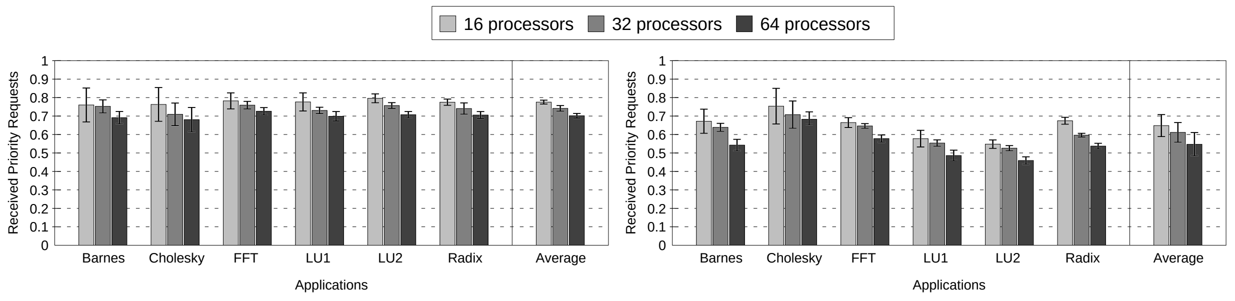
(b)

Barnes	16384 particles
Cholesky	Input file tk29.O
FFT	65536 complex data points
LU1	512x512, contiguous
LU2	512x512, non-contiguous
Radix	256K keys, radix of 1024

simulate the whole suite. 20 simulations were run for each application and system. The points of the figures shown in the next sections were obtained by averaging the results for each application as described in [34]. Besides, figures show the 95% confidence intervals.

We evaluate the referred system assuming that processors are connected through an off-chip 2D mesh. Although we have also evaluated the packing technique assuming a Multistage Interconnection Network, we do not show those results because they are qualitatively similar. We assumed three virtual channels. Responses use the first virtual channel, transient requests use the second one, and the third virtual channel is for priority requests.

First, we show the results of applying our proposal to the priority request mechanism. The evaluated proposal allows the packing of requests for different memory blocks. To simplify the implementation, we assume a single ordered path for all the priority requests and we only allow to pack priority read requests. We also show the comparison between the packing technique using the typical timeout (twice the processor's average miss latency), labeled as *t2*, and the packing technique using a more suitable timeout for the simulated systems (three times the processor's average miss latency), labeled as *t3*, as described in 4.6. This evaluation is done in terms of received priority requests, endpoint traffic, average latency of completing starved requests, link utilization, power consumption, 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 to the data



(a) typical timeout (t2)   (b) suitable timeout (t3)

Figure 10: Normalized endpoint traffic (in messages) due to priority requests when using packing and different timeouts

obtained for the protocol without using packing (baseline system).

Second, we apply the packing technique to token 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 to the data obtained for Token Coherence without using the packing technique (baseline system). Finally, we apply the packing technique simultaneously to priority requests and token responses.

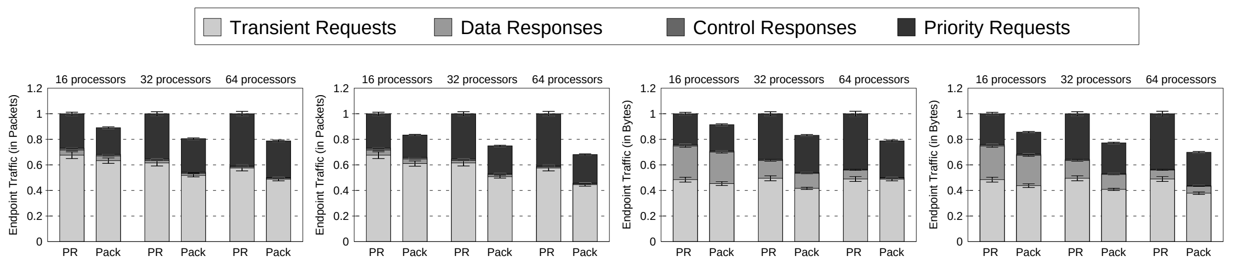
6.2. Evaluation of Packing over Priority Requests

Figure 10(a) shows the endpoint traffic due to the receipt of priority requests when the packing technique is applied. As shown, the number of received priority requests lowers significantly thanks to the packing. This is mainly due to two reasons. On the one hand, several priority request messages are merged into a single message, thereby reducing the number of received requests. On the other hand, the packing technique lowers the number of detected starvation situations. This is because it reduces the congestion in the interconnection network, which in turn makes the transmission of messages faster. Consequently, tokens do not stay for so long travelling between nodes, which increases its availability at nodes. Notice that the reduction in traffic is higher as the system size increases, reaching a reduction of about 30% (on average) in a 64-processor system.

