Int J Parallel Prog (2018) 46:1110–1138
https://doi.org/10.1007/s10766-018-0557-y

MOSAIC: A Scalable Coherence Protocol

Lucia G. Menezo[1] · Valentin Puente[1] · Pablo Abad[1] · Jose-Angel Gregorio[1]

Received: 29 August 2016 / Accepted: 19 January 2018 / Published online: 29 January 2018
© Springer Science+Business Media, LLC, part of Springer Nature 2018

Abstract The coherence protocol presented in this work, denoted MOSAIC, introduces a new approach to face the challenges of complex multilevel cache hierarchies in future many-core systems. The essential aspect of the proposal is to eliminate the condition of inclusiveness through the different levels of the memory hierarchy while maintaining the complexity of the protocol limited. Cost reduction decisions taken to reduce this complexity may introduce artificial inefficiencies in the on-chip cache hierarchy, especially when the number of cores and private cache size is large. Our approach trades area and complexity for on-chip bandwidth, employing an integrated broadcast mechanism in a directory structure. In energy terms, the protocol scales like a conventional directory coherence protocol, but relaxes the shared information inclusiveness. This allows the performance implications of directory size and associativity reduction to be overcome. As it is even simpler than a conventional directory, the results of our evaluation show that the approach is quite insensitive, in terms of performance and energy expenditure, to the size and associativity of the directory.

Keywords Multicore computer architecture · Memory hierarchy · Cache coherence protocol

✉ Pablo Abad
abadp@unican.es

Lucia G. Menezo
lucia.gregorio@unican.es

Valentin Puente
vpuente@unican.es

Jose-Angel Gregorio
monaster@unican.es

[1] University of Cantabria, Santander, Spain

1 Introduction

The new paradigm of multiple cores inside a chip presents some new challenges. Among them, we can consider the so-called bandwidth-wall [1] as one of the most crucial. This obstacle is due to the limited growth in the number of pins and the operating frequency due to physical and packaging cost restrictions. Off-chip communication necessities grow as the number of cores and their complexity increase. However, the available off-chip bandwidth does not increase at the same rate, becoming a bottleneck in the whole CMP. Some studies [2] predict that this problem will limit the number of cores that can be introduced inside the chip. Fortunately, there is a wide variety of solutions that are able to mitigate the problem. Among them, the one that appears to have most benefits is the introduction of large amounts of memory inside the chip.

However, the efficient organization and management of large amounts of memory associated with each of the cores is not a straightforward task. There is a consensus among computer architects which assigns some cache memory to each of the cores in a stepped way at different levels but, from the performance point of view, there cannot be "steps" with excessive difference in capacity among them [3]. There is much less unanimity about the distribution of the last level cache (LLC) and its characteristics. Some distribute it as a private cache and others design it to be shared among some or all the cores in the chip. This decision will have a significant effect on the CMP usage. In almost all the CMPs implemented so far (Bulldozer [4], Haswell [5], Sparc T5 [6]), the LLC is shared among the cores in the chip because it seems that this improves memory utilization. Other companies are tending to maintain the LLC as local caches for each core although all the banks are used as victim cache by the rest [7]. Although the most common number of levels used nowadays is three [7,8], there are already some new commercial systems which include more levels, such as the IBM z196 [9] or IBM Power 8 [10] which includes a fourth. As soon as the technology enables it, with mechanisms such as 3D stacking [11], more levels will be introduced inside the chip.

In any case, from the moment there are multiple copies of the same block in the system, coherence has to be enforced. Therefore, it is important to decide how it will be managed: via hardware and/or via software. There are numerous works analyzing the advantages of exposing to the programmer-compiler the capability of handling the data coherence of the blocks allocated in the private caches [12,13]. However, most of those studies are focused on performance comparison, i.e. execution time, and only consider a very specific type of applications. When taking into account general purpose applications, with the large amounts of memory in a multi-level hierarchy, coherence management is not a trivial task. For this reason, programming parallel applications without the hardware support to do this might hinder the productivity of programmers because they will have to pay too much attention to this duty. Although not unanimously, a large part of the industry and academia believes that the overall future chip general-purpose multiprocessors (CMP) will have some sort of hardware mechanism for cache coherence. Therefore, as in Martin et al.'s discussion in [14], we also believe that for the next few years, data coherence maintenance should be guaranteed by hardware and the main target of this work has been the search for efficient strategies to achieve this.

The responsibility of the coherence protocol is to ensure that all the potential copies of a memory block scattered throughout different caches are coherent. A large number of cores and complex cache hierarchies might increase coherence protocol responsiveness. On the one hand, having a large number of cores in the chip makes it unfeasible to rely on broadcast-based coherence protocols. Although in current commercial CMPs this is the predominantly used approach [4,7,15,16], it has foreseeable difficulties to achieve success with larger numbers of cores in the system, due to their higher energy requirements [17]. On the other hand, complex cache hierarchies increase the likelihood of having multiple copies of shared blocks scattered throughout private levels, which is challenging for pure directory-based coherence protocols. The private section of the cache hierarchy in current systems is quite large but it will be greater in the medium term as the memory wall effects become more relevant. Therefore, the amount of information required by directory protocols will increase.

Under the previously depicted context, we have developed a coherence protocol suitable for confronting the problem comprehensively. Mosaic [18] is constructed on top of a conventional directory protocol [19], but instead of using inclusiveness to guarantee system correctness, Mosaic will use a token coherence correctness substrate [20]. The proposal inherits the Token Coherence protocols' simplicity, their lack of precise sharing knowledge and the power efficiency of conventional Directory protocols. Additionally, Mosaic circumvents not only most of the multicast traffic of Token Coherence, but also the inelegant starvation avoidance mechanisms needed due to the lack of serialization points.

Although from a performance and cost point of view non-inclusiveness is desired, the common assumption is that inclusiveness is inescapable to keep coherence protocol complexity manageable [21,22]. As a matter of fact, Mosaic is simpler than a plain directory coherence protocol and any block stored in private caches may not be tracked (i.e. no entry will necessarily be allocated in the directory). The protocol is engineered to reconstruct the entry under demand (i.e. if a core misses at its private cache levels for an untracked cache block that is stored in other cores' private cache). Our proposal is utilizable using in-cache or sparse directories [19]. We will show that even with extremely small directories and/or associativity, it is possible to sustain the performance and energy consumption of the system. The key aspect of this remarkable achievement is that token counting allows the data stored in LLC to perform directory entry reconstructions without any extra traffic.

As the reader may remember, token coherence [20] is based on assigning a fixed number of tokens per cache block and requires at least one token to read and all of them to write. The most common case is that the data accessed will be private so LLC will have data with all the tokens. Taking this into account, LLC will, in most cases, have all the tokens, making it unnecessary to broadcast a message to reconstruct the block. In this way, LLC data will serve indirectly as the most effective filter to determine whether a data block is shared or not. The new coherence protocol, Mosaic, is able to take advantage of the bandwidth availability inside the chip, increasing the scalability of directory-based systems. Thus the protocol allows the size of the directory to be scaled with the number of cores. When the number of cores increases, so it does the

number of messages, but MOSAIC compensates this through the filtering carried out by the LLC and the use of token counting. Thus the protocol is capable of reducing two of the main aspects which limit the scalability in this type of systems: directory size and interconnection network traffic.

The rest of the paper is organized as follows: Sect. 2 introduces the basic coherence protocol schemes. Section 3 explains the proposal. In Sects. 4 and 5 we introduce the evaluation methodology and the performance evaluation. Section 6 explains in-cache configuration architecture. Finally, Sect. 7 states the main conclusions of the paper.

2 Coherence Protocol Schemes and Shortcomings

2.1 Broadcast

Among the options to design coherence protocols for small to medium scale multi-cores are the broadcast-based proposals. Their main characteristic is the reduced global latency of the whole system, exploiting the high bandwidth availability inside the chip in this kind of systems. The use of scalable point-to-point interconnection networks and the scalable cache hierarchy designs implemented, such as NUCA [23], make this bandwidth profuse inside the chip. If we add to these characteristics the appearance of 3D stacked systems [11] and the utilization of low-swing links [24], bandwidth is substantially increased and the energy cost of moving data faster is reduced. Currently there are a substantial number of CMP coherence protocol proposals that share this point of view [17,20,25] and most of the ideas use broadcasting as the mechanism to overcome indirection at intermediate ordering points. The impact of the shortcomings that these protocols might have can be much less than is commonly assumed. Namely:

1. *The multicast traffic required for on-chip cache requests will increase network consumption* It is true that power consumption is affected by multicast traffic, but the final effect depends on the network characteristics. As is known, if the network has hardware support for multicast messages [26,27], their impact could be reduced because each network resource is used at most once per request. This happens because the message is only replicated when it has to go via different paths to reach its destinations. When no multicast support is included, one message will have to be sent for each of the destinations and so each resource will be used many times. According to [26], using multicast support could save up to 70% in the network Energy Delay Square Product (ED2P).
2. *Excessive network cache bandwidth consumption could increase contention and significantly increase on-chip latency* Although this may potentially ruin the rationale of snoop-based coherence protocols, a correctly dimensioned design (see for example, the configurations in Table 4) for the cache hierarchy capable of decoupling the number of cores and the on-chip cache bandwidth will prevent it. Under these circumstances, on-chip communication bandwidth will scale in proportion to core count and/or its aggressiveness.

3. *Extra cache tag lookups produced in these protocols will increase cache energy consumption* If we take into account the growing leakage in each technological advance [2], the area devoted to cache, and the substantial benefit in terms of performance obtained by snoop-based coherence, the increased tag snoop energy might be quickly amortized by the benefits in dynamic energy.

Therefore, although it is obvious that if the number of cores is very high this method may not be a suitable basic method to maintain the coherence, the previous three points allow us to conclude that the scalability is higher than many authors have estimated because of their failure to consider the previously mentioned aspects that may be essential.

2.2 Directory

Historically, directory-based coherence protocols have been used to address the scalability problem in multiprocessor systems. However, as the memory wall effects become more relevant, more on-chip cache capacity will be required and therefore large private caches will be needed. These large capacities require large storage necessities to keep all the coherence information about all the data copies in the system.

There are two main approaches to keep this information: *in-cache* and *sparse* directories [19]. For in-cache directories, each block stored in LLC has the tag and data attached to the block state and the sharers' information (sharers bit vector, pointers, etc.). The coherence controller uses this information to deal with incoming requests and having a precise knowledge of the block's sharing status is necessary to guarantee correctness. Therefore, LLC inclusiveness with the previous level caches is necessary because it is the only way to have knowledge of private level contents. For small private levels, this approach has a substantial overhead because, in order to keep track of the sharing status of a handful of data blocks, any LLC block has to have a substantial storage space reserved per block (at least $log_2 P$ bits for P processors). Additionally, the effective capacity of the LLC will be reduced since there will be progressively more blocks that will have to be dedicated to maintaining this information and fewer blocks dedicated to victim cache for private replacements.

When a sparse directory design is chosen, the total effective capacity of the LLC is recovered because directory entries are allocated under demand and therefore the overhead is proportional to the aggregate private cache level's size (and not to the LLC size). When a block arrives at the chip in response to a request, a new directory entry is allocated. It will have to include at least, the block tag, the block state and the sharer vector (or any other core representation). This entry is allocated in a separate structure from the data. In current NUCA caches, to guarantee scalability, the most extended strategy is to bank the directory throughout the chip, keeping the data and directory slices connected to the same router [28]. In most cases, the address-to-slice mapping used is statically determined by the lowest bits (closest to the byte offset) of the address.

The capacity and associativity of the directory has to be sufficient to keep private-level cache tags. In small systems [29] with small private caches and low associativity,

the coverage can be full, commonly denoted Duplicate Tag Directory. Nevertheless, for medium-to-large numbers of cores, there is no feasible way to use such large-scale associativity. Then, each entry of the directory can only maintain a subset of all the possible block tags that can be stored in private caches, inducing conflict misses. Each one of these conflict messages means an invalidation of the corresponding blocks in the private caches, with a negative effect on the performance and, what is worse, the magnitude of this negative effect strongly depends on the specific characteristics of each application. It should be noted that this inefficiency arises because of the directory inclusiveness, i.e. any block stored at any private cache level should have an entry allocated in the directory structure.

3 Proposal

Our proposal is a new coherence protocol that does not require inclusiveness to guarantee correctness and which is still considerably simpler than a traditional directory protocol. The main idea is to take advantage of the bandwidth availability inside the chip in order to avoid the necessity of inclusiveness with the aim of reducing one of the main problems that the directory approach has: the space needed to hold the coherence information for all the cache blocks stored in the private levels.

3.1 Conceptual Approach

The MOSAIC protocol is focused on reducing one of the main problems that the conventional directory approach has when dealing with a large number of processors and with large number of blocks kept in the private levels: the space needed to hold all their coherence information. The cost of the directory is proportional to the size and variety of the private levels. In order to avoid this directory constraint, MOSAIC does not evict blocks from the private levels when there is not enough space in the directory and some coherence information has to be removed to allocate new coherence lines. This means that the blocks can be kept in the private caches, although the directory is not tracking them anymore. Thus, coherence information inclusiveness is completely removed from the directory, allowing some restrictions to be eliminated when deciding the size of the directory.

Without this inclusiveness enforcement property, when a request is received and a miss occurs in the directory, it is not possible to know whether the requested data block is allocated in the off-chip memory, in the LLC and/or in any of the private levels. For this reason, the coherence protocol needs to have a special mechanism to locate all the possible copies of the requested data.

In order to be able to collect all the coherence information associated with a requested block, after any subsequent miss in the directory, an on-chip reconstruction of the directory entry is initiated. This reconstruction process starts by checking whether the requested block with all the tokens is present in the LLC. If it is not, a broadcast message is sent to all the private caches asking for information about the requested block. This process will end when all the coherence information associated with that block (i.e. the sharers of the block and their state) has been collected. By

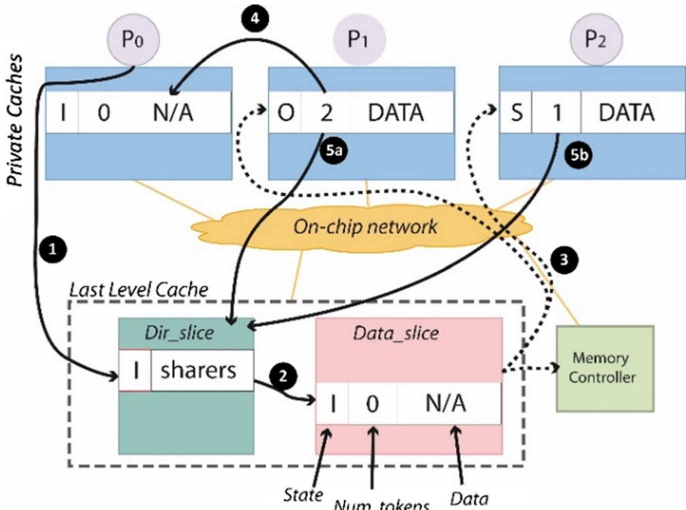

Fig. 1 Sketch of MOSAIC's concept after a request from P0 misses in the LLC and in the directory

using token counting [20], the process is kept simple and negative acknowledgements [16] are avoided. This is possible because only the private caches that have the data block with some tokens have to reply to the broadcast reconstruction message. These replies will include the number of tokens that they have, so by adding all of them the directory will know when it has finished the reconstruction process. It is important to bear in mind that the directory will not store the number of tokens each private cache has and it will only store which of them have a copy (i.e. the sharers) and which one has the owner token.

To explain the whole process in a more graphical way, Fig. 1 presents a schematic sketch of how MOSAIC behaves. The example starts with a *read request* from processor P_0 that, after missing in its private cache, sends a read request to the controller of the directory slice ❶. If the directory does not have any information about the requested data block, it checks whether it is present in the LLC ❷, and if it is not, it starts a reconstruction process broadcasting a reconstruction message looking for the data block needed ❸. During this reconstruction process, requests to the same address will be attended sequentially. The reconstruction message sent by the directory has two objectives: to build the directory sharers' information and to solve the request that initiated the whole process. For this last goal, the reconstruction message includes information about who started the reconstruction and for which type of request it did so. Thus, the corresponding private caches will be able to know when and how they have to reply to the requestor. This means that, for example, in Fig. 1, since the starting request is a read request, only the private cache holding the owner token will be in charge of solving it. For this reason, P_1 sends a copy of the data block with one of its tokens to P_0 ❹. To achieve the first goal of the reconstruction process, the directory needs to collect all the information about the requested data block. So it needs to know who is holding any tokens associated with that address and also how many of them are held, in order to know when the directory has fin-

ished collecting all the information. In Fig. 1, P_1 and P_2 send the information about their tokens to the directory ❺. All these operations (event/actions) are under the control of the coherence controllers, which are always implicitly attached to every memory module and directory structure. For a *write operation*, the reconstruction process requires that all of the sharers forward their tokens to the requestor (invalidating their copies) without sending any message to the directory. The requesting processor, after collecting all the tokens, will notify the directory with a completion message. Remember that there is a fixed number of tokens assigned to each block (usually set to the number of processors in the system) and this value is well-known by all the coherence agents of the system. In any case, once the entry is fully constructed, if the directory needs to evict it, because of lack of space in the directory after a subsequent miss, MOSAIC does not need to invalidate any of the private copies. It can replace the entry silently because it will be reconstructed if necessary.

3.2 Design Details of Mosaic

The MOSAIC coherence protocol may be used either in a sparse directory or in an in-cache directory. The only difference between them is in the coherence controller that is in charge of constructing the line, which is the element holding all the coherence information and acting as the directory. This coherence controller can be a standalone structure in the sparse design or part of the LLC controller in the in-cache design.

In both designs, sparse and in-cache directory, the main states that might be considered are the ones giving name to the coherence protocol: *Modified (M), Owner (O), Shared (S), Allocated (A), Invalid (I) and Constructing (C).* The meaning of the first three and the invalid state are well known, but the new states *A* and *C* provide the key implementation details of the MOSAIC protocol. The *C* state indicates when an entry in the directory is being constructed and the *A* state defines when a line is fully constructed with all the coherence information attached. Each of the designs, sparse and in-cache directory, has its own necessities and more importantly, its own possibilities for optimizations. For this reason, these states vary a little from one to another. Thus, while the in-cache solution needs to allocate a line whether it is being shared or not, the sparse design isolates the size of the LLC from the size of the private caches by using a separate structure. This means that some optimizations are specific for each solution, especially those reliant on the in-cache solution having the data block next to the sharing information. These two are separated in the sparse design. This opens up the possibility to react differently and to maintain some tokens in the LLC whether the block is going to be read or written. Later, we will see a specific case of this concerning the instruction fetches. This, however, obliges the replacement of the whole block when an eviction has to be done, meaning that if the LLC includes some tokens, this replacement cannot be done completely "silent" like in the sparse solution. In short, the two solutions are very similar, but their differences force us to differentiate some of their events and actions. Next, specific design details of each of them will be seen using the table-based transitions method [30].

3.2.1 Sparse Directory Specification

In a sparse design, the directory does not have data copies attached to each line. For this reason, having the M, O or S state in those entries does not apply, because the only necessary information is whether the entry is already constructed (A), being constructed (C) or invalid (I). When the directory controller is constructing a line, the block enters a transitory state, *C_S* or *C_X*, depending on whether the reconstruction process was started by a read request *(C_S)* or a write request *(C_X)*. This requirement is also mandatory for the Allocated state (A) which is divided into *A_S* or *A_X* after a *GetS* or *GetX* request respectively. Table 1 provides a brief description of each state in each one of the controllers.

Besides the state of the block, the coherence information that each of the entries in the directory should include is: the sharers of that block, the core holding the owner token (as it will be in charge of forwarding data if necessary) and a token-count field of that block. Any existing method to maintain the sharer information may be chosen [31,32]. However, a full bit vector will be assumed throughout this paper to simplify the presentation of the proposal.

A simplified version of the transition table of the sparse directory controller working with MOSAIC is shown in Table 2. When receiving a request (*GetS* or *GetX*), if the block is not present in the directory (*state I*), this controller initiates a reconstruction process like the one explained in the previous section. Note that this reconstruction

Table 1 MOSAIC protocol main states in both sparse and in-cache directories

States	Sparse-description	In-cache-description
I	Invalid. Block is not present in the sparse directory	Invalid. Block is not present in the last level cache
C_S	Constructing the block after receiving a read request (GETS)	
C_X	Constructing the block after receiving a write request (GETX)	
C_I	Does not apply	Constructing the block after receiving an instruction fetch (GetI) from a core
A	Allocated. Block is fully constructed with all the coherence information about that block	
A_S	Allocated and a read request (GETS) has been received from a core. Waiting for an unblock message	
A_X	Allocated and a write request (GETX) has been received from a core. Waiting for an unblock message	
A_I	Invalidating a block	
S	Does not apply	Shared. Block with valid data & one token
O	Does not apply	Owned. Block with valid data & at least the owner token
M	Does not apply	Modified. Block with valid data & all the tokens

process is different depending on whether the request is a *GetS* or a *GetX* and so the state the entry has to change to is different (*C_S* or *C_X* respectively). The behavior of the sparse directory when receiving an instruction cache request is the same as the behavior of a data cache read request.

During the reconstruction, when the controller receives information about some tokens' location (*event: Token Info*), it adds that sharer to the sharers' bit vector and updates the number of known located tokens. When the request triggering the reconstruction is a *GetS*, the cache with the owner token of the block will send a copy of the data with one of its tokens to the requestor. After that, it will inform the directory about how many tokens it has left. When the requestor finishes its request, it sends an unblock message (*event: Unblock*).

If the request is a *GetX*, all the caches with a copy of the requesting block will have to forward their tokens to the requestor, which will send the unblock message when it has collected all of them and so its request is finished. In this case, the directory controller will add the requestor as the exclusive sharer of the data (*state C_X, event Unblock*).

If the coherence information needed is in the directory (*state A*), all the data locations are known so the directory only has to forward the request to the appropriate sharer. If it is a read request (*GetS*), it sends it to the cache holding the owner token; if it is a write request (*GetX*), it sends it to all the sharers of the block.

The directory needs to be informed about all the replacements occurring in the private levels in order to always have updated information about the sharers. Any private cache replacing a block sends a request with the tokens (*event: PUT Tokens*), or if it has the owner token with the data (*event: PUT Data*) to the directory. The directory increases the number of tokens it owns (this is why the entry needs to have a token count field). When receiving a data replacement, if the entry is not constructed (*state I*) or there is no pending request *(state A*), data and all the tokens are written back to LLC (*action: write data in LLC*).

Replacements that occur while the line is being constructed (*C_X* or *C_S*) or when the directory is still dealing with a request (*A_S* or *A_X*) have to be handled carefully. Write requests are easier (*C_X*), because when the directory receives a replaced data block (*event: PUT Data*) or replaced tokens (*PUT Tokens*) it just forwards them to the pending requestor (*action: bounce data to requestor*). Read requests on the other hand are a little trickier, because a lot more possible situations can occur. On some occasions, the directory might be in charge of solving the pending read request with the replaced data, but it cannot be fully sure about this without more information, because it does not know whether the request has already been solved. If the reconstruction request arrived at the owner before it made its replacement, it has dealt with the pending request. If it arrived after the replacement, it could not do so because it did not have any tokens. The next section details more graphically some of these possible situations.

When the replacement message arrives with all the tokens attached the answer is clear, the request has not been solved and the directory needs to do so itself. On the contrary, if the replacement message does not include all the tokens, the directory controller is not able to know whether one of the missing tokens was sent to the requestor or not. The only way to know without sending extra control messages or

Table 2 MOSAIC sparse directory controller transition table

States \ Events	GETS	GETX	Token Info	Last Token Info	Unblock	PUT Data	PUT Tokens	Silent Replace	Replace with tokens	Ack From LLC
I	• initiate reconstruction C_S	• initiate reconstruction C_X				• write data in LLC	• write tokens in LLC			
C_S	wait	wait	• add sharer • update num tokens known	• add last sharer • wait for Unblock	• add last sharer A	• bounce data to requestor	• update tokens	wait	wait	
C_X	wait	wait			add exclusive sharer A	• bounce data to requestor	• bounce tokens to requestor	wait	wait	
A	• fward req to Owner A_S	• mcast req to all sharers A_X				• update tokens	• update tokens	• invalidate block I	• invalidate block • write Tokens in LLC A_I	
A_S	wait	wait			• add new sharer A	• update tokens	update tokens	wait	wait	
A_X	wait	wait			• remove old sharers • add exclusive sharer A	• bounce data to requestor	• bounce tokens to requestor	wait		
A_I	wait	wait				• write data in LLC	• write tokens in LLC			• remove block I

Shaded cells indicate control actions, lighter boxes indicate stalling the requests and darker ones error transition

negative acknowledgements is to finish constructing the whole entry and locate all the tokens. When the reconstruction is over, if the pending requestor did not send any token information, it means it did not receive any response and the directory needs to send one.

As the reader may appreciate, all these corner cases require the addition of more states and more events indicating these situations with their corresponding extra transitions. However, they were not included in Table 2 to avoid extra complexity for the reader and only the most common cases are illustrated. The full protocol specification for this design may be found in [33].

In a correct construction of the system, the directory and the LLC for the same address are side by side. This gives MOSAIC a great opportunity for optimization. When a request is sent to the directory, the LLC can be accessed in parallel. Although the entry is not present in the directory, if the data block is found in the LLC with all the tokens, it is possible to avoid the broadcast reconstruction request because it is known that no other copy of the block is located in any of the private caches and the directory entry reconstruction will proceed without broadcast. This speeds up the entry reconstruction and more importantly, *it filters most of the multicast messages* sent to the private caches in the CMP. As LLC capacity will be substantially higher than the number of blocks tracked by the directory, this will be the most habitual scenario for actively used private data blocks, which is the common case. Therefore, in most situations the data and all the tokens will be allocated there.

3.2.2 In-cache Directory Specification

The in-cache implementation of MOSAIC has a substantial number of similarities with the sparse version. Nevertheless, its different structure means the addition of new states and in some cases the possibility of some optimizations. A LLC controller working with a MOSAIC protocol also needs to provide information about the situation each data block is in. For this reason, it is then not sufficient to define only whether an entry is constructed or not, but it is also necessary to indicate the state that valid data are in. Therefore, in contrast to the sparse design, now there are three additional possibilities, which are that a data block can be shared (S), owned (O) or modified (M). The A state is still necessary, because block sharing information may be valid (entry constructed), while data copy is not. The C states are now a group of three different states. As well as distinguishing whether the reconstruction process is started with a read request (C_S) or a write request (C_X). Additionally, MOSAIC is optimized to react differently when there is an instruction fetch, in which case the entry is in C_I state. A brief description of the main states is given in Table 1.

Table 3 shows the main additional transitions occurring in the LLC controller compared to the sparse case (Table 2). The main difference is the existence of the C_I state whose aim is to optimize the protocol when receiving an instruction fetch. Unlike the sparse directory design, where this optimization would make the reconstruction process more complex, in the in-cache design it is possible thanks to having the sharing information next to each data block. For loads (non-instructions), MOSAIC always tries to send the data block along with *all the tokens* to the requestor in order to facilitate the following writes on that block, emulating an *exclusive (E)* state behavior. If the block has all the tokens, the controller may write in it without sending any request (i.e. upgrade miss) and the more tokens it has, the easier it will be to collect the remaining

Table 3 Additional transition states of MOSAIC in-cache LLC controller transition table

States \ *Events*	GETS	GETI (GET Instruction)	GETX	Data from Memory	Token Info	Unblock (Last Token Info from Requestor)	PUT Data	PUT Tokens	Replacement
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .
C_I	wait	wait	wait	• update Data	• add sharer • update #tokens known	• add last sharer	• write data • bounce data + 1 token to requestor	• update tokens	wait
A	• fward req to Owner → A_S	• fward req to Owner → A_S	• mcast req to sharers → A_X				• update data + tokens → M/O	• update tokens	• invalidate block → I
S	• forward request to Owner	• forward request to Owner	• send tokens • multicast request to all sharers				• update data	• update tokens	• replace Tokens
O	• send Data + all tokens	• send Data + 1token	• send Data + all tokens • mcast req to all sharers					• update tokens	• replace Data
M	• send Data + all tokens	• send Data + 1token	• send Data + all tokens						• replace Data

Shaded cells indicate control actions, lighter boxes indicate stalling the requests and darker ones error transitions

ones. Moreover, avoiding maintaining tokens in LLC favors silent entry evictions in case of replacements. Therefore, when constructing an entry, if the requested data block is present in off-chip memory, it is sent with all the tokens to the requestor. However, instructions will not be written during the execution and they may be part of shared code, so it does not make sense to initially send them with all tokens to the requestor. Instead, when off-chip memory receives a reconstruction request for an instruction, it sends a copy of the block with one token to the requestor and another copy with the rest of the tokens (including the owner) to LLC. In Table 3, when the entry is in *C_I*, it may receive a data block from memory (*event: data from Memory*) and when it receives the *Unblock* message from the requestor, it changes its state to *O* (*owner*). Thus, if those instructions are later requested by other cores, they will receive a copy with a token simply using a 2-hop process: requesting to LLC and LLC sending data to the requestor.

Another detail to take into account in the in-cache version is that replaced data has to be distinguished in order to know to which state the block needs to change to after writing it back. In Table 3, only the event *PUT Data* appears, but note that with only this event, it is not possible to know to which state the controller has to go to when the entry is in the *I* or *A* state. Once again, the complete and detailed documentation of the coherence protocol for the in-cache design can also be found in [34].

One of the main disadvantages of the in-cache structure is that, in some cases, replacements cannot be silent. When it is necessary to construct a line and there is no available space for it in the LLC, the coherence protocol needs to replace one block to construct a new one. If the data block is in the *A* state, the eviction can be made silently, but if it has some tokens, it has to replace these tokens writing them back in off-chip memory. Although LLC evictions are done the same way, in the sparse version of MOSAIC, the construction of a line does not mean an eviction from the LLC.

3.3 Running Examples

Now that the reader has a vision of the details of the coherence protocol, we can review the conceptual approach seen at the beginning of this section, but focusing on precisely describing what happens with the entry states and the rest of the copies in the system. Figures 2 and 3 shows the representation of two consecutive reads in a 4-core CMP with a MOSAIC sparse directory. We have added one additional processor (P_3) to the conceptual approach example in order to be able to see the behavior when there is a second read after the line has been reconstructed. In the same circumstances, Fig. 4 shows the representation of two possible situations that may happen when replacements occur. This helps to provide a glimpse of all the possibilities that have to be considered, taking into account all corner cases for which the system has to be prepared. The initial situation in Fig. 2 is with P1 having the data block with all the tokens except for one, which is in P_2's private cache with another copy of the data block. P_0 issues a read request *(GetS)* to the directory ❶ because it does not have the data block in its private cache (which might be composed of multiple levels). The directory does not have any entry allocated for the requested address so it broadcasts

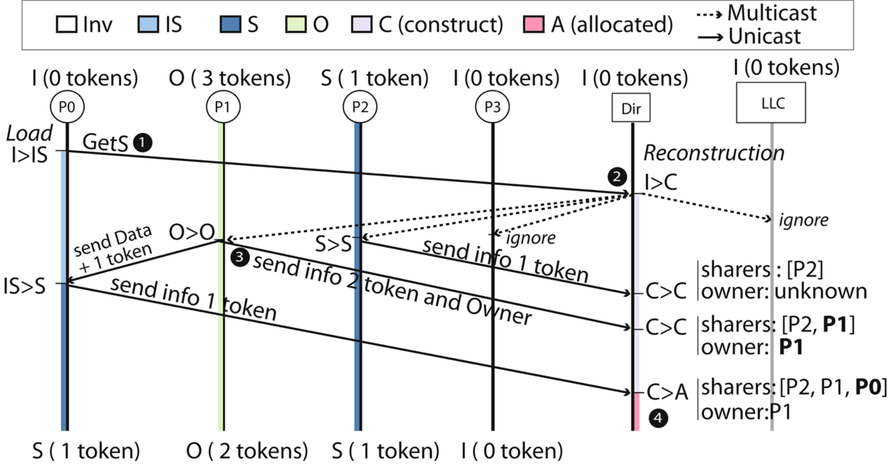

Fig. 2 Example of MOSAIC coherence protocol when a read request arrives at the directory and no entry for the requested block is allocated. P0 issues a GetS operation and the directory has to initiate the reconstruction process

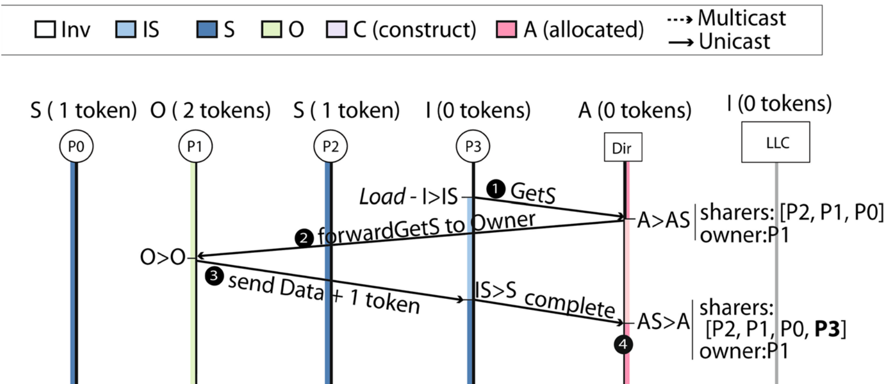

Fig. 3 Example of MOSAIC coherence protocol when a read request arrives at the directory and it finds the entry for the requested block constructed with all the coherence information

a reconstruction message ❷ asking for all the token information and indicating that P_0 needs a copy of the data block with at least one token. Processors that do not have any token ignore the request (like P_3) and processors with the data block in a shared state (such as P_2) send information about how many tokens they have. The processor holding the owner token (in this case P_1) is in charge of dealing with the initial request, so it sends a copy of the data block with one token to P_0 ❸ and sends information about all the tokens left to the directory.

While the directory is receiving messages with the token location information, it updates the sharers' vector and it increases the number of known tokens that it has received so far. It will also receive information about which processor holds the owner token. Thus, when it knows where all the tokens are and who the owner of the block is,

Fig. 4 Two possible situations when owner token is replaced during reconstruction in MOSAIC

the directory is able to ensure that the entry information is completed. In our example, this occurs when the last token information arrives from the requestor ❹, when the directory can change the state to *A* indicating that the entry is allocated with all the information updated.

After this process, any other request for that address arriving at the directory will find the entry fully reconstructed and it can be dealt with directly, like in a conventional directory protocol. This situation is shown in Fig. 3 where, using the final situation from the previous figure as the starting point, P3 issues another read request to the directory❶. This time, when the request arrives at the directory, the line is fully constructed and it includes all the necessary coherence information. Therefore, as it knows that the owner of that data block is P_1, it only has to forward ❷ the read request to P_1 and the owner will reply to P_3 with a data block copy and one token ❸. After P_3

finishes its read request, it sends a complete message to the directory, which will add it as another sharer and set the entry state back to the stable state A ❹. Similarly, in the case of a write request, if necessary, the reconstruction process is carried out in a similar way, with the main difference that each core having a copy of the block sends all their tokens to the requestor and invalidates its own copy.

Starting from the same initial situation as in Fig. 2, Fig. 4 shows two additional examples with events that are more interleaved. In this case, P1 replaces its block and sends its data with all the tokens (including the owner token) to the directory. This replacement can occur in two moments of the running example: after the reconstruction broadcast is received T or before U. In both cases, when the data replaced arrives at the directory, this is not able to know whether the sender has solved the request or not ❶. The only way to deal with this situation is to wait until the directory completely constructs the entry and knows the exact location of all the tokens. In Fig. 4 (top), P0 receives the data block from the owner ❷ so it sends a complete message like it did in the previous example. This complete message allows the directory to determine that the request was solved ❸ and that the replaced data can be forwarded to the LLC (message not shown). On the contrary, in Fig. 4 (bottom), with the 3 tokens received from the replacement and ❷ the information of the token that P2 has sent, the directory has located all the tokens for that block and checks that no complete message was received from the requestor ❸. Then the directory knows that it is responsible for sending the data block with all the tokens to the requestor. Protocol tables in [33,34] enable any event combination to be determined.

Token counting is a key component in MOSAIC, because it simplifies all the handshaking used to reconstruct directory entries and it avoids the use of negative acknowledgements as well as the necessity of timeouts or any other type of control messages. It also allows corner cases to be correctly handled as a consequence of data forwarding and ignoring requests, as seen in the previous example. For example, if the broadcast request going to P3 is delayed and this core gets the data block before receiving it, P3 will not respond properly. However, this data and token movement will only cause an efficiency issue and not a correctness one because although MOSAIC ignores requests in the absence of tokens, it does not ignore data block messages (which include tokens) when they arrive at a cache which is not expecting it. These unexpected data blocks will be bounced to their home node, which works as a serialization point and detects the anomaly in the corresponding entry. This would be silently invalidated (like in a replacement) and the single-writer and multiple-reader correctness invariant will still be accomplished. Note that this inefficiency case is highly improbable so we considered that it was better to use it than the indirection of data and tokens through the directory to keep complete ordering of the network messages.

4 Evaluation Methodology

4.1 System Configuration and Simulation Stack

To analyze MOSAIC, we will use aggressive out-of-order cores, similar to those used by commercial systems [4,7,15]. The rationale behind this decision is that instruction-

level parallelism (ILP) performance should not be underestimated [35]. Out-of-order processors will exert a high pressure on the coherence fabric. Since the number of pending instructions per core could be large, the concurrent coherence operations could be orders of magnitude bigger than those observed with a large count of simple in-order cores.

In our particular case, we will use 4-wide issue cores with 128 in-flight instructions and up to 16 pending memory operations. The numbers of cores chosen in our evaluation are 8, 16 and 32 cores per CMP. The on-chip hierarchy configuration, like in [4,7,15], is composed of three levels. The first two are private, strictly non-inclusive layers. The third level is similar to the one proposed in [15], shared following a static NUCA [23] approach. In contrast with this system, instead of an ultra-wide ring network (which in part, is imposed by the coherence protocol used) we will use a mesh network, which is characterized by a better on-chip bandwidth scalability and better performance/cost ratio [36]. In order to avoid protocol deadlock we will assume four virtual channels. By using Dimension Order Routing (DOR) we avoid packets from one source to the same destination overtaking each other. The routers in the network can handle multicast traffic natively [26]. Although for this size of system a broadcast protocol might obtain better performance [17,37], our objective is to prove that MOSAIC is capable of overcoming classic directory limitations, which will be a requirement with a much higher number of cores in the CMP.

Comparing MOSAIC with a conventional directory-based protocol, varying the directory properties (i.e. associativity and capacity) might be enough to understand the advantages of the proposal. MOSAIC can work in both sparse and in-cache directories. We begin our analysis with the former and later we will carry out a similar performance analysis with in-cache directory designs. A summary of the main parameters used in our analysis is shown in Table 4.

The main tool for our evaluation was GEMS [38]. With this tool, it is possible to perform full-system simulations. Coherence protocols have been implemented using the SLICC language (Specification Language for Implementing Cache Coherence). For the power modeling we use CACTI 6.5 [39] for modeling the cache and DSENT [40] as the network modeling tool.

The workloads considered in our study are ten multi-programmed and multi-threaded applications (scientific and server) running on top of the Solaris 10 OS. The numerical applications are three from the NAS Parallel Benchmarks suite (OpenMP implementation version 3.2 [41]). The server benchmarks correspond to the whole Wisconsin Commercial Workload suite [42]. The remaining class corresponds to multi-programmed workloads using part of the SPEC CPU2006 suite running in rate mode (where one core is reserved to run OS services).

5 Performance

When the number of cores is large, conventional directory protocols have to face limitations in two main factors, capacity and associativity. Next, we will analyze how sensitive MOSAIC is to the two parameters and compare its results with those from a conventional sparse directory implementation, as described in Sect. 2.2. This

Table 4 8-Core, 16-core and 32-core CMP system configuration

			8-Core CMP	16-Core CMP	32-Core CMP
Core Arch.	Functional units		4 × I-ALU/4 × FP-ALU/4 × D-MEM		
	ROB size/issue width		128, 4-way		
	Frequency		3 Ghz		
	Outstanding requests per core		16		
Private caches	L1	Size/associativity/block size/access time	32 KB I/D, 2-way, 64B, 1 cycles		
	L2	Size/associativity/block size/access time/type	128 KB unified, 4-way, 64B, 2 cycles, exclusive with L1		
Shared L3	Size/associativity/block size		2MB*#cores, 16-way, 64B		
	NUCA mapping		Static, interleaved by LSB		
	Slice access time		6 cycles		
Mem.	Capacity/access time		4 GB, 240 cycles		
	Memory controllers/BW		2/32 GBs	4/64 GBs	8/128 GBs
Network	Link latency/link width		1 cycle, 16B 4 virtual channels		
	Mesh topology		4 × 4	6 × 6	8 × 8
	Router latency/flow control/routing		1 cycle/Wormhole/DOR		

implementation used as a reference will be a directory with duplicate tags and it will be denoted as BASE. Under this configuration there will not be any private cache invalidations due to directory misses and therefore there should not be performance differences between MOSAIC and BASE. From that initial configuration, capacity and associativity will be modified to see how each protocol reacts.

We will start with small private caches of a 2-way 32 KB L1 I/D and unified victim cache of 4-way and 64 KB L2. Assuming in both cases a block size of 64 bytes, for these cache sizes the number of required entries in the directory to avoid capacity misses is 2048*#cores. Until Sect. 5.5, we will assume that the number of cores in the CMP is eight. Therefore, assuming 8 bytes per directory entry (enough to store tag and sharing information), the total directory size required to avoid capacity misses will be 128 KB. The storage overhead will grow with the number of cores since the aggregate private cache will increase (the number of entries needed in the directory) and the sharing vector will be larger (the size of the entries in the directory).

With the aim of minimizing the access time to data in data slices and avoiding bottlenecks in the accesses, we distribute the directory in 16 slices (as many slices as the LLC) (Fig. 1). The slice interleaving of data and directory entries over LLC uses the least significant bits of the address. For the same addresses, the directory slice and data slice are 1 cycle away. To completely avoid conflict misses in the directory, the required associativity will be 64. This large associativity is necessary because on each entry we need as many ways as the sum of both of the private levels' associativity times the number of cores [i.e. (L1I associativity + L1D associativity + L2 associativity) $*$ #cores]. Obviously, any realistic proposal should reduce this value to be scalable.

5.1 Decreasing Directory Associativity

Initially, we will determine the sensitivity of a conventional directory protocol and MOSAIC when the associativity is reduced, i.e. how the two protocols react when the number of conflict misses in the directory is increased. In order to perform this analysis, we keep the directory capacity fixed at 128 KB and modify the associativity from 64-way to 1-way per set. As associativity goes down the number of conflicts grows, because even though there is space for all potential blocks stored in private caches, some of them may conflict in the directory. Obviously, from an implementation point of view it does not make much sense to reduce associativity to 1. However, this setting allows us to analyze a key aspect of the system scalability given that when the number of cores, each with its own private cache, is increased, the number of blocks mapping to the same entry in the directory will also increase.

Figure 5 shows how the conventional base Directory protocol (BASE) and MOSAIC impact on cache level behavior when the number of directory conflicts is increased. Unsurprisingly, BASE directory has a bad reaction to that change in the associativity, forcing a large number of misses in private levels due to directory invalidations. In some applications, such as *Omnetpp* (where the cores are not sharing any data), the misses in those levels are multiplied by two. Nevertheless, and as expected, MOSAIC

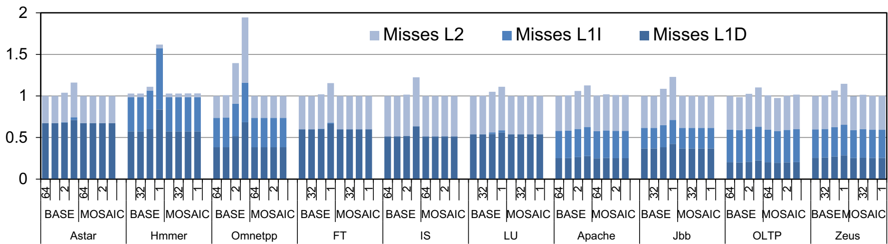

Fig. 5 Normalized number of misses at the private levels when sparse directory associativity is changed for a conventional coherence protocol (BASE) and MOSAIC

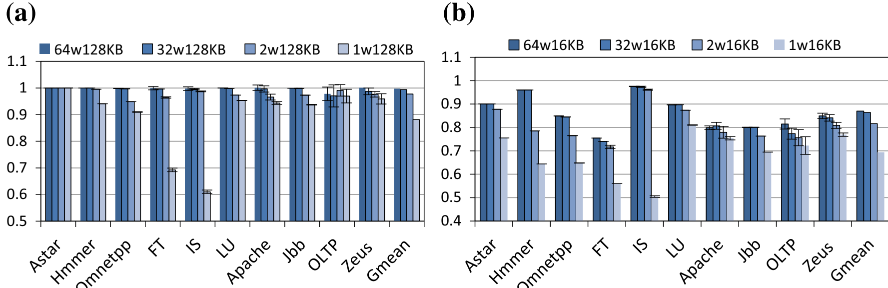

Fig. 6 **a** MOSAIC execution time normalized to BASE, while varying the associativity of a fully sized sparse directory (i.e. 16K entries). **b** MOSAIC execution time normalized to BASE, while varying the associativity for a directory with one eighth of fully sized sparse directory (i.e., 2K entries)

is completely insensitive to any associativity modifications. These results indicate that the implementation cost can be the same as the simple directly mapped configuration without any performance penalty.

The final performance differences depend on each type of application, i.e. its behavior in private caches using a duplicate tag directory. Figure 6a shows these results, indicating that the MOSAIC protocol could be up to 40% faster than the BASE protocol. For the combination of system size and applications used, the most remarkable effects are found in extreme situations when even with capacity to track all private blocks, the performance will fall, on average, 12%. Previous works, such as [31], have identified limited associativity as a major issue in directory coherence protocols. MOSAIC overcomes this problem completely since a simple directly mapped directory is capable of maintaining the performance.

5.2 Sensitivity to Capacity and Conflict Misses

The second effect that might influence performance is the capacity misses in the sparse directory. The combination of capacity misses induced by limited directory storage as well as the associativity reduction previously seen will increase total conflict misses. To compare how both effects might impact on each protocol, we reproduce the previous analysis but reducing the directory capability to track only *an eighth* of the private

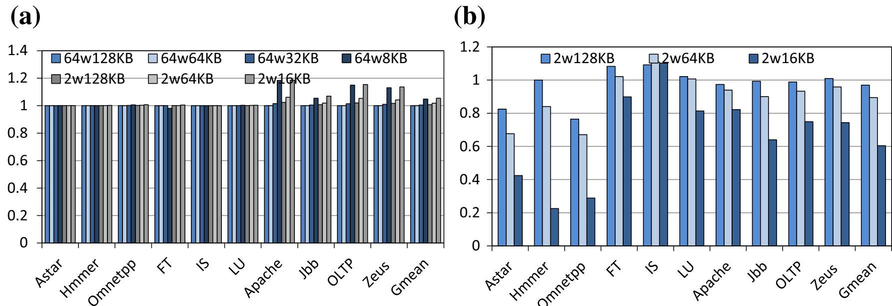

Fig. 7 **a** Average network link utilization of MOSAIC normalized to a duplicate-tag directory varying directory capacity and associativity. **b** Average network link utilization of MOSAIC normalized to BASE directory

cache capacity, i.e. up to 2K blocks. Again, such low values allow us to check the goodness of our proposal under a heavy pressure with such a tough configuration. In some way, it allows us to forecast what would happen if a large number cores map their private cache entries to a scalable directory. Figure 6b reproduces the results provided in Fig. 6a with the new directory capacity.

In this new configuration, misses in private cache for BASE, although not shown, are substantially higher. After reducing the size of the directory, even with an associativity of 64, capacity conflicts in the directory have a relevant impact on performance, degrading it by up to 20%. The capacity misses seem to be more relevant in applications with a higher sharing degree (i.e. commercial workloads [42]). Applications with a reduced working set (such as *hmmer*) are less sensitive to capacity misses in the directory. With this directory size, conflicts are more probable than in the fully sized directory and consequently associativity now has a greater influence on performance.

5.3 Bandwidth and Energy Overhead of Mosaic

Since the rationale of MOSAIC is to trade directory cost for on-chip bandwidth and additional snoops in private caches, we need to analyze the energy overheads. The first step in this analysis is to quantify how directory cost reduction influences the on-chip bandwidth consumption. If the network is using routers with support for handling multicast traffic [26], the real measure of bandwidth and energy consumption for the interconnection network is given by the average link utilization and not the end-point traffic consumption. Figure 7 shows the average link utilization for the initial configuration (i.e. exclusive 32 KB L1 and 64 KB L2) when the capacity of the directory or its associativity is reduced. The values are normalized for a duplicate tag directory, i.e. capacity for 16K entries (128 KB) and 64-way associative. The results show that on average and under the worst conditions (i.e. a 2-way associative directory, with an eighth of the capacity of the full directory) the traffic is just 5% higher than a duplicate tag directory.

Focusing our attention on each class of applications, multi-programmed workloads are completely insensitive to directory configuration. Since in these applications there is no information shared between the cores, this is the expected behavior. More note-

worthy is the behavior of scientific applications, where there is a substantial amount of shared and highly contended data. In such cases, the directory replacement algorithm prevents the eviction of actively shared data and entries of private blocks are more prone to being replaced. Consequently, traffic does not change.

Server workloads seem to be the most sensitive, since in this case the amount of shared data is large, most of them being code. Therefore these blocks will be accessed in read-only mode and the directory will be less frequently accessed. As a consequence, the chances of evicting an actively shared entry are higher than in numerical applications and so too are the chances of requiring a multicast to reconstruct these entries. Nevertheless, even in the most adverse (and unpractical) directory configurations, this increment is less than 20%, which is substantially less than in broadcast coherence protocols [17,20,37].

The key point for this behavior is that multicast is only generated when, after a miss in the sparse directory, the data and tokens available in LLC are not enough to fully reconstruct the sharing information. If the block has all the tokens, it can be ensured that there are no copies in any private caches and consequently the multicast can be avoided. Since LLC can be very large, the most usual case will be this one and, therefore, multicast will be required only if the data is really shared. In contrast, if we compare the bandwidth consumption of MOSAIC and BASE protocols when the directory is simplified, the results are very different. As Fig. 7b indicates, the BASE protocol requires more on-chip bandwidth in most cases, especially when the directory is highly limited. In the most extreme case, i.e. a 16 KB, 2-way associative directory, BASE requires up to 40% extra bandwidth consumption on average. The main reason for this is that MOSAIC has fewer misses in the private caches and directory evictions are silent. For instance, in SPEC applications all processors have mostly independent executions so the conflicts that occur in the sparse directory with a conventional directory protocol induce a large number of invalidation messages to the private levels. These invalidation requests replace the data needed by the processors, which may still be useful. Subsequent misses will require extra communication with the directory. In contrast, MOSAIC leaves these data in the private levels avoiding extra misses in the sparse directory because they is private data and so they will not be requested again. In this way it avoids requests and data travelling back and forward through the network. When the difference in the number of misses between the two protocols is small and applications have a high sharing degree, broadcast messages of the reconstruction requests are more noticeable. With highly contended shared data, such as in numerical applications, the replacement algorithm of the directory inhibits evictions of actively used data and therefore the external invalidations in caches with BASE are fewer (at least with directory configurations that are not highly constrained). Under this configuration MOSAIC memory misses might increase the traffic due to the multicast traffic required to deal with them. Although this multicast traffic might be avoided using simple solutions such as [43], it seems irrelevant in most applications. The most relevant case is *IS*, which has a large MPKI. Even in these cases, the extra traffic is less than 10%. In server applications, shared blocks rarely change their state (from *S*) and they have the same probability to be evicted as private data blocks. Consequently, the number of invalidations of useful data in private caches is larger. The result is that the extra traffic required to deal with this situation is much greater than with MOSAIC.

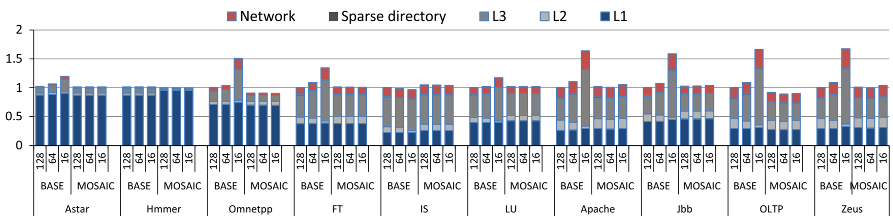

Fig. 8 Total dynamic energy used by caches and network normalized to the directory-based coherence protocol with aggregate 128 KB sparse directory. Different sizes: 128, 64 and 16 KB (8, 4 and 1 1 KB per slice)

The previous discussion partially addresses the potential added costs. To complete it, we need to look at the energy consumption, with emphasis on the cache hierarchy. Results of this analysis are shown for both protocols in Fig. 8 when using a 2-way associative sparse directory with three different sizes: 128, 64 and 16 KB. The results have been normalized to 128 KB and a 2-way directory size of BASE protocol and they are coherent with the traffic results: MOSAIC reacts in a more energy efficient way than the BASE protocol when the directory size is constrained.

5.4 Evicting Private Blocks Only

When trying to increase the effectiveness of the directory storage, some recent works [44,45] have focused their proposals on the deactivation of the coherence only for private blocks. Unlike MOSAIC, in these proposals shared blocks are still being treated as in a traditional sparse directory. Even though logic leads us to think that it is not necessary to maintain coherence in private blocks, there are other situations where shared blocks do not have to be considered for coherence either. There are many applications with read-only and shared blocks (like instructions for example), whose exclusion from the directory does not mean any loss in performance while leaving space for other highly shared data blocks. On the other hand, many blocks are initially private blocks, but they become shared at some moment of the execution, causing unnecessary traffic.

The effectiveness of both possibilities has been analyzed by comparing our proposal with the implementation of Stash [44] whose behavior is very similar to MOSAIC, but with the difference that shared blocks are not retired from the directory. As expected, results are very similar in those applications with a low-sharing degree as in the SPEC cases. Differences of between 5 and 10% occur when executing numerical applications; 20% for transactional applications which have a large working set with a high-sharing degree of code. The reasons for these results are clear. First, Stash limits the silent replacements only to the private blocks which removes the possibility of maintaining in the directory only highly-shared entries, especially when the pressure on the directory is strong. Therefore, MOSAIC's capacity to reduce the directory is obvious. Second, in those applications with a high sharing degree of read-only blocks, like the transactional ones, the (unnecessary) invalidation of shared data blocks because of lack of space in Stash is highly costly in terms of time and traffic since each of the sharers will

cause a miss in the next read. Finally, from the energy point of view, the performance improvement reduces the time dedicated to the application's execution and so the EDP as well.

Additionally, although in the results it has not been taken into account, MOSAIC reconstructions are more efficient because of the use of tokens since only the cores with some tokens reply, which reduces the reconstruction time and the generated traffic. Only if the requested data blocks are *always* shared by *all* the cores is the amount of traffic and the time needed the same in the two protocols. The effect of the additional bit of the LLC added in Stash [44] to avoid unnecessary broadcasts is the same as the effect of the tokens in the LLC in MOSAIC. Finally, it is convenient to recall the importance of how the applications with large footprints obtain most benefit from our proposal, since these characteristics are the same as the increasing needs of the Cloud and Big Data applications.

5.5 Scalability

One important aspect of our proposal is the scalability. Although the full-system simulation of a realistic architecture, including the OS, like the one used in this work limits the size of the system, we have doubled the whole system size twice to check the trend of each of the results shown until now. Obviously, when increasing the number of cores, the number of LLC banks also has to be scaled and consequently the size of the interconnection network has to be scaled too. We used a 6 × 6 mesh (16 cores) and 8 × 8 mesh (32 cores) which means an increase of 50 and 100% respectively in the bisection bandwidth. The rest of the parameters have not been modified.

The main obstacle that will limit the scalability of our proposal is undoubtedly the interconnection network, which will have to manage the increased traffic of the new system. Figures 9 and 10 show the link utilization of MOSAIC normalized to a duplicate tag directory when varying the directory capacity and associativity. As can be seen, even having increased the bisection from 4 to 6 links (50%) or 8 links (100%), the increment in traffic is far from the bisection increment, even with the most unfavorable directory configurations.

If these results are compared to the corresponding results with 8 cores (Fig. 7) the difference compared with the duplicate tag directory is even greater. Even in extreme situations, such as the one corresponding to a 2-way set associative directory

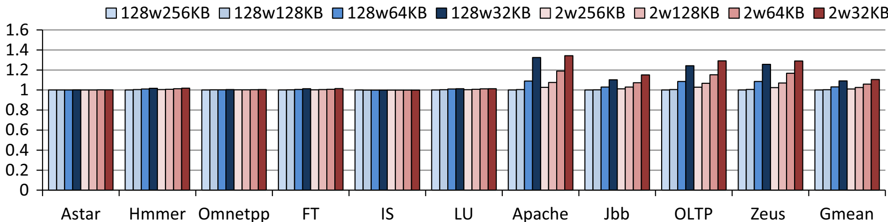

Fig. 9 Link utilization of MOSAIC normalized to a duplicate tag directory (128-way associative, 256 KB) varying directory capacity and associativity in a 16-core CMP

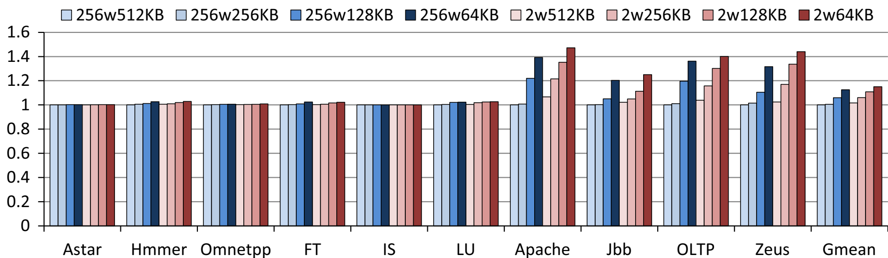

Fig. 10 Link utilization of MOSAIC normalized to a duplicate tag directory (256-way associative, 512 KB) varying directory capacity and associativity in a 32-core CMP

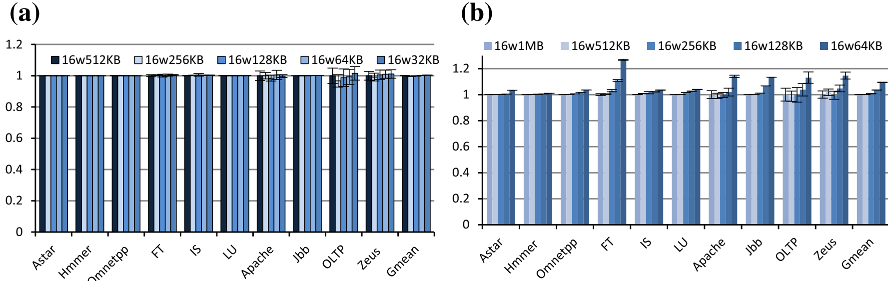

Fig. 11 **a** MOSAIC execution time normalized to duplicate tag directory, for a Nehalem-like private caches configuration varying directory capacity. **b** The same experiment for the BASE protocol

with capacity to track an eighth of the private caches, only 15% more on average is consumed than with a Duplicate Tag Directory. The rationale for this is that misses (due to directory invalidations) in private caches take longer to be resolved in LLC due to the larger size of the system.

Therefore, it seems reasonable to say that MOSAIC will scale up to systems with tens of out-of-order cores, such as those considered in this work. Note that this represents approximately an order of magnitude when considering in-order cores.

5.6 Realistic Private Cache Configuration

Up to now, we have been using limited private cache capacity and associativity. If we consider the configuration of commercial systems [4,7,15], L2 caches have between 1/8 and 1/4 of L3 capacity and both L1 and L2 have a larger associativity. Therefore, we will next carry out a sensitivity analysis for the size of the directory with a realistic configuration for private caches.

In this particular case, we try to mimic the L2 cache configuration in Intel's Nehalem (4-way 32 KB of L1s and 8-way 256 KB of L2). We will keep the associativity fixed at 16 ways (like in the data banks) and vary the capacity of the directory, from double [13] the full directory (i.e. 640 KB) to *a tenth of full directory* (i.e. 32 KB). Figure 11 presents the average execution time for each application normalized to the double-sized directory where, even with the smallest capacity, there is no performance impact. When

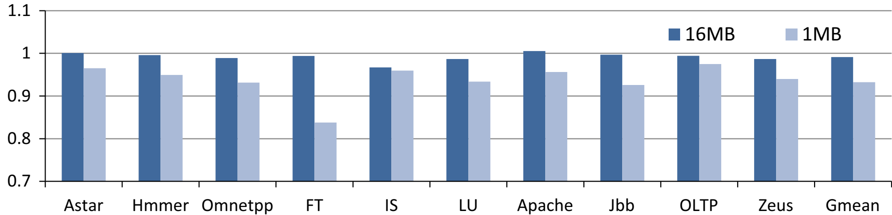

Fig. 12 Execution time of MOSAIC normalized to BASE directory when using in-cache MOSAIC in an 8-core CMP and varying the LLC size

reproducing the same experiment for the BASE protocol, the performance impact is greater than 20% in some cases.

6 In-cache Analysis

Although the previous results have been focused on a sparse directory configuration, MOSAIC has also been implemented on *in-cache* configuration architecture. It is well-known that introducing coherence information into the LLC under specific requirements has some advantages. The most important one is to avoid having to duplicate the cache block tags and the associated logic in an extra directory. It is only necessary to add the sharing information of each block to each entry in the LLC. However, this space saving decreases as the relation between the aggregate private cache capacity and the total LLC size increases. For a Nehalem-like structure where the LLC is 4 times larger than the aggregate private cache capacity, when needing more than 32 bits per block, it is more area-efficient to use a sparse directory structure separated from the LLC.

In any case, in-cache MOSAIC has the same advantages over BASE as the ones seen in the sparse design analysis supporting the same arguments. When the relation between the total amount of private cache capacity and the LLC size is closer to 1, i.e. the same size in both of them, MOSAIC'S advantages are greater. This happens because, as almost all the entries in the LLC are used to track blocks in the private caches and so there are not too many victim entries, whenever there is a replacement in the LLC, MOSAIC will not invalidate any data block allocated in private cache while the BASE directory will.

As the relation between private and shared capacities gets further from 1 and so the number of invalidations of blocks in the private caches decreases, so does the performance difference between the two protocols. Figure 12 shows the execution time of BASE and MOSAIC normalized to the BASE directory with two different size relations: 1/16 and 1.

In spite of the better performance of in-cache and the inconvenience of the necessary logic for maintaining separate structures, the sparse directory allows the size of the directory itself and the size of the LLC to be decoupled. This gives an important advantage to this architecture design from a scalability point of view, because as the number of cores increases, the total capacity of the private caches also increases and so

it will be necessary to increase the LLC size to maintain the off-chip traffic bounded. Even though neither in-cache or sparse MOSAIC requires a quadratic increase in the storage space needed for keeping the coherence information, the sparse solution does not limit the size of the LLC as the in-cache does. Anyway, if the number of cores is not very large and the private/shared size ratio is low, the in-cache MOSAIC is as efficient as the sparse solution and with a lower complexity.

7 Conclusions

A new coherence protocol that addresses the challenges of complex multilevel cache hierarchies in future many-core systems has been implemented. In order to limit coherence protocol complexity, inclusiveness is required to track coherence information throughout levels in this type of systems, but this might introduce unsustainable costs for directory structures. Cost reduction decisions taken to reduce this complexity may introduce artificial inefficiencies in the on-chip cache hierarchy, especially when the number of cores and private cache size is large. The coherence protocol presented in this paper, denoted MOSAIC, introduces a new approach to tackle this problem. In energy terms, the protocol scales like a conventional directory coherence protocol, but relaxes the shared information inclusiveness. This allows the performance implications of directory size and associativity reduction to be overcome. MOSAIC demonstrates that inclusiveness is avoidable and can be removed from a directory coherence protocol, while maintaining the complexity constrained. In fact, MOSAIC is even simpler than a conventional directory. The results of our evaluation show that the approach is quite insensitive, in terms of performance and energy expenditure, to the size and associativity of the directory.

Acknowledgements Funding was provided by Spanish Government (Grant Nos. TIN2015-66979-R and TIN2016-80512-R).

References

1. Rogers, B.M., Krishna, A., Bell, G.B., Vu, K., Jiang, X., Solihin, Y.: Scaling the bandwidth wall: challenges in and avenues for CMP scaling. Int. Symp. Comput. Archit (ISCA) **37**(3), 371 (2009)
2. ITRS.: Roadmap 2012. http://www.itrs.net/links/2012itrs/home2012.htm (2012)
3. Prieto, P., Puente, V., Gregorio, J.A.: Multilevel cache modeling for chip-multiprocessor systems. IEEE Comput. Archit. Lett. **10**(2), 49–52 (2011)
4. Butler, M.: "AMD 'Bulldozer' Core—a new approach to multithreaded compute performance for maximum efficiency and throughput," In: IEEE HotChips Symposium on High-Performance Chips (HotChips 2010) (2010)
5. Hammarlund, P., Martinez, A.J., Bajwa, A.A., Hill, D.L., Hallnor, E., Jiang, H., Dixon, M., Derr, M., Hunsaker, M., Kumar, R., Osborne, R.B., Rajwar, R., Singhal, R., D'Sa, R., Chappell, R., Kaushik, S., Chennupaty, S., Jourdan, S., Gunther, S., Piazza, T., Burton, T.: Haswell: the fourth-generation intel core processor. IEEE Micro **34**(2), 6–20 (2014)
6. Feehrer, J., Jairath, S., Loewenstein, P., Sivaramakrishnan, R., Smentek, D., Turullols, S., Vahidsafa, A.: The oracle sparc T5 16-core processor scales to eight sockets. IEEE Comput. Soc. **33**(2), 48–57 (2013)
7. Kalla, R., Sinharoy, B., Starke, W.J., Floyd, M.: Power7: IBM's next-generation server processor. IEEE Micro **30**(2), 7–15 (2010)

8. Molka, D., Hackenberg, D., Schone, R., Muller, M.S.: Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In: 2009 18th International Conference on Parallel Architectures and Compilation Techniques, pp. 261–270 (2009)
9. Busaba, F., Blake, M.A., Curran, B., Fee, M., Jacobi, C., Mak, P.-K., Prasky, B.R., Walters, C.R.: IBM zEnterprise 196 microprocessor and cache subsystem. IBM J. Res. Dev. **56**(1), 1:1–1:12 (2012)
10. Starke, W.J., Stuecheli, J., Daly, D.M., Dodson, J.S., Auernhammer, F., Sagmeister, P.M., Guthrie, G.L., Marino, C.F., Siegel, M., Blaner, B.: The cache and memory subsystems of the IBM POWER8 processor. IBM J. Res. Dev. **59**(1), 3:1–3:13 (2015)
11. Topol, A.W., La Tulipe, D.C., Shi, L., Frank, D.J., Bernstein, K., Steen, S.E., Kumar, A., Singco, G.U., Young, a M., Guarini, K.W., Ieong, M.: Three-dimensional integrated circuits. IBM J. Res. Dev. **50**(4), 491–506 (2006)
12. Choi, B., Komuravelli, R., Sung, H., Smolinski, R., Honarmand, N., Adve, S.V., Carter, N.P., Chou, C.-T.: DeNovo: rethinking the memory hierarchy for disciplined parallelism. In: 20th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 155–166 (2011)
13. Howard, J., Dighe, S., Vangal, S.R., Ruhl, G., Borkar, N., Jain, S., Erraguntla, V., Konow, M., Riepen, M., Gries, M., Droege, G., Lund-Larsen, T., Steibl, S., Borkar, S., De, V.K., Van Der Wijngaart, R.: A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling. IEEE J. Solid-State Circuits **46**(1), 173–183 (2011)
14. Martin, M.M.K., Hill, M.D., Sorin, D.J.: Why on-chip cache coherence is here to stay. Commun. ACM **55**(7), 78 (2012)
15. Kurd, N., Douglas, J., Mosalikanti, P., Kumar, R.: Next generation Intel®micro-architecture (Nehalem) clocking architecture. In: IEEE Symp. VLSI Circ., pp. