Efficient and Scalable Starvation Prevention Mechanism for Token Coherence

Blas Cuesta, Antonio Robles, *Member, IEEE Computer Society,* and José Duato

Abstract—Token Coherence is a cache coherence protocol that simultaneously captures the best attributes of the traditional approximations to coherence: direct communication between processors (like snooping-based protocols) and no reliance on bus-like interconnects (like directory-based protocols). This is possible thanks to a class of unordered requests that usually succeed in resolving the cache misses. The problem of the unordered requests is that they can cause protocol races, which prevent some misses from being resolved. To eliminate races and ensure the completion of the unresolved misses, Token Coherence uses a starvation prevention mechanism named persistent requests. This mechanism is extremely inefficient and, besides, it compromises the scalability of Token Coherence since it requires storage structures (at each node) whose size grows proportionally to the system size. While multiprocessors continue including an increasingly number of nodes, both the performance and scalability of cache coherence protocols will continue to be key aspects.

In this work we propose an alternative starvation prevention mechanism, named priority requests, that outperforms the persistent request one. It is able to reduce the application runtime more than 20% (on average) in a 64-processor system. Furthermore, thanks to the flexibility shown by priority requests, it is possible to drastically minimize its storage requirements, thereby improving the whole scalability of Token Coherence. Although this is achieved at the expense of a slight performance degradation, priority requests still outperform persistent requests significantly.

Index Terms—Cache coherence, Token Coherence, starvation prevention, scalability.

1 Introduction

Shared-memory multiprocessors [1], [2], [3], [4] are quite popular nowadays because they provide high performance while being relatively easy to program. To coordinate the different caches, these systems require a cache coherence protocol, which has high impact on the system performance. Multiprocessors include an increasingly number of nodes and, consequently, to perform well, they require low-latency and scalable cache coherence protocols. Although a broad quantity of protocols has been proposed, they are based on two traditional ideas, each one with their own advantages and disadvantages. On the one hand, snooping-based protocols [5] commonly provide low-latency cache misses when they rely on bus-like interconnects, which are not scalable. However, they can also be applied to non-ordered interconnects at the expense of introducing indirection or using a greedy algorithm [6], which increases the miss latency. On the other hand, directory-based protocols [7] can use low-latency interconnects (not bus-like) which scale better. However, the communication among nodes is performed through a slow component called directory, which introduces indirection and increases the latency.

To simultaneously capture the best attributes of these two approximations, a new class of protocols based on tokens (Token Coherence [8]) has been proposed. Hence, unlike the snooping-based and directory-based approximations, Token Coherence is able to exploit any unordered interconnect while providing low-latency cache misses. This is possible thanks to transient requests, which are fast messages that usually succeed in solving cache misses. However, transient requests may fail when contending for the same memory block because they are unordered and may lead to the occurrence of protocol races. To solve races and guarantee miss completion, Token Coherence uses a starvation prevention mechanism. Initially, several implementations of a mechanism named persistent requests [9] were proposed. Some of these proposals are extremely inefficient since they use additional components (arbiters) that introduce indirection and increase the latency of cache misses. There exists another proposal without arbiters that outperforms the other implementations, but it is still inefficient. Besides, this approach requires a table at each node whose size grows proportionally to the number of nodes in the system and which requires associative lookup. These facts represent a serious drawback because there is a trend towards incorporating more and more nodes onto systems, being the scalability a key issue.

Despite their advantages, actual multiprocessor systems do not implement protocols based on tokens because they have serious drawbacks too. Most of these drawbacks are caused by both the timeouts used to detect starvation and the mechanism used to solve it (the persistent request mechanism). Since the problems related to timeouts have already been tackled in other works [6], here we focus on addressing the main problems of the persistent request mechanism, thereby making more feasible the incorporation of Token Coherence

- *B. Cuesta, A. Robles, and J. Duato are with the Department of Computer Engineering, Universidad Politécnica de Valencia, Camino de Vera, s/n, 46021, Valencia, Spain.*
E-mail: {blacuesa, arobles, jduato}@gap.upv.es

into actual systems. To this end, in this work we propose an alternative starvation prevention mechanism, named priority requests [10]. The priority request mechanism relies on a total order of requests. However, in this case, unlike the snooping-based approximations, the ordering is provided by the routing algorithm. Hence, priority requests can resolve starvation in a way much more elegant, natural, and efficient than that used by the persistent request mechanism. Furthermore, the flexibility shown by priority requests allows to define a strategy to decouple the size of the storage structures required at each node from the number of system nodes. Therefore, the size of the structures required to solve starvation can be drastically reduced.

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 this paper. Section 3 analyzes the problems of the starvation prevention mechanisms proposed up to now, which motivate this work. In Section 4, we describe the proposal of a more efficient and scalable starvation prevention mechanism. In Section 5, we show and discuss the main contributions of the proposals made in this work. Finally, in Section 6, we summarize the main conclusions of our work.

2 Background and Related Work

2.1 The Token Coherence Protocol

Token Coherence [8] is a framework for producing coherence protocols. It captures the best aspects of the traditional approximations by decoupling the correctness substrate (which provides coherence) from the performance policy (which provides efficiency). Thus, coherence can be ensured without the need of relying on bus-like interconnects or directories.

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

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)

Although the rules above ensure coherence, they do not ensure forward progress. This must be guaranteed in all cases not explicitly disallowed by the token counting rules. 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 to the same block (protocol race). Thus, to ensure that all attempts to read or write a block will eventually succeed, Token Coherence uses the *persistent request* mechanism. A processor invokes a persistent request whenever it detects it has failed to collect sufficient tokens during a timeout interval. The substrate arbitrates among the outstanding persistent requests to determine the current active request for each block. Once decided, the substrate broadcasts the active persistent requests to all the system nodes. The nodes must both remember all the active persistent requests and redirect their tokens (those tokens currently available and those received in the future) to the requesting processor until the starved requester deactivates it. A processor initiates the deactivation of its persistent request when it receives sufficient tokens and the read/write operation is accomplished.

To determine the active persistent request, several alternatives have been proposed [9]. The first option is a *single centralized arbiter.* In this case, persistent requests are directed to the arbiter. The arbiter stores all the received persistent requests in a table and activates a single one by a broadcast. Nodes remember the active request by a single-entry persistent request table. While a persistent request is active, all processors send their tokens to the initiator. Once the initiator has (1) received sufficient tokens, (2) received valid data, and (3) observed the activation of its own persistent request, it sends a message to the arbiter to deactivate the request. The arbiter deactivates it by informing all nodes, which delete the entry from their tables.

To avoid that a single centralized arbiter becomes a bottleneck, several arbiters can be used (*banked arbitration*). Each arbiter is responsible for a fixed portion of the global address space. In this case, the nodes each must have a persistent request table that contains one entry per arbiter. The process of preventing starvation is similar to the process followed by a single arbiter.

The solutions based on arbiters present the problem of indirection: activation/deactivation messages are sent to the arbiter, which is in charge of broadcasting them. This delays the delivery of persistent requests and, therefore, increases the latency. To avoid it, a *distributed arbitration* option was proposed. When a processor detects possible starvation, it sends a persistent request directly to all the processors and the home memory module. Each one of them remembers the received persistent requests in a table. Since the table may have multiple persistent requests for the same memory block, the persistent request issued by the processor with the lowest identifier is the active persistent request. Same as before, the processors forward their tokens to the initiator of the active persistent request. Once the requester has received sufficient tokens, it deactivates its request by broadcasting a deactivation message. To prevent higher-priority processors from starving out lower-priority processors, this approach uses a simple strategy: a high-priority processor cannot issue a persistent request until all the persistent requests received before its last deactivation have been deactivated too.

Both the token counting rules and the starvation prevention mechanism ensure correct operation in all cases. However, they do not provide efficiency nor fast operation. The component that deals with it is the performance policy. Thus, in absence of persistent requests, this component decides when and to which processors the system should send coherence messages (requests and responses). Three different policies have been proposed to solve cache misses. First, in *TokenB (Token Broadcast)*, transient requests are directly broadcast to all the processors and the home memory module. Second, in *TokenD (Token Directory)*, transient requests are first sent to the home memory module, where a directory decides to which processors (if any) it should forward the request. Third, *TokenM (Token Multicast)*, transient requests are directly sent to a predicted destination set of processors based on the observation of past events.

Processors respond to transient requests as they would do in a MOESI protocol depending on: (1) the request type (read or write), (2) the recipient state, and (3) the presence of active persistent requests. If after twice the processor's average miss latency the request has not been completed, the persistent request mechanism is used. Note that, if there is not any active persistent request, transient requests are efficiently served. Furthermore, their service is quite fast because they do not need to wait for completing one transient request to begin to serve the next one. However, we cannot get those advantages by persistent requests because they override the performance policy and a persistent request cannot be served until the issuer of the active persistent request informs about its completion. In fact, persistent requests make transient requests lose their advantages because transient requests cannot be served while there is at least one active persistent request.

2.2 Related Work

Michael R. Marty et al. [6] proposed several improvements over the persistent request mechanism. The concept of persistent read request is introduced, which lets processors keep one token instead of forwarding all of them. Furthermore, the starvation prevention mechanism is used in several ways: after resending several times the same transient request, after sending once a transient request, or when a highly-contended block is detected (to avoid the timeout). Furthermore, a delay is introduced after modifying a memory block, thereby ensuring that a processor will hold permissions for the block long enough to perform a short critical section.

In [12] Michael R. Marty et al. proposed using a Ring-Order protocol to order requests and, therefore, to avoid generating protocol races, which eliminates the need to reissue requests or to issue persistent requests. However, this protocol is restricted to systems with ring interconnects, being impossible its implementation in systems with other interconnects.

Arun Raghavan et al. proposed in [13] an alternative method to prevent starvation. 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 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 requests are served one by one and a scheme based on acknowledgments between the nodes and the directory is used, which may increase the cache miss latency considerably.

Niket Agarwal et al. proposed in [14] a technique that lets the network order requests in a distributed manner. When a request is injected into the network, it is assigned a snoop-order. The snoop-orders are used by the NICs to deliver all the requests in the same order. An important problem of this proposal is the fact that requests must remain stored in NIC buffers until they can be delivered in order. Although several alternatives have been proposed, the number of buffers grows proportionally to the system size (either in NICs or in routers), which is not scalable. In addition, expiration messages are broadcast to avoid deadlock, which increases the generated traffic significantly. In order for this technique to work, point-to-point ordering and deterministic routing algorithms are required. In addition, to provide finite destination buffering, the routers must implement techniques that add considerable complexity.

We proposed in [15] a mechanism to pack several

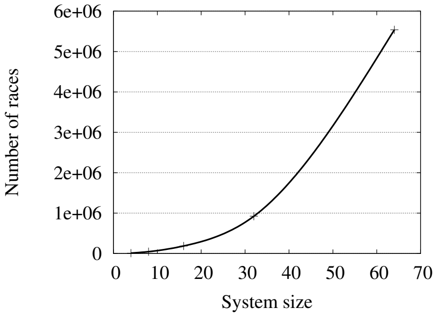


Fig. 1. Number of protocol races for Barnes

priority requests into just one, which reduces the harm of broadcasts and improves their scalability. That mechanism is complementary to that proposed in [16], which allows to simultaneously serve several transient/starved requests by using a single multicast response.

3 Motivation

In very small systems, the starvation prevention mechanism used by Token Coherence will be rarely used since the probability of contention among processors is low and unordered requests quickly resolve cache misses. However, in larger systems, the average latency to solve a cache miss increases. As a result, the probability of contention (simultaneous misses for the same block) increases and, therefore, the starvation prevention mechanism will be more frequently used. This is shown in Figure 1, which illustrates the number of cache misses that must be resolved by the starvation prevention mechanism during the execution of the Barnes application. In relative terms, while in a 4-processor system only 6% of the misses require the use of the starvation prevention mechanism, it increases as long as the system is getting larger (14% with 8 processors, 24% with 16 processors, 31% with 32 processors, and 36% with 64 processors). Thus, although in small systems the starvation prevention mechanism has little impact on the overall performance, in medium and large systems it significantly affects the whole performance, being desirable an efficient mechanism.

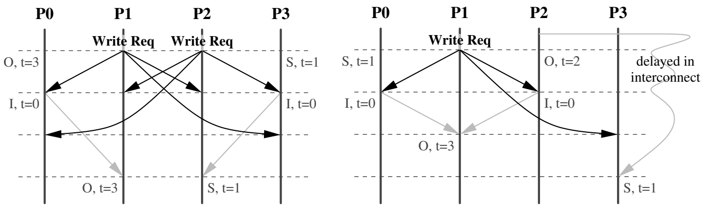
To date, two implementation options of persistent requests have been proposed: one based on centralized arbiters and another based on distributed arbiters. The first option uses one or more arbiters. Processors first send persistent requests to the arbiter. This activates one of the requests and forwards it to all the processors. Processors use a table (which has as many entries as arbiters) to remember all the active persistent requests. The main advantages of this option are its simplicity, it requires tables whose size does not depend on the number of processors, and associative lookup is not required. Nevertheless, it presents a serious disadvantage that Token Coherence seeks to avoid: the indirection (which considerably) increases the latency. On the contrary, with the distributed arbiters option, processors communicate directly and indirection is avoided. Furthermore, the lack of centralized arbiters removes the associated queueing and highly-contended blocks can be better managed. Thanks to this, the distributed arbiters option outperforms the centralized arbiters, mainly in small/medium systems. However, this option presents some disadvantages that make it unsuitable for medium/large systems. In particular, it requires tables that (1) require associative lookup and (2) their size depends on the number of processors. As a result, they do not scale and its application in medium/large systems is not suitable. In addition, the distributed arbiters option does not use any ordering point for persistent requests. In consequence, nodes may receive them in different order, which may create temporal races and penalize the performance. The temporal races are solved by a fixed-priority scheme that decides the order in which the received requests must be served. However, this scheme may create load imbalance when using certain types of non-fair synchronization.

The described disadvantages are inherent to each implementation option, but persistent requests also present other problems common to all of them. In particular, one of the most harmful disadvantages is the fact that persistent requests override the performance policy, thereby losing efficiency and low-latency. Thus, while transient requests can be served simultaneously, persistent requests must be served one after another, which increases the latency noticeably. Furthermore, persistent requests are so strict that they do not allow the service of transient requests while they are active. Consequently, all the transient requests generated while a persistent request is active will not be served, which leads to new races. This problem is aggravated by the deactivation messages since, apart from increasing the network traffic, they increase the time that the persistent requests are active.

To improve the starvation prevention mechanism and avoid most of the problems caused by persistent requests, in this work we propose an improved version of the priority request mechanism. In [10], a preliminary version was presented. Here, we extend it with a more refined and scalable version, including a more extensive evaluation process too. In particular, in this work we provide the priority request mechanism with an effective strategy that decouples the table size from the system size. The result is a new starvation prevention mechanism that contributes to significantly increase the efficiency and scalability of Token Coherence.

4 The Priority Request Mechanism

In snooping and directory-based protocols, requests always succeed in resolving cache misses because they are ordered either by the interconnect or by the directory. However, in token-based protocols, requests are not ordered in any way and, therefore, they may generate protocol races. Furthermore, requests are not remembered and the token unavailability may also cause the requests to fail.



(a) Protocol Race (b) Tokens Unavailability

Fig. 2. Examples of unsuccessful transient requests. t stands for the token count and O, S, and I refer to the Owner, Shared, and Invalid states, respectively

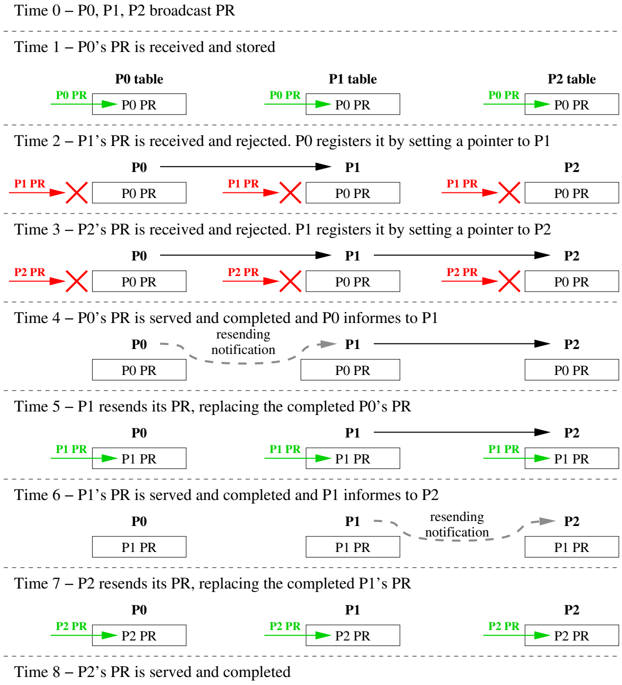

Time 0 – P0, P1, P2 broadcast PR
Time 1 – P0's PR is received and stored
P0 table
P1 table
P2 table
P0 PR
P0 PR
Time 2 – P1's PR is received and rejected. P0 registers it by setting a pointer to P1
P0
P1
P2
P1 PR
Time 3 – P2's PR is received and rejected. P1 registers it by setting a pointer to P2
P2 PR
Time 4 – P0's PR is served and completed and P0 informes to P1
resending
notification
Time 5 – P1 resends its PR, replacing the completed P0's PR
P1 PR
Time 6 – P1's PR is served and completed and P1 informes to P2
resending
notification
Time 7 – P2 resends its PR, replacing the completed P1's PR
P2 PR
Time 8 – P2's PR is served and completed

Fig. 3. Sketch of how the priority request mechanism works. Single-entry tables are assumed. *PR* stands for Priority Requests

A **protocol race** can occur anytime multiple transient requests simultaneously contend for the same memory block. Since they are unordered messages, their receipt order may differ between nodes, which results ambiguous and may prevent their completion. This situation is illustrated in Figure 2(a). *P1* and *P2* each broadcast a transient write request to collect all the block tokens, which are initially shared by *P0* (O state) and *P3* (S state). Due to the lack of global order, *P0* receives the requests in {*P1*,*P2*} order and *P3* receives them in {*P2*,*P1*} order. Consequently, *P0* serves *P1*'s request and *P3* serves *P2*'s. As a result, both *P1*'s and *P2*'s requests fail to collect all the block's tokens.

The other reason why a transient request can fail is **token unavailability**. Transient requests can only be served at their arrival because nodes do not remember them. Therefore, transient requests upon memory blocks whose tokens are not available at their arrival will fail. Figure 2(b) shows this situation. *P1* requests all the block's tokens by a transient write request. When it arrives at nodes, they forward to the issuer all the tokens they hold at that moment. However, since there are some tokens in transit (traveling from *P2* to *P3*) that are not forwarded, *P1*'s request fails.

When those situations happen, Token Coherence requires the use of a starvation prevention mechanism that can ensure the completion of all the cache misses. In this work, we propose a new starvation prevention mechanism named priority requests alternative to the persistent request one. To ensure that all attempts to read or write a block will eventually succeed, our mechanism relies on priority requests which (1) are ordered messages (thereby avoiding the protocol races) and (2) are remembered by nodes (thereby solving the token unavailability problem). Hence, since the two situations that can cause a request to fail are resolved, the completion of priority requests is guaranteed.

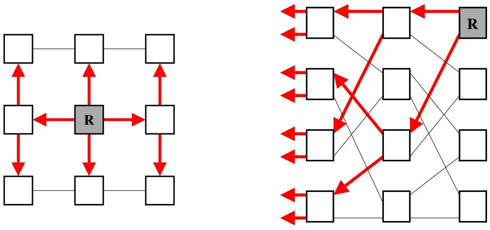
Although persistent requests also ensure completion, priority requests have two essential advantages over them. First, priority requests are as efficient as transient requests because they do not need to override the performance policy. Second, their storage requirements can be drastically reduced without causing a serious degradation to the global system performance.

4.1 General Working Scheme

When a processor detects possible starvation, the priority request mechanism is applied as follows:

1) The starved processor composes and sends a priority request.
2) The priority requests are broadcast through ordered paths. As a result, all nodes see all priority requests in the same order.
3) At their arrival, priority requests are stored in a table. If the table is full and the request cannot be inserted, the incoming request is rejected and its storage is postponed until a table entry is freed.
4) Nodes serve the stored priority requests in the same order as they arrived.
5) The issuers of the stored priority requests inform of their completion to the issuers of the rejected priority requests (if any), by a resending notification.
6) When the issuer of a rejected request receives a resending notification, it resends its priority request, replacing the completed one.
7) Nodes continue to serve the stored priority requests until completing all of them.

Figure 3 shows a sketch of how the proposed mechanism works. At time 0, *P0*, *P1*, and *P2* broadcast a priority request. At time 1, *P0*'s request is received and, since tables are empty, it is stored. At time 2, *P1*'s request is received and, since tables are full, it is rejected. Besides, given that the last received request was issued by *P0*, it remembers that *P1*'s request has been rejected (arrow from *P0* to *P1*). Similarly, at time 3, *P2*'s request is received and rejected and *P1* remembers it. The stored



(a) Mesh (b) MIN

Fig. 4. Examples of ordered paths in (a) a mesh and (b) a MIN. The shaded switches (R) represent the root

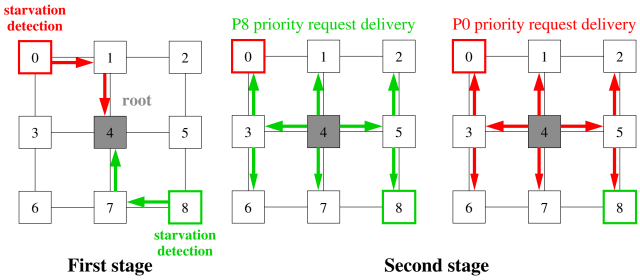


Fig. 5. Examples of how two priority requests are ordered

request (*P0*'s) is served and, when it completes (time 4), its issuer informs to the issuer of the next rejected request (*P1*). When *P1* receives the resending notification (time 5), it resends its priority request, which implicitly informs of the completion of the stored one. Thus, on its arrival, the resent request replaces the stored one. The same happens for the *P2*'s request after completing *P1*'s.

4.2 Ordered Paths

Putting messages in order by bus-like interconnects or directories may entail serious problems that token protocols seek to avoid. Thus, to provide a global order for priority requests without trusting on those methods, in this work we propose to use routing algorithms based on *ordered paths*. An ordered path is just a sequence of switches and links that comprises (at least) all the switches which the system nodes are connected to. The first switch of this sequence is usually referred to as the *root* switch and it acts as ordering point. The main property of ordered paths is that they provide a unique route to reach each node. Therefore, assuming a FIFO transmission, it is not possible that messages routed through the same ordered path overtake each other. The way to define the ordered paths depends on the network topology. For example, in this work we assume both a mesh and a Multistage Interconnection Network (MIN) [17]. On these networks, an efficient solution can be the spanning trees. Figure 4 shows how a spanning tree is used to make an ordered path. The switch *R* represents the root and the arrows represent the path that the messages follow from the root.

Since the issuer of a priority request may not be directly connected to the root switch of the ordered path, the routing of priority requests is divided in two stages. In the first stage, they go from their senders to the root switch without being delivered to the nodes connected to the visited switches. In the second stage, the priority requests follow the ordered path from the root and, unlike the first stage, they are delivered to all the nodes connected to the visited switches. Thus, all the priority requests sent through the same ordered path are delivered in the same order. Figure 5 illustrates an example of how two priority requests are put in order by using this method. *P0* and *P8* detect possible starvation and send two priority requests. Both priority requests are first routed towards the root (switch 4). *P8*'s request is the first to reach the root and, in the second stage, it is broadcast through the ordered path. After routing the *P8*'s request, the root routes the *P0*'s. As they cannot overtake each other along the path and the path to each node is unique, all the system nodes receive first *P8*'s request and then *P0*'s. Note that, if there is a single ordered path for all the priority requests, the root switch may become a bottleneck in medium and large systems. To avoid it, since we only need to maintain the order between the priority requests upon the same block (but not between priority requests upon different blocks), several ordered paths can be used.

The single difference between the path followed by priority requests and the rest of messages is that the formers follow non-minimal paths (a two-stage routing scheme through the root switch). Priority requests are routed along each stage by using the same routing algorithm as that used by the rest of messages (e-cube in meshes or up/down [18] in MINs). However, to avoid introducing cyclic channel dependencies that could lead to deadlock, in some topologies (v.g. meshes) it may be required to carry out a virtual channel transition immediately after traversing the root switch. Hence, an additional virtual channel may be required to route priority requests in some network topologies.

Notice the differences between using ordered paths to provide global order and the traditional methods. On the one hand, when directories are used to provide global order, the directories consume and process the received requests, which takes long time. In case of ordered paths, the root switch is only used to route the requests (it does not consume them) and, therefore, it hardly penalizes the latency. On the other hand, unlike totally-ordered interconnects and logical buses (hierarchical switches), which usually lack scalability, the ordered paths can be implemented in any network topology.

4.3 Storage Structures

The proposed mechanism requires some data structures. Namely, a *priority request table* to remember the outstanding priority requests, an *Ack* register to know the service sequence of the rejected requests, and a *Counter* to determine the value of the *Ack* register.

At their arrival, priority requests are remembered in tables, at least, until being completed. To this end, each processor and memory controller holds a *priority request table*. Each entry stores information about a specific priority request, which comprises the following fields: **valid bit** (1 bit), which indicates whether the entry is valid; **issuer** (2 bytes), which is the issuer's identifier (processor number); **address** (5 bytes), that is the physical address of the requested memory block; **identifier** (2 bytes), which is the priority level (arrival order) of the request; **operation** (1 bit), which is used to distinguish between read and write requests; **state** (1 bit), that indicates if the priority request was completely served or is pending; and *used* bit, which indicates whether the information held by that entry can be included in a priority request or not (we will see this in more detail later).

When a node receives a priority request, it remembers the request by inserting the corresponding information in the table: the *valid bit* is set; the requester identifier is inserted in the *issuer* field; the requested memory block address, in the *address* field; the *operation* bit is set according to the request type (read/write); the *state* field is set to pending; and the *used* bit is set to not used. Besides, the received request is assigned an *identifier*, which will be used to (1) unequivocally identify it and (2) know the order in which it was received. Since this identifier is related to the arrival order, then the lower the request identifier is, the higher priority the request has. To determine the identifier of every request, each node uses a local counter. Given that all the priority requests are received in the same order, all nodes will assign to them the same value. The sequence numbers provided by the counters are large enough, in practice, so that priority requests always have unambiguous identifiers. 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 completing the subsequent 65535 priority requests. 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, the processors would temporarily pause the generation of new priority requests until the delayed one was completed and removed. To this end, nodes use a strategy based upon algorithms for handling finite sequence numbers in retransmission schemes.

Since the priority request table can have multiple entries that contain the same address, the entry for the request with the lowest identifier and marked as pending corresponds to the request with the highest priority. Notice that the entries that contain different addresses refer to requests upon different memory blocks. As they do not contend for the same block, they cannot cause protocols races and their service is independent.

The number of entries of the priority request table can be configured between a minimum (one entry) and a maximum (as many entries as the maximum number of outstanding priority requests per processor × number of processors). When tables are the maximum size, they will be able to contain all the outstanding priority requests. Although this is the most effective option, it has the problem that the table size depends on the system size and this may jeopardize the scalability of the whole protocol (in terms of storage requirements, latency, and power due to the required associative access). This can be avoided by using tables with less entries. However, with those tables, all the outstanding priority requests may not be able to be stored simultaneously. In that situation, only some priority requests will be stored and the other will be rejected. Once a stored request completes, its issuer will inform of the completion to the issuer of a rejected priority request (using a point-to-point message called *resending notification*). To be aware of the existence of rejected requests and to know the processor to which a resending notification must be sent, nodes use a register called **Ack**. To estimate the value of this register, each node will additionally require a **Counter** register. The value of *Counter* ranges from 0 to N - 1, being N the number of table entries.

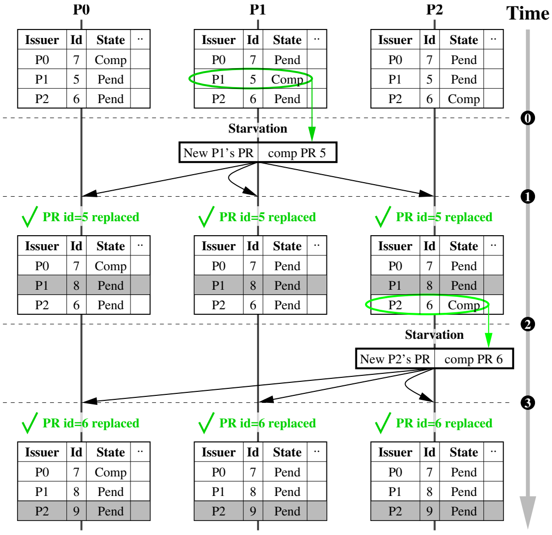


Fig. 6. Replacement of the completed priority requests with new outstanding priority requests

To sum up, the minimum requirements of priority requests are a single-entry table and as many *Ack* and *Counter* registers as maximum number of simultaneous requests per node. Hence, in terms of storage, the proposed mechanism significantly reduces the requirements of persistent requests. Furthermore, since the table can be configured to use just a few entries, the associative lookup that it requires would not be a problem.

4.4 Ensuring the Priority Request Storage

Every outstanding priority request must be remembered in the tables so that it can be served. Given that the proposed mechanism does not use explicit messages to remove the completed requests from tables, some outstanding requests might find the tables full of completed requests at their arrival, which would prevent

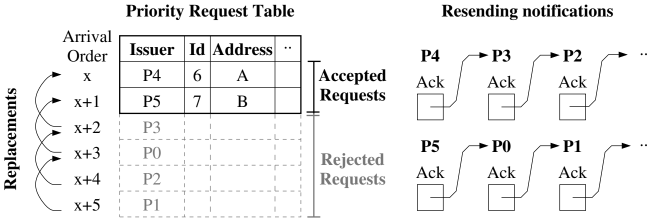


Fig. 7. Policy followed to ensure the storage of all priority requests. 2-entry tables are assumed (N = 2)

their storage. In addition, the storage of outstanding requests could also fail due to the fact that tables can have less entries than the total number of simultaneous outstanding priority requests. As a result, if each processor issues a priority request, they will not be able to be simultaneously stored. Next, we explain how the mechanism deals with these situations. However, for clarity, we address it separately in two sections.

4.4.1 Removal of Completed Priority Requests

When tables are the maximum size, each processor is associated with a fixed number of table entries (as many entries as the maximum number of simultaneous requests that it can issue). The priority requests issued by certain processor can only be stored in one of its associated entries. Given that one processor can be associated with more than one entry, when it issues a new priority request, it must indicate in which entry (among those associated to it) it should be stored. To this end, the header of the priority request messages includes a field called *completed PR*. On a priority request sending, the issuer includes in the *completed PR* field the identifier of one of its stored requests that is completed. Thus, on a priority request arrival, processors certainly know the entry where the incoming request must be stored (i.e., the entry where the completed request is stored). Doing so, nodes are implicitly informed of the priority request completion. Notice that this strategy ensures the storage when tables are the maximum size because the priority requests issued by certain processor can only replace its own (completed) priority requests. Figure 6 illustrates an example of how this idea works. The example illustrates a 3-processor system. The number of outstanding priority requests per processor is limited to one and tables are the maximum size (3-entries). To simplify the figure (and the following ones), the table entries only comprise the fields required to follow the example. Initially, the tables are full. Note that, although all the stored requests are completed, only their issuers are aware of it (see the *State* field). At time 0, *P1* issues a new priority request. This is possible because its previous priority request (id 5) is already completed. Hence, in the *completed PR* field of the new priority request, *P1* includes the identifier 5. When the nodes receive it, they know that the completed request (id 5) can be replaced with the new one. The same happens for the *P2*'s request.

4.4.2 Use of Reduced Size Tables

When tables are not the maximum size, each processor cannot be associated with a fixed number of entries because there are less entries than the maximum number of simultaneous outstanding requests. Therefore, when a processor issues a new priority request, it may or may not have a completed priority request (issued by itself) in its table. If its table contains a completed request issued by itself, it includes the identifier of that request in the *completed PR* field, which will ensure its acceptance and storage. However, if its table does not have such information, it issues a priority request with the *completed PR* field set to *Nill*. That priority request will be rejected, but this is done just to inform of the necessity of a table entry. Thanks to this, the processor will receive a message (resending notification) informing of the completion of one of the stored requests. Now, the processor is allowed to issue again a priority request, setting the *completed PR* field to the value received in the resending notification. Thus, the resent priority request is sure to be accepted.

Let us assume an M-processor system with N-entry tables (where $M > N$). If each processor in the system issues a priority request at the same time, only a maximum of N priority requests will be stored, whereas the remaining M - N requests will be rejected. Processors serve the stored requests and, as they complete, their issuers inform of the completion to the issuers of the rejected ones. When a processor is informed of the completion of a stored request and it has a pending request that was rejected, it sends its request again, which will replace the completed one. To use a fair replacement policy, processors use the arrival order of requests. 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. Figure 7 shows an example of how the replacement policy works when 6 priority requests are received and tables only have 2 entries. The priority requests are received in the order [*P4*,*P5*,*P3*,*P0*,*P2*,*P1*]. Only the *P4*'s and *P5*'s requests are stored, whereas the others are rejected. When *P4*'s request completes, *P4* informs of its completion to the issuer of the priority request received and rejected N^{th} positions after the arrival of its own request (i.e., *P3* request). This is done by using a *resending notification*. This message includes the identifier of the completed request (*id 6*). When the issuer of the rejected request (*P3*) receives the resending notification, it realizes of the completion of the *P4*'s request and it resends its priority request, including the received information (*id 6*) in the *completed PR* field. Thus, when the resent priority request is received, it will be accepted and stored because it will replace a completed one. Notice that, to avoid races, when a processor sends the identifier of a completed priority request in a resending notification, it cannot include such an identifier in new local requests.

To remember the node (just one) to which a certain

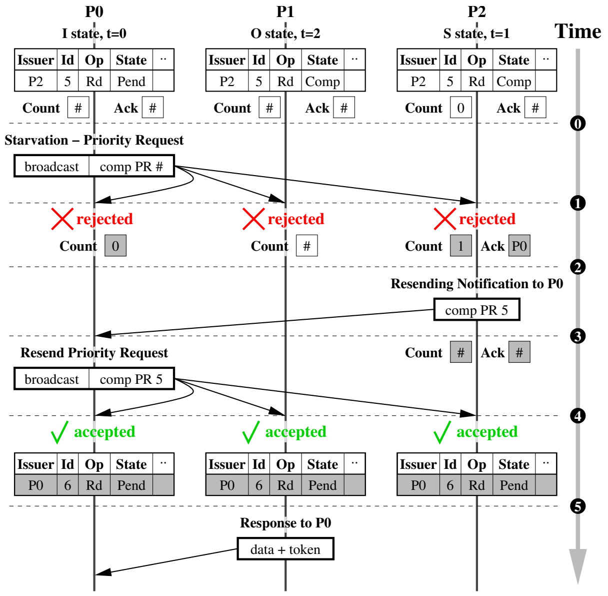


Fig. 8. Resending of rejected priority requests. The value # stands for *Nill* and *t* stands for the token count. Single-entry tables are assumed (N = 1)

processor has to send a resending notification (if any), the *Ack* register is used. Its value is estimated by using a *Counter*. Initially, *Counter* is disabled. When the issuer of a priority request receives its own priority request with the *completed PR* field set to *Nill*, the *Counter* is set to 0, indicating that the count has begun. From this moment, the *Counter* value is increased by 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 *Counter* value to overflow (a value higher than *N - 1*), the issuer of such a request is registered in *Ack* and the *Counter* is disabled. Making so, *Ack* will store the identifier of the node that issued a priority request *N* positions after the priority request issued by itself. Thus, nodes can know the priority requests that were rejected and their arrival order. However, note that the information about the arrival order of the rejected requests is distributed (instead of replicated) among some the nodes. In particular, the arrival order is remembered by linking a set of nodes by using their *Ack* registers, constituting a linked list. It will be able to have as many lists as the number of table entries, as shown in Figure 7 (two lists for a 2-entry table).

As soon as a node completes its stored priority request, if its *Ack* register holds a valid value, it sends the resending notification to the [*Ack*] processor. When the notification is sent, the value of *Ack* is set to *Nill* and the table entry holding the identifier included in the notification is marked as used (*used* bit). Doing so prevents such information from being included in another priority request issued by the node. Notice that, if a processor marks one of its priority requests as completed and that information can 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 from starving out the rejected requests and guarantees that each identifier is included *only once* in a *completed PR* field.

Figure 8 shows an example of how the whole process works. In this example, we assume three processors and single-entry tables. Initially, the tables are full. The stored request with *id 5* is completed, but only its issuer (*P2*) and the owner (*P1*) are aware of it. *P0*'s *Counter* and *Ack* registers are set to *Nill* (#). *P1* is in Owner state and its *Counter* and *Ack* registers are also set to *Nill*. However, *P2*'s *Counter* is set to 0 because the stored priority request belongs to it. At time 0, *P0* broadcasts a priority request for the same memory block. The priority request does not hold any information about the completed priority request to replace (*completed PR* 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 because tables are full. However, as *P0* rejects its own request, it initializes its *Counter* to 0. *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 exceeds the maximum count (0), it stores in its *Ack* register the issuer (*P0*) of the rejected request. Since *P2*'s priority request is completed and it holds a valid value in *Ack*, at time 2 it sends a resending notification to *P0*, notifying the completion of the priority request with *id 5*. At time 3, *P0* receives the notification and it immediately proceeds to resend its priority request, including in the *completed PR* field the *id 5*. Thus, when the resent priority request is received at time 4, all the processors accept and store it because it can replace a stored priority request. As *P1* is in Owner state, at time 5, it will serve it.

4.5 Avoiding Serving Completed Priority Requests

Since we do not use explicit messages to inform of the priority request completion, it may happen that the nodes serve priority requests that have already been completed. Although this does not cause the mechanism to fail, this is quite inefficient. To avoid it, the response messages include information of the completed requests. Let us analyze separately the response messages due to priority read and write requests. With respect to priority read requests, only the owner node is in charge of serving them. Thus, upon the reception of a priority read request, the owner processor stores it in its table, serves it by a data response, and marks it as completed. The rest of processors keep their tokens because they do not have to serve it. Therefore, to avoid serving a completed priority read request, a processor will need to know about its completion only when it becomes the owner. To this end, when a node sends the owner token in a response, it will have to indicate in it the priority read requests completed until that moment. By doing so, as soon as a node receives the owner token, it will realize of the priority read requests that have already been completed.

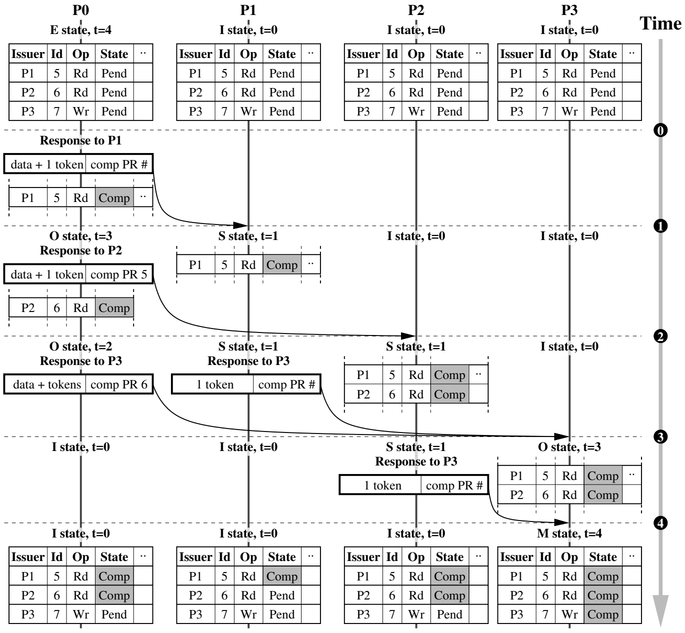


Fig. 9. Spread of the information about the priority request completion. The value # stands for *Nill* and *t* stands for the token count. 3-entry tables are assumed ($N = 3$)

In case of priority write requests, all the processors holding tokens will have to send them. Unlike 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. Therefore, this node has to inform of its request completion each time it sends a response message to another processor.

From this analysis, we conclude that, to avoid serving again the completed priority requests, all the response messages sent by the owner should indicate the priority requests completed until that moment. To this end, the header of data response messages includes a field (*completed PR*) that indicates **just the last completed priority request**. Since priority requests for the same memory block are completed in order, it implicitly indicates that all the previous priority requests are completed too. To set the value of this field, before sending a response, the owner looks in its table for the priority request (upon the same memory block as the token/s included in the response) marked as completed and with the highest identifier, which will be included in the *completed PR* field[1]. On a response reception, processors first mark as completed all the entries (upon the same memory block as the response) holding a priority request with an identifier lower than or equal to *completed PR*, then use the tokens, and finally forward them to the pending priority requests that require them (if any). Note that, nodes do not have to wait for the information included in the response messages (they only have to wait for the tokens) to serve the next stored outstanding priority request, which speeds up the race resolution. On the contrary, the deactivation messages used by persistent requests contribute to increase the miss latency since nodes cannot serve subsequent requests until receiving the deactivation.

Figure 9 shows an example of how the information about completed requests is spread by the responses. In the depicted scenario, *P1*, *P2*, and *P3* have sent a priority request. *P1*'s request is the highest priority request. *P0* (the owner) serves it by a data response and marks it as completed. Next, at time 1, it proceeds to serve the highest priority request at that moment (*P2*'s) and marks it as completed too, indicating in the response the last completed request (*id 5*). Finally, at time 2, *P0* serves the last stored request (*id 7*), indicating in the response the last completed request (*id 6*). As it is a write request, *P0* does not mark it as completed. When *P3* receives the response, it realizes that the priority request with *id 6* and all the previous ones (the one with *id 5*) are already completed, updating its table accordingly. This prevents *P3* (the new owner) from serving again *P1*'s and *P2*'s requests. Notice that although *P0*, *P1*, and *P2* maintain some completed requests marked as pending, it is not a problem because none of these nodes is able to continue to serve those priority requests until they receive the token owner from *P3*.

4.6 Performance Policy

In this work we assumed a TokenB policy adapted to the priority request mechanism. It proposes:

- On a cache miss occurrence, a transient request is broadcast to all the processors and the home memory module.
- If after twice the processor's average miss latency the transient request has not been completed, a priority request is broadcast to all the processors and memory modules.
- Processors and memory modules respond to both transient and priority requests as they would do in a traditional MOESI protocol. Since deactivation messages are not needed, several transient/priority requests can be served concurrently.

Figure 10(a) shows the state transition diagram due to the reception of transient/priority requests. Together with each transition, it is indicated (in gray) whether the processor has to serve the request by sending a response message. *T/P Read (write) R* refers to transient/priority read (write) requests. The dotted arrows indicate that from that state, the node cannot be completely sure about the completion of the stored priority request and, therefore, it will remain in the table marked as pending. The two additional *Sp* and *Ip* states are equivalent to *S* and *I*, respectively. The difference is that those states (marked

[1] Obviously, if the owner node cannot find such information in its table, it will be because it has already been removed from all the tables and, therefore, it is not longer necessary to send it again.

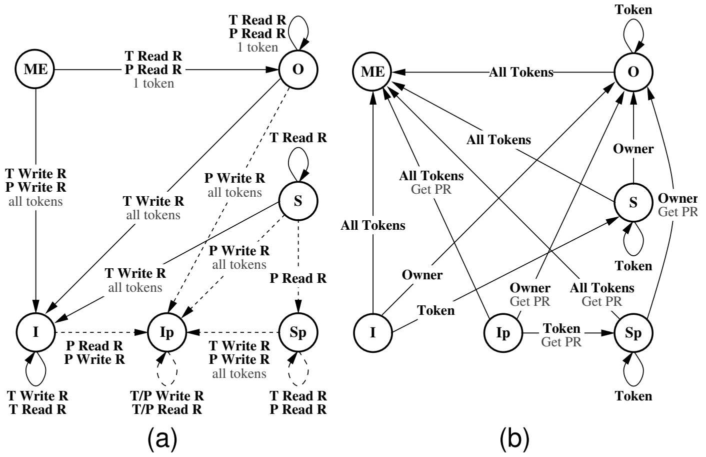


Fig. 10. State transition diagram due to (a) request reception and (b) response reception. T and P stands for transient and priority, respectively

with the p subscript) indicate that there is, at least, one pending priority request in the table. Note that, from the *Sp* state a transient write request could be served. This is because transient and priority requests can coexist and their service can be performed concurrently. Notice that, servicing a transient write request upon a block in *Sp* does not threaten the forward progress because that state indicates that there are only priority read requests, but not priority write requests. Therefore, since a node in *Sp* does not have to serve read requests, the tokens can be sent to other requests that require them (e.g., transient write requests).

Figure 10(b) depicts the state transition diagram due to the reception of response messages. Note that, when a node is in a state marked with the p subscript and it receives some tokens, (1) it transitions to another state, (2) gets from its table the highest priority request that is pending (*Get PR*), and (3) serves it as indicated in Figure 10(a) (if appropriate). It will continue to serve the pending priority requests until completing all of them or lacking requested tokens. In addition, if a processor is in Ip/Sp, it transitions to I/S when all the pending priority requests for that block are marked as completed (as a result of updating the table by using the information received in the *completed PR* field of a message). These transitions are not shown in the figures to simplify them.

4.7 Optimizing Rejected Priority Requests

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). According to this, with small tables, the majority of the starved requests will require two priority requests to be resolved. Though this is correct, it entails to flood the network twice. To alleviate this problem, when a processor holds all the tokens requested by a priority request that is going to be rejected, it will proceed to serve it immediately. Hence, a single priority request may be enough to solve the starvation. Note that, in this case, the priority request has been served without being stored, in a similar way to transient requests. This simple strategy lets the network traffic be alleviated due to the reduction of broadcast messages. Besides, the average latency of resolving starvation will decrease (as the response messages may be received sooner), which in turn will contribute to improve the overall performance.

TABLE 2
System parameters

split L1 I&D caches	32 KB, 4-way, 1 cycle
unified L2 caches	8 MB, 4-way, 6 cycles
cache block size	64 bytes
DRAM latency	80 cycles (4 GB)
memory controllers	6 cycles
network link latency	3 cycles
switch cross latency	1 cycle
network routing latency	1 cycle
message header size	8/10 bytes*

TABLE 3
Applications and parameters

FFT	65,536 complex data points
Cholesky	input file tk29.O
Radix	256k keys, radix of 1024
Barnes	16384 particles
LU	512×512 matrix, block size of 16
FMM	256 particles
Ocean	258 × 258 ocean
Volrend	head_scaledown2

5 Experimental Results

5.1 System Configuration and Benchmarks

We evaluate our proposal with full-system simulation using Virtutech Simics [19] extended with the Wisconsin GEMS toolset [20] which enables detailed simulation of multiprocessor systems. Additionally, we have extended it with a multiprocessor interconnection network simulator developed by the Parallel Architecture Group [21]. We simulate a single-threaded 32-processor Sparc v9 system. Processors are in-order, single-issue processors. Each node includes a processor, split L1 caches, unified L2 cache, and coherence protocol controllers. Table 2 shows the system parameters, which are similar to those chosen in [22]. Note that the latency of memory (80 cycles) is quite optimistic, but we just assumed this value to speed up the simulations. The message header is 8 bytes when it does not include the *completed PR* field, whereas it is 10 bytes when it does includes it.

The used workloads consist of the applications from the SPLASH 2 suite shown in Table 3. We have chosen only these applications because of the difficulties to simulate all the applications from the SPLASH suite (due

TABLE 4
Table Size (in Bytes)

Pers	Prio32	Prio16	Prio08	Prio04	Prio02	Prio01
256	320	160	80	40	20	10

to their time requirements). 20 simulations were run for each application. The points of the figures shown in the next section were obtained by averaging the results for each application as described in [23].

We evaluate the referred system assuming that processors are connected through a mesh and a MIN network with the perfect-shuffle permutation. However, since the results for these networks are very similar, in this work we only show the results for the MIN interconnect. Although only two virtual channels are required for the MIN, in the simulations we assumed three virtual channels for increasing the performance. Response messages and resending notifications use the first virtual channel, transient requests use the second one, and the third virtual channel is for persistent/priority requests. Priority requests use a single ordered path. The size of the network messages (transient requests, responses, and persistent/priority requests) accounts for the additional information that they include.

We compare the persistent request mechanism implementing a distributed arbitration scheme, with point-to-point ordering, and a fixed-priority scheme against the priority request mechanism with different tables sizes. Note that the assumed persistent request mechanism incorporates the optimizations suggested in different works [6], [9]. Therefore, it is the most efficient version of the mechanism. We assume only one simultaneous outstanding persistent/priority request per processor like done in other works [8].

5.2 Performance Evaluation

We compare Token Coherence using the persistent request mechanism (*Persistent R*) against Token Coherence using the priority request mechanism with tables proportional to the system size, that is, with 32-entry tables (*Priority R*). In addition, we also compare those mechanisms against the priority request mechanism using 16-entry tables (*Priority-16*), 8-entry tables (*Priority-8*), 4-entry tables (*Priority-4*), 2-entry tables (*Priority-2*), and single-entry tables (*Priority-1*). Table 4 shows the size (in bytes) of the tables for each option. The comparison is performed in terms of number of requests suffering starvation, number of control messages (data-less and tokens-less) used to manage starvation, network traffic, latency of resolving a starved request, and runtime. Besides, it is also shown how the runtime scales depending on the system size. The results shown in the following figures are normalized to the values obtained for Token Coherence using persistent requests (*Persistent R*).

Figure 11 illustrates the normalized number of requests suffering starvation. As shown, when Token Coherence uses the priority request mechanism (*Priority R*), the number of starved requests reduces significantly (10% on average) because (1) tokens are always managed efficiently since the performance policy is not overridden, (2) transient requests can be served simultaneously with priority requests, and (3) it is not necessary to wait for the reception of acknowledgments to complete the service of priority requests. When the size of the priority request tables lowers, the number of starved requests diminishes even more, reaching 25% of reduction on 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 requests 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 12 shows the number of control messages used to manage the generated starvation situations. These messages include activation/deactivation messages in *Persistent R*, priority requests in *Priority R*, and priority requests and resending notifications when tables with reduced size are used. As depicted, *Priority R* needs significantly less control messages (up to 60% on average) than *Persistent R* to manage starvation situations. This happens because, first, *Priority R* generates less starvation situations and, second, each starved request requires less control messages: while *Priority R* uses one message per starved request (one priority request), *Persistent R* uses two messages (one activation message and one deactivation message). 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 *Persistent 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% on average with respect to *Persistent R*. However, note that the number of broadcast messages, which are the most harmful messages for the overall performance, is still lower (between 20% and 35%) than the total number of broadcast messages generated by *Persistent 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 *Persistent R*. Besides, in this last case, the number of broadcast messages is 45% lower than that generated by *Persistent R*.

The normalized total traffic (in packets) generated by Token Coherence is depicted in Figure 13. *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 fact that less cache

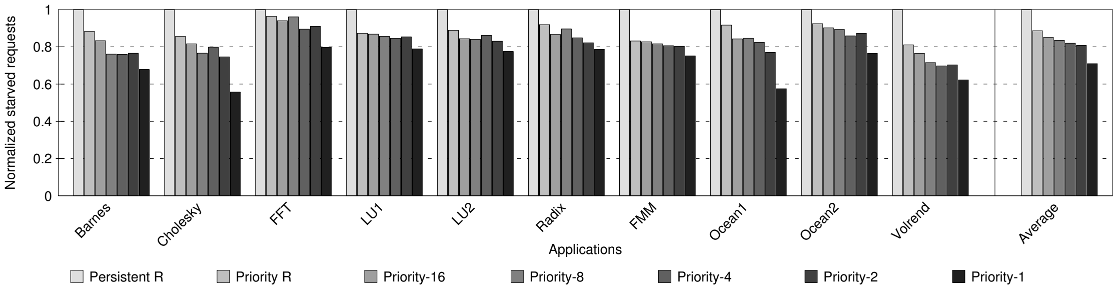


Fig. 11. Normalized number of starved requests

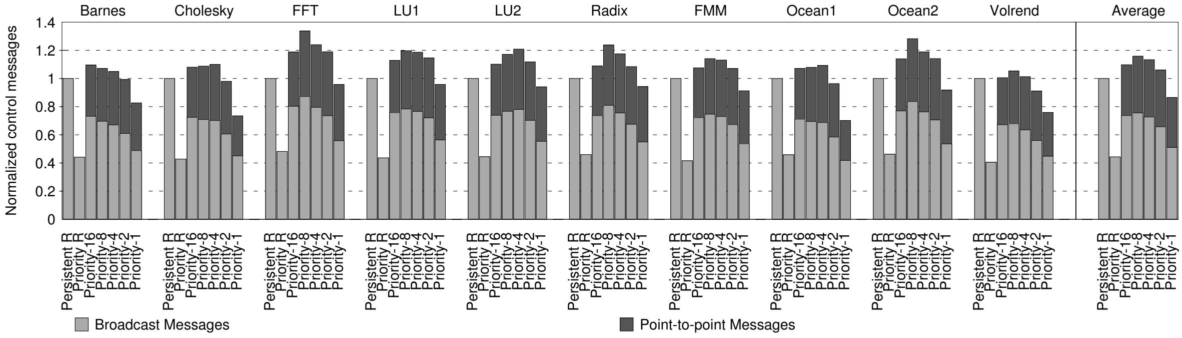


Fig. 12. Normalized number of starvation control messages

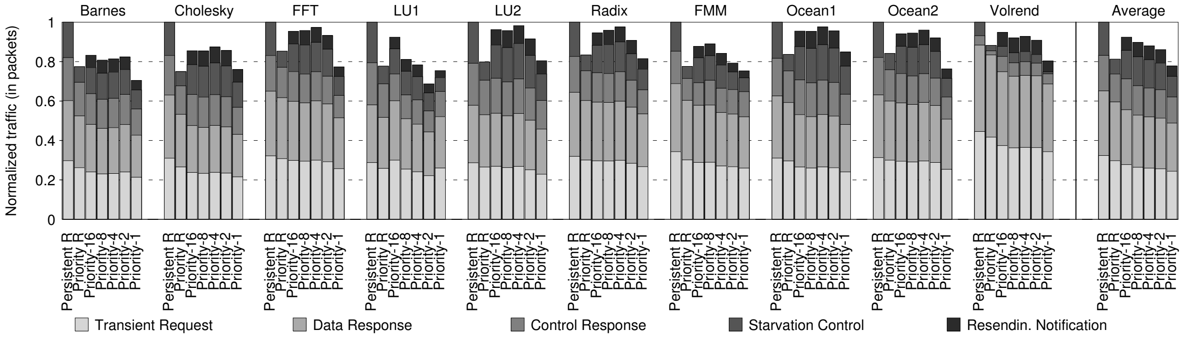


Fig. 13. Normalized injected traffic

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. Furthermore, 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 high reduction of starved requests, the overall traffic is more or less similar to that generated by *Priority R*.

Figure 14 shows the 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 14, *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 *Persistent 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

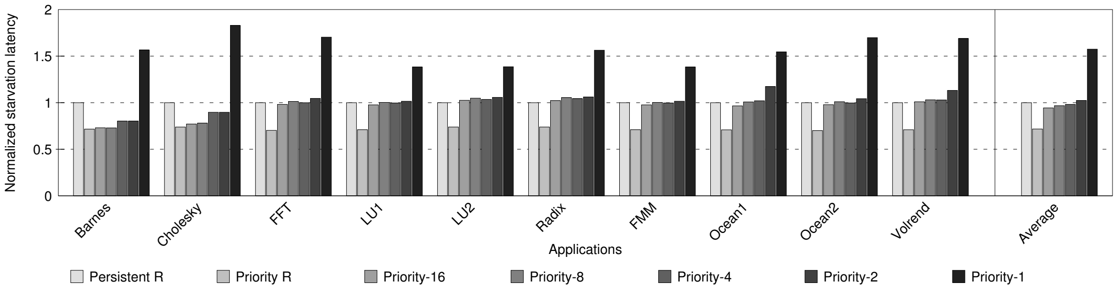


Fig. 14. Normalized starvation latency

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 4.7). This is possible thanks to the fact that priority requests are ordered messages, which usually suffices to solve most of the protocol races.

Figure 15 illustrates the normalized runtime of the applications. As shown, *Priority R* reduces the runtime between 8 % and 20 % (on average, more than 10 % in a MIN and more than 8 % in a mesh). 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, even though the table are reduced to a single entry, the runtime is even lower than that of the most 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).

Figure 16 depicts how the performance of priority requests scales according to the system size. Because of space reasons, the figure only shows the average runtime of the applications. As shown, although in small systems (4, 8, and 16 processors) the starvation prevention mechanism does not significantly affect the runtime of applications, in medium systems (32 and 64 processors) it does. Thus, as the size of the system increases, the differences between priority requests and persistent requests are more and more significant, reaching 22% of reduction on average in a 64-processor system.

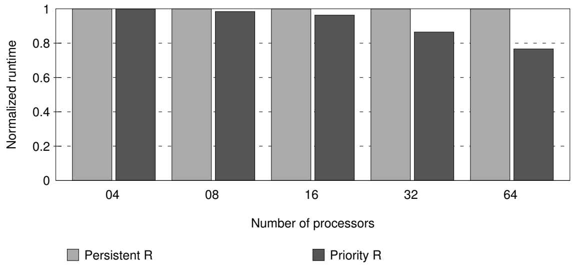


Fig. 16. Normalized runtime depending on the system size when tables are the maximum size

Finally, Figure 17 shows the runtime of the applications depending on the system size when using 2-entry tables. In this case, the runtime is normalized to that obtained for the priority request mechanism using as many entries as number of processors. As you can observe in the figure, when 2-entry tables are used, the runtime of the applications is practically the same to that obtained when using tables proportional to the system size. Only in the 32-processor system, a slight increase of the runtime (less than 2%) can be appreciated. This figure gives an idea of how scalable (in runtime terms) the proposed strategy is.

6 Conclusions

We have proposed a new starvation prevention mechanism, named priority requests, for Token Coherence. Unlike persistent requests, starvation situations are resolved without requiring explicit acknowledgments and without overriding the component that provides high performance. As a consequence, Token Coherence using the priority request mechanism solves starvation situations faster and generates less network traffic. This fact contributes to reduce the execution time of applications, specially when races are common (like in medium and large systems).

Additionally, the priority requests can improve the scalability of Token Coherence by decoupling its storage requirements from the system size, while still maintaining the overall performance. Despite the fact that the table size can be reduced to 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.

Acknowledgments

The authors thank the anonymous reviewers for their useful and valuable comments to improve the quality

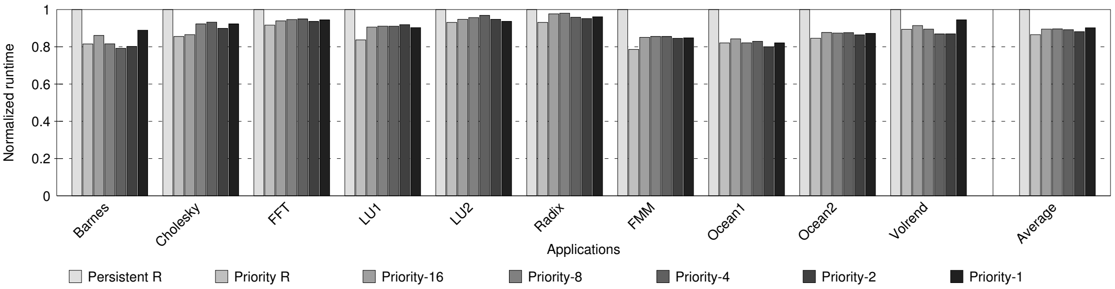


Fig. 15. Normalized runtime

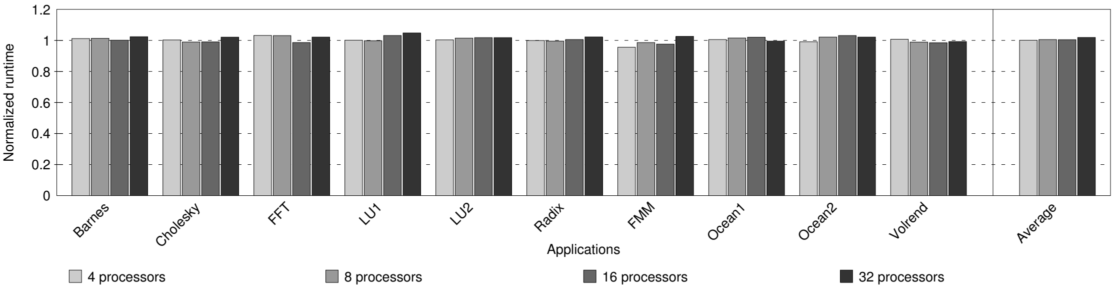


Fig. 17. Runtime (normalized to priority requests using tables with the maximum size) depending on the system size when using 2-entry tables

of this paper. This work was partially supported by the Spanish MEC and MICINN, as well as European Commission FEDER funds, under Grants CSD2006-00046 and TIN2009-14475-C04-01. Antonio Robles is taking a sabbatical granted by the Universidad Politécnica de Valencia for updating his teaching and research activities.

REFERENCES

[1] Poonacha Kongetira et al. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, 2005.

[2] H. Q. Le et al. IBM POWER6 microarchitecture. *IBM J. Res. Dev.*, 51(6):639–662, 2007.

[3] Owen Liu. AMD technology: power, performance and the future. *CHINA HPC '07: Proceedings of the 2007 Asian technology information program's (ATIP's) 3rd workshop on High performance computing in China*, pages 89–94, 2007.

[4] J. A. Kahle et al. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[5] James R. Goodman. Using cache memory to reduce processor-memory traffic. *ISCA '83: Proc. of the 10th annual inter. symp. on Computer architecture*, pages 124–131, 1983.

[6] 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. *HPCA '05: Proceedings of the 11th Int. Symp. on High-Performance Computer Architecture*, pages 328–339, 2005.

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

[8] Milo M. K. Martin et al. Token Coherence: decoupling performance and correctness. *ISCA '03: Proc. of the 30th annual inter. symp. on Computer architecture*, pages 182–193, 2003.

[9] Milo M. K. Martin. Token Coherence. *The University of Wisconsin - Madison*, 2003. Supervisor-Mark D. Hill.

[10] Blas Cuesta, Antonio Robles, and Jose Duato. An effective starvation avoidance mechanism to enhance the Token Coherence protocol. *PDP '07: Proc. of the 15th Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing*, pages 47–54, 2007.

[11] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. *SIGARCH Comput. Archit. News*, 14(2):414–423, 1986.

[12] Michael R. Marty and Mark D. Hill. Coherence ordering for ring-based chip multiprocessors. *MICRO 39: Proc. of the 39th Annual IEEE/ACM Int. Symp. on Microarchitecture*, pages 309–320, 2006.

[13] Arun Raghavan et al. Token tenure: Patching token counting using directory-based cache coherence. *MICRO 41: Proc. of the 41th Int. Symp. on Microarchitecture*, 2008.

[14] N. Agarwal et al. In-network snoop ordering (inso): Snoopy coherence on unordered interconnects. *International Symposium on High Performance Computer Architecture (HPCA)*, February 2009.

[15] Blas Cuesta, Antonio Robles, and José Duato. Switch-based packing technique for improving Token Coherence scalability. *PDCAT'08: Parallel and Distributed Computing, Applications and Technologies*, pages 83–90, 2008.

[16] Blas Cuesta, Antonio Robles Martinez, and Jose Francisco Duato Marin. Improving Token Coherence by multicast coherence messages. *PDP '08: Proc. of the 16th Euromicro Conf. on Parallel, Distributed and Network-Based Processing)*, pages 269–273, 2008.

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

[18] Herbert Sullivan and T R Bashkow. A large scale, homogeneous, fully distributed parallel machine, i. *SIGARCH Comput. Archit. News*, 5(7):105–117, 1977.

[19] Peter S. Magnusson et al. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002. IEEE Computer Society Press.

[20] Milo M. K. Martin et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[21] GAP - Parallel Architecture Group. *http://www.gap.upv.es./*.

[22] Jayaram Bobba et al. Performance pathologies in hardware transactional memory. *SIGARCH Comput. Archit. News*, 35(2):81–91, 2007.

[23] Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. *HPCA '03: Proc. of the 9th Int. Symp. on High-Performance Computer Architecture*, page 7, 2003.

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*.