Figure 10(b) shows that the reduction in endpoint traffic is even more impressive when the packing technique is applied together with a more suitable timeout, reaching about 47% on average in a 64-processor system. This is because the mechanism with an unsuitable timeout

Table 3: Percentage of transient requests that cause priority requests. Average values for all applications. *PR* is the baseline protocol, *Pack_t2* refers to the packing technique using the typical timeout, and *Pack_t3* refers to the packing technique using a more suitable timeout

	16 processors	32 processors	64 processors
PR	19.94%	28.82%	35.54%
Pack_t2	16.51%	25.42%	29.41%
Pack_t3	14.32%	21.37%	25.13%



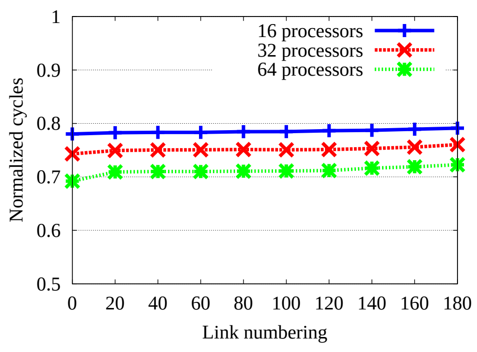
(a) t2, in packets (b) t3, in packets (c) t2, in bytes (d) t3, in bytes

Figure 11: Normalized total endpoint traffic. *PR* stands for Token Coherence using only normal priority requests and *Pack* stands for Token Coherence using the packing technique

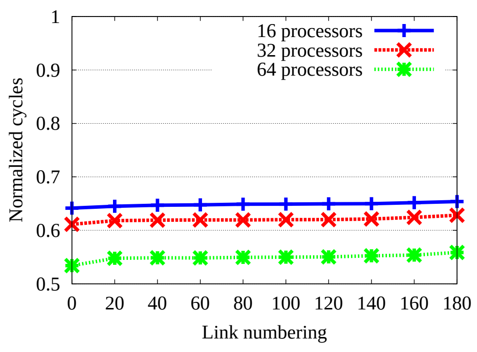
wrongly detects starvation when some messages are simply delayed in the interconnect due to the use of larger messages (packs). However, the use of a more suitable timeout avoids the detection of such situations, which prevents nodes from issuing unnecessary priority requests. In this situation, almost all the priority requests that coincide at switches have most of their fields in common, which makes the packing technique more effective, thereby reaching higher reductions of endpoint traffic due to priority requests, despite triggering a lower number of priority requests.

Table 3 gives the percentages of transient requests that finally cause the generation of priority requests or, in other words, the percentage of cache misses that must be resolved with the starvation prevention mechanism. As shown, as the system size grows, the percentage of cache misses that require the use of the starvation prevention mechanism increases. However, the increase becomes more moderate when using the packing technique and, even more, when a suitable timeout is additionally used.

Figure 11 illustrates the normalized overall endpoint traffic generated during the execution of the applications. *PR* (Priority Requests) refers to the baseline system, whereas *Pack*



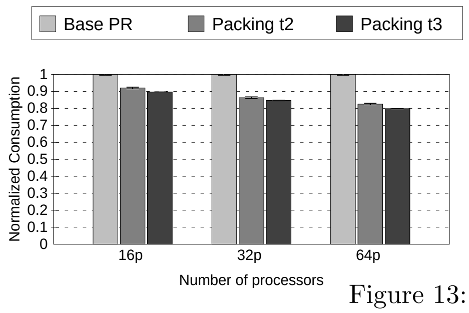
(a) typical timeout (t2)



(b) suitable timeout (t3)

Figure 12: Link utilization normalized to the link utilization without packing of priority requests

refers to the same system using the proposed packing technique. We only show the average values in order to reduce the number of figures. As shown in Figure 11(a), the use of the packing technique entails an important reduction in the overall endpoint traffic. *Transient Requests* refers to the endpoint traffic due to the injected transient request messages. This traffic is directly proportional to the number of cache misses (one cache miss causes the issuing of a transient request). As shown, the packing technique causes a reduction in the number of cache misses, which, in turn, reduces the number of transient requests. This happens mainly due to two reasons. First, the number of issued priority requests decreases, which lets processors hold tokens longer, thereby reducing the number of cache misses. The second reason is that messages have lower latencies because the interconnect is less congested. As a result, processors are blocked waiting for tokens and memory blocks during less time and they can access their cached data more quickly, which reduces the probability that it has to forward the tokens too soon (before it has finished using them in the short term). The reduction of transient requests causes, in turn, the number of responses (*Data Responses* and *Control/Token Responses*) to lower. Besides, as deduced from previous figures, the number of requests suffering starvation also decreases. 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. This reduction in traffic is even more sharply when the packing technique is used along with a more suitable timeout, reaching about 35% of reduction on average in a 64-processor system, which is shown in Figures 11(b) and Figure 11(d).



Average consumption of network elements

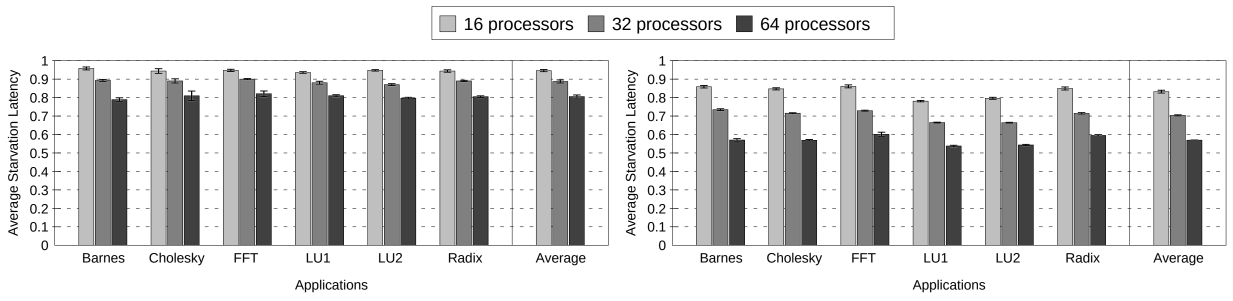
Buffer - Dynamic consump.	6.7×10^{-14} W
Buffer - Static consump.	3.0×10^{-5} J
Crossbar - Dynamic consump.	2.0×10^{-12} W
Crossbar - Static consump.	1.9×10^{-4} J
Link consump.	4.1×10^{-4} W

Figure 13: Power consumption

Figure 12 shows the normalized number of cycles that every switch link is busy due to priority request transmission. The data in the figure only takes into account the utilization of the links used for going from the root switch to the destinations, but not from the issuer to the root. This is because while a priority request is going to the root switch, it is only forwarded through one link (it behaves like a point-to-point message). However, when the priority request is going from the root to the destinations, it is forwarded through several links and it is at that time when priority requests behave like broadcast messages. Hence, to avoid showing links with a very low utilization, we only show the highly utilized ones. Data are normalized to the number of cycles that links are busy due to priority request transmission in the baseline system. Like in the previous section, we only show the average results. As Figure 12(a) illustrates, the proposed packing technique not only reduces the endpoint traffic, but also the overall traffic that goes through the network. The reduction ranges from about 20% in a 16-processor system to 30% in a 64-processor system. Notice that the reduction grows with the system size mainly because the more processors the system has, the more congested the network is. If the network is more congested, messages will be longer in the switch buffers, which will increase the chances of generating a single pack.

Figure 12(b) shows that the packing technique together with a suitable timeout interval can significantly increase the reduction, reducing in about 45% the traffic crossing the network due to priority requests.

Figure 13 shows the normalized power consumption of the different evaluated options. This only takes into account the power consumption in the interconnection network, which is constituted by the consumption of (1) the internal crossbar, (2) the switch buffers and decoupling buffers, and (3) the links. Therefore, the data shown in the figure do not include

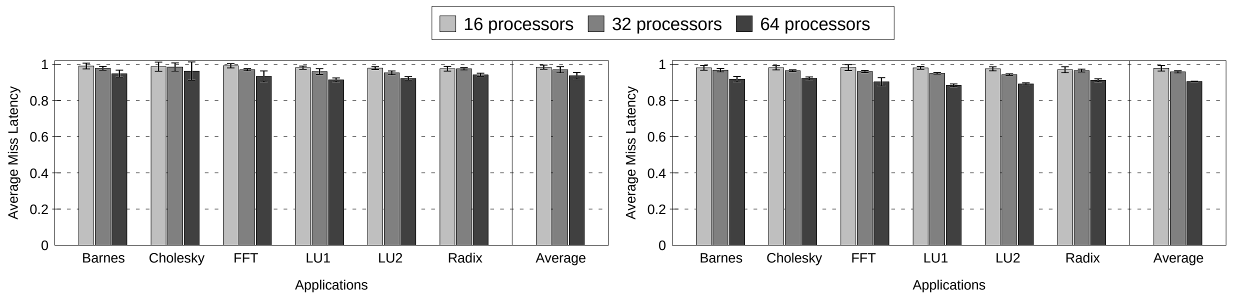


(a) typical timeout (t2) (b) suitable timeout (t3)

Figure 14: Normalized latency of completing starved requests when priority requests are packed

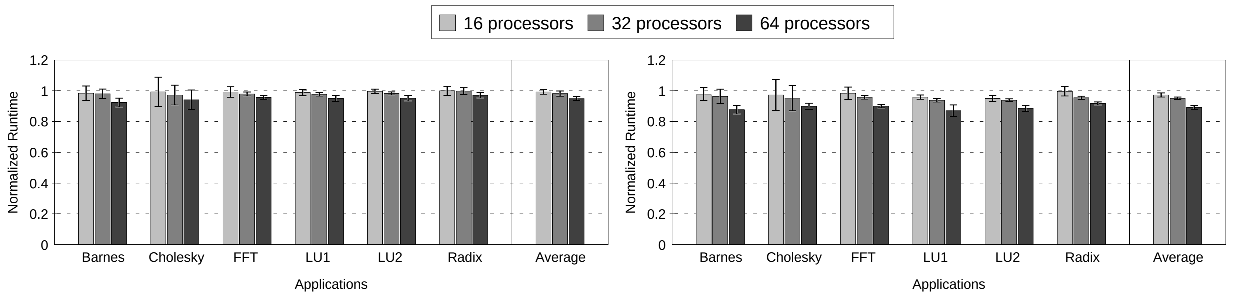
the consumption of caches or other components. To estimate the consumption, we have used ORION 2.0 [35] (assuming a 45nm technology with HP transistors) to model a simple switch. With this model, we obtain the parameters shown in the table of Figure 13. We use those parameters to estimate the consuption of every switch (depending on the number of required buffers and the traffic that crosses it). As shown in the figure, the fact of requiring some additional decoupling buffers does not causes the power consumption to increase. Rather, the impressive saving in traffic allows to significantly reduce the consumption in the crossbar and in the links and this reduction offsets the small increase due to use of additional decoupling buffers. According to these data, the power consumption in the interconnect decreases about 20% in a 64-processor system (on average).

Figure 14 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 14(a), the packing of priority requests decrements the latency of resolving the detected starvation situations. This is possible because, since the interconnection network is less congested, the injected traffic does not suffer so high delays, especially the traffic injected during the starvation situations. Thus, responses sent to solve such situations have lower latencies, which contributes to reduce the average latency of completing starved requests. Besides, when several priority requests are packed into a single message and a node receives it, the latency between the service of sequential priority requests also decreases, which in turn allows nodes to speed up the service of consecutive priority requests. As shown in the figure, since in large systems the reduction in traffic is more significant, the reduction in latency is also higher. This result can



(a) typical timeout (t2) (b) suitable timeout (t3)

Figure 15: Normalized latency of cache misses when priority requests are packed



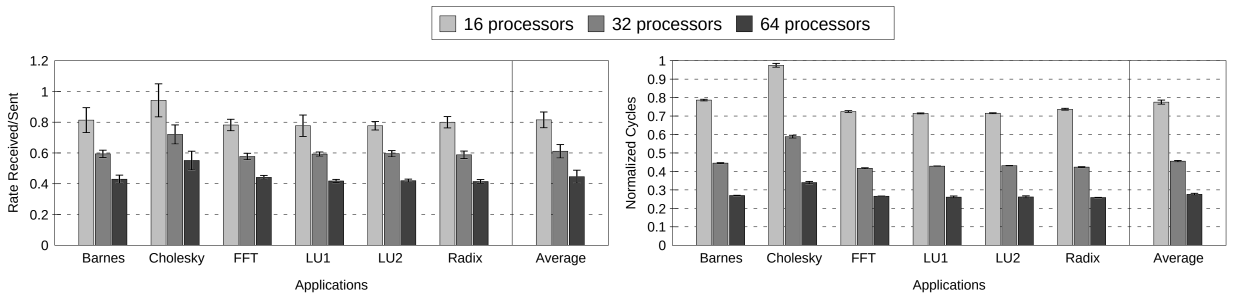
(a) typical timeout (t2) (b) suitable timeout (t3)

Figure 16: Normalized runtime when packing priority requests

be improved even more by using a more suitable timeout, such as Figure 14(b) illustrates. Thus, using the packing technique and the timeout *t3* leads to reduce the average latency of solving starvation in about 40% on average in a 64-processor system.

Thanks to the reduction in both the latency of resolving starvation and traffic, the average latency of solving cache misses decreases. This is illustrated in Figure 15. According to this figure, the packing of priority requests leads to reductions in the latency of cache misses of about 4% on average. This reduction increases when a more suitable timeout is used, reaching cache miss latencies about 10% lower on average in a 64-processor system.

Finally, Figure 16 depicts the normalized runtime of the applications. As shown in Figure 16(a), although the packing technique leads to reductions in both traffic and latency, it hardly affects the runtime of the applications in 16-processor and 32-processor systems. This is because, as stated in other works [8], in small/medium systems the use of broadcast messages is not a serious threat for the performance. In fact, the TokenB protocol, which is based on broadcast, outperforms other protocols based on directory and is the most



(a) (b)

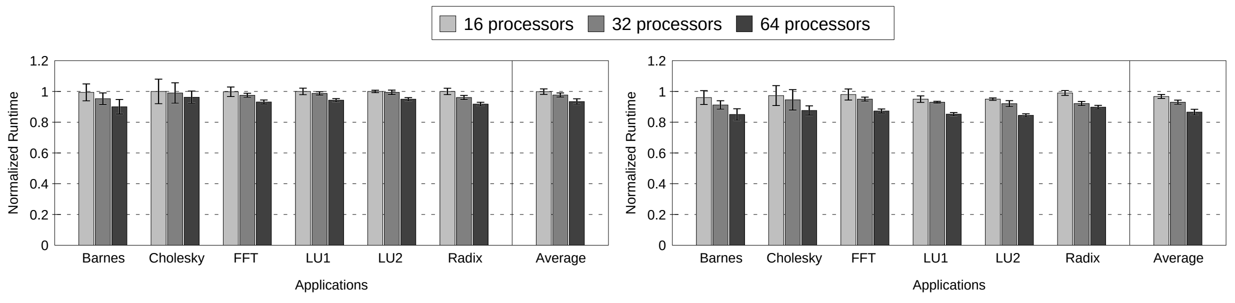
Figure 17: (a) Received/Sent token response rate when applying packing over responses and (b) normalized latency of token responses (without data) when applying packing

efficient option for systems with a moderate number of nodes. However, as the system size increases, the impact of broadcast messages on the overall performance is more considerable because the bandwidth provided by the interconnection network does not scale with the system size. Hence, in 64-processor systems (or larger ones), the use of broadcast messages represent a serious threat for performance. Therefore, in those systems the proposed packing technique will not only contribute to reduce the traffic, but also to reduce the runtime of applications. Furthermore, this performance improvement can be more significant when the packing technique is used along with a suitable timeout, which is shown in Figure 16(b). Hence, the packing technique over priority requests could reduce about 10% the runtime of applications (on average) in a 64-processor system.

6.3. Evaluation of Packing over Token Responses

In this section we evaluate the effects of applying the proposed packing technique just to token responses, which alleviates the problem caused by the use of non-silent invalidations in writes over highly-shared blocks, as explained in Section 5.1.

Figure 17(a) illustrates the rate of received/sent token responses. Obviously, when the packing technique is not used (baseline system), the rate is 1 because, since token responses are unicast messages, processors receive the same number of token responses as they sent. However, when several token responses are packed in transit at switches, processors receive less token responses than they injected. As shown in Figure 17(a), the packing technique significantly decreases the rate of received/sent token responses according to the system size,



(a) (b)

Figure 18: (a) Normalized runtime of applications when applying the packing technique to token responses and (b) normalized runtime when applying packing simultaneously over priority requests and token responses

reaching a reduction of almost 60% on average in case of a 64-processor system. Unlike in the case of priority requests, this reduction will not lead to a significant reduction in the traffic transmitted through the network because (1) this traffic does not represent a significant part of the whole traffic (they are unicast messages and they are only sent in case of writes) and (2) the packing of token responses is likely performed close to the destination. Therefore, although in this case the packing technique does not lead to a reduction of injected or transmitted token responses, it will entail a reduction of received token responses, which will alleviate the input buffers of NICs or nodes.

Besides reducing the congestion of the input buffers, applying the packing technique over token responses can help us to decrease their average latency, which is depicted in Figure 17(b). As shown, the packing technique has an impressive impact on the average latency of token responses, reaching a reduction of approximately 70% on average in a 64-processor system. Notice that, when the packing technique is not used and several token responses are directed to the same destination, their receipt is serialized. However, if several token responses are packed into a single message, their receipt is simultaneous, which contributes to reduce their average latency. Consequently, the packing technique speeds up the process for collecting all the tokens associated to a block, making writes over highly-shared blocks much faster.

Figure 18(a) shows the runtime of applications when packing token responses. As shown, in 16- and 32-processor systems, the packing technique does not have a significant influence on the runtime of applications because, in those systems, there are few nodes and, therefore,

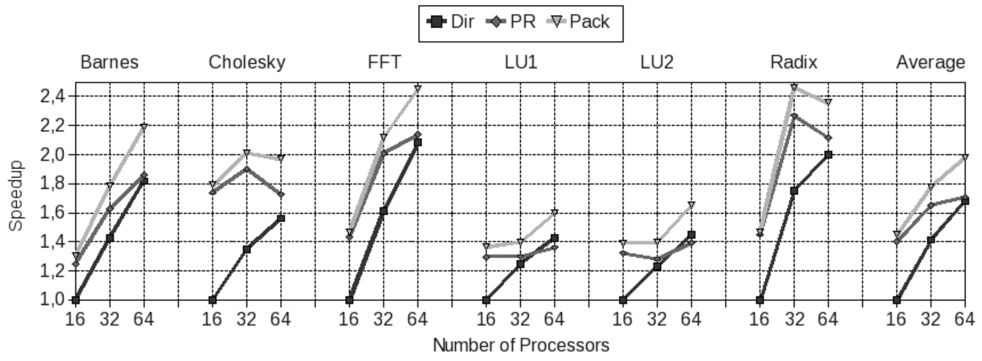


Figure 19: Speedup of applications in a 16, 32, and 64-processor system when using a directory-based protocol (*Dir*), TokenB using priority requests (*PR*), and TokenB using priority requests and the packing technique over priority requests and token responses with suitable timeout(*Pack*). Data are normalized to those for the directory protocol in a 16-processor system

tokens are not highly-spread, thereby taking short time to collect all of them. However, in larger systems, a block can be shared by more processors and, consequently, tokens are more spread. Hence, in those systems, the process to collect all the tokens may take long time and the packing technique can help nodes to shorten that latency, which influences the system performance. Thus, in a 64-processor system the packing technique can reduce the runtime of applications in about 9% on average. Note that, when we apply the packing technique over priority requests, we are only acting during the starvation situations, whereas when the packing technique is applied over token responses, it is being used in the common case.

6.4. Joint Evaluation

After evaluating the packing technique over priority requests and token responses separately, we also want to show the effects of a joint evaluation. Figure 18(b) depicts the normalized runtime of applications when the proposed packing technique is simultaneously applied over priority requests and token responses and a suitable timeout is used (t3). As shown, the packing technique may lead to a more significant reduction in runtime. Furthermore, as the system size increases, the saving is higher, which indicates that the mechanism provides certain scalability to the coherence protocol. Hence, in a 64-processor system, the runtime of applications diminishes by 15% on average.

Finally, Figure 19 illustrates the performance comparison between a directory-based protocol (*Dir*), TokenB using the priority request mechanism (*PR*), and TokenB using the priority request mechanism and the packing technique over priority requests and token responses along with a suitable timeout (*Pack*). This figure illustrates that the performance of the directory protocol scales quite well when the system size increases from 16 to 64 processors. On average, in 64-processor systems, it can reach a speedup of about 1.7. When using *PR*, the performance is significantly improved when running in 16 and 32-processor systems. However, due to the lack of scalability of broadcasts, when running in a 64-processor system, the performance hardly improves and it is likely that in larger systems it will get worse. This situation changes when using *Pack* because it widens the scalability of broadcast messages, being able to get a speedup of about 2 on average in a 64-processor system with respect to the directory protocol in a 16-processor system. Hence, thanks to the applied packing technique, Token Coherence using TokenB outperforms a typical directory protocol in the range of the evaluated system sizes.

7. Conclusions

In this work we have proposed an effective strategy that alleviates the scalability problems of the broadcast messages used by the priority request mechanism of Token Coherence. This technique is implemented in the network switches and it is only used when the required bandwidth exceeds that provided by the network. In this situation, several broadcast messages are packed into a single one. In doing so, the bandwidth requirements of broadcasts diminish drastically and they become more scalable. As shown in the evaluation, the reduction of the traffic affects other class of messages since the overall traffic network is significantly alleviated. As a result, the latency of cache misses lowers, which in turn contributes to improve the runtime of applications.

The proposed switch-based packing technique can be applied to the responses generated due to invalidations too. In making so, the number of token responses and their latency reduce, which contributes to improve the latency of writes over highly-shared blocks. Therefore, we could conclude that the proposed packing technique can help Token Coherence to reduce the congestion caused by the required coherence messages, thereby significantly improving the performance, efficiency, and scalability of Token Coherence.

References

[1] S. Phillips, VictoriaFalls: Scaling Highly-Threaded Processor Cores, 19th Annual IEEE Symposium on High Performance Chips (Hot Chips) (2007).

[2] B. Sinharoy, POWER7 multi-core processor design, 42nd Annual IEEE/ACM International Symposium on Microarchitecture (2009).

[3] O. Liu, AMD technology: power, performance and the future, 3rd workshop on High performance computing in China (ATIP's) (2007) 89–94.

[4] J. R. Goodman, Using Cache Memory to Reduce Processor-Memory Traffic, 10th Annual International Symposium on Computer Architecture (ISCA) (1983) 124–131.

[5] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. K. Martin, D. A. Wood, Improving multiple-CMP systems using Token Coherence, 11th International Symposium on High-Performance Computer Architecture (HPCA) (2005) 328–339.

[6] A. Agarwal, R. Simoni, J. Hennessy, M. Horowitz, An Evaluation of Directory Schemes for Cache Coherence, SIGARCH Computer Architecture News 16 (1988) 280–298.

[7] M. M. K. Martin, M. D. Hill, D. A. Wood, Token Coherence: Decoupling Performance and Correctness, 30th Annual International Symposium on Computer Architecture (ISCA) (2003) 182–193.

[8] M. M. K. Martin, Token Coherence, The University of Wisconsin - Madison (2003). Supervisor - M. D. Hill.

[9] A. Raghavan, C. Blundell, M. M. K. Martin, Token tenure: Patching token counting using directory-based cache coherence, 41th International Symposium on Microarchitecture (2008).

[10] B. Cuesta, A. Robles, J. Duato, An effective starvation avoidance mechanism to enhance the Token Coherence protocol, 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) (2007) 47–54.

[11] P. Sweazey, A. J. Smith, A class of compatible cache consistency protocols and their support by the IEEE Futurebus, 13th Annual International Symposium on Computer Architecture (ISCA) 14 (1986) 414–423.

[12] F. Baskett, T. Jermoluk, D. Solomon, The 4D-MP Graphics Superworkstation: Computing + Graphics = 40 MIPS + 40 MFLOPS and 100,000 Lighted Polygons per Second, 33rd IEEE Computer Society International Conference (COMPCON) (1988) 468–471.

[13] M. S. Papamarcos, J. H. Patel, A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories, 11th Annual International Symposium on Computer architecture (ISCA) (1984) 348–354.

[14] A. L. Cox, R. J. Fowler., Adaptive Cache Coherency for Detecting Migratory Shared Data, 20th Annual International Symposium on Computer Architecture (ISCA) (1993) 98–108.

[15] A. Moshovos, G. Memik, B. Falsafi, A. Choudhary, Jetty: Filtering Snoops for Reduced Energy

Consumption In SMP Servers, International Symposium on High Performance Computer Architecture (HPCA) (2001).

[16] V. Salapura, M. Blumrich, A. Gara, Design and Implementation of the Blue Gene/P Snoop Filter, International Symposium on High Performance Computer Architecture (HPCA) (2007).

[17] J. F. Cantin, M. H. Lipasti, J. E. Smith, Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking, International Symposium on Computer Architecture (ISCA) (2005).

[18] A. Moshovos, Regionscout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence, International Symposium on Computer Architecture (ISCA) (2005).

[19] N. Agarwal, L.-S. Peh, N. K. Jha, In-network coherence filtering: Snoopy coherence without broadcasts, IEEE/ACM International Symposium on Microarchitecture (MICRO) (2009).

[20] N. D. E. Jerger, L.-S. Peh, M. H. Lipasti, Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence, IEEE/ACM International Symposium on Microarchitecture (MICRO) (2008) 35–46.

[21] N. D. E. Jerger, L.-S. Peh, M. H. Lipasti, Virtual circuit tree multicasting: A case for on-chip hardware multicast support, International Symposium on Computer Architecture (ISCA) (2008) 229–240.

[22] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, D. A. Wood, Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors, SIGARCH Computer Architecture News 31 (2003) 206–217.

[23] H. Wang, D. Wang, P. Li, J. Wang, C. Li, Reducing Network Traffic of Token Protocol Using Sharing Relation Cache, Tsinghua Science & Technology 12 (2007) 691–699.

[24] A. Raghavan, C. Blundell, M. M. K. Martin, Token tenure and PATCH: A predictive/adaptive token-counting hybrid, ACM Transactions on Architecture and Code Optimization 7 (2010) 6:1–6:31.

[25] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, B. Hughes, Cache hierarchy and memory subsystem of the AMD opteron processor, IEEE/ACM International Symposium on Microarchitecture (MICRO) 30 (2010) 16–29.

[26] D. Kim, J. Ahn, J. Kim, J. Huh, Subspace snooping: Filtering snoops with operating system suport, 19th International Conference on Parallel Architectures and Compilation Techniques (PACT) (2010) 111–122.

[27] B. Cuesta, A. Robles, J. Duato, Efficient and scalable starvation prevention mechanism for Token Coherence, Accepted for publication in IEEE Transactions on Parallel and Distributed Systems (TPDS) (2011).

[28] B. Cuesta, A. Robles, J. Duato, Switch-Based Packing Technique for Improving Token Coherence Scalability, Parallel and Distributed Computing, Applications and Technologies (PDCAT) (2008) 83–90.

[29] B. Cuesta, A. Robles, J. Duato, Improving Token Coherence by multicast coherence messages, 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP) (2008) 269–273.

[30] M. R. Marty, M. D. Hill, Virtual hierarchies to support server consolidation, International Symposium on Computer Architecture (ISCA) (2007) 46–56.

[31] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, Simics: A full system simulation platform, Computer 35 (2002) 50–58.

[32] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, D. A. Wood, Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset, SIGARCH Computer Architecture News 33 (2005) 92–99.

[33] GAP - Parallel Architecture Group, `http://www.gap.upv.es/`, 2011.

[34] A. R. Alameldeen, D. A. Wood, Variability in Architectural Simulations of Multi-Threaded Workloads, 9th International Symposium on High-Performance Computer Architecture (HPCA) (2003) 7–.

[35] A. B. Kahng, B. Li, L.-S. Peh, K. Samadi, ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration, Design, Automation Test in Europe Conference Exhibition (DATE) (2009) 423–428.

Author Biography and Photograph

Blas Cuesta received the MS degree in Computer Science from the Universidad Politécnica de Valencia, Spain, in 2002. In 2005, he joined the Parallel Architecture Group (GAP) in the Department of Computer Engineering at the same university as a PhD student with a fellowship from the Spanish government, receiving the PhD degree in computer science in 2009. He is working on designing and evaluating scalable coherence protocols for shared-memory multiprocessors. His research interests include cache coherence protocols, memory hierarchy designs, scalable cc-NUMA and chip multiprocessor architectures, and interconnection networks.

Antonio Robles received the MS degree in physics (electricity and electronics) from the Universidad de Valencia, Spain, in 1984 and the PhD degree in computer engineering from the Universidad Politécnica de Valencia in 1995. He is currently a full professor in the Department of Computer Engineering at the Universidad Politécnica de Valencia, Spain. He has taught several courses on computer organization and architecture. His research interests include high-performance interconnection networks for multiprocessor systems and clusters and scalable cache coherence protocols for SMP and CMP. He has published more than 70 refereed conference and journal papers. He has served on program committees for several major conferences. He is a member of the IEEE Computer Society.

José Duato received the MS and PhD degrees in electrical engineering from the Universidad Politécnica de Valencia, Spain, in 1981 and 1985, respectively. He is currently a professor in the Department of Computer Engineering at the Universidad Politécnica de Valencia. He was an adjunct professor in the Department of Computer and Information Science at The Ohio State University, Columbus. His research interests include interconnection networks and multiprocessor architectures. He has published more than 380 refereed papers. He proposed a powerful theory of deadlock-free adaptive routing for wormhole networks. Versions of this theory have been used in the design of the routing algorithms for the MIT Reliable Router, the Cray T3E supercomputer, the internal router of the Alpha 21364 microprocessor, and the IBM BlueGene/L supercomputer. He is the first author of the Interconnection Networks: An Engineering Approach (Morgan Kaufmann, 2002). He was a member of the editorial boards of the IEEE Transactions on Parallel and Distributed Systems, the IEEE Transactions on Computers, and the IEEE Computer Architecture Letters. He was a cochair, member of the steering committee, vice chair, or member of the program committee in more than 55 conferences, including the most prestigious conferences in his area of interest: HPCA, ISCA, IPPS/SPDP, IPDPS, ICPP, ICDCS, EuroPar, and HiPC. He has been awarded with the National Research Prize *Julio Rey Pastor 2009*, in the area of Mathematics and Information and Communications Technology and the *Rei Jaume I Award on New Technologies 2006*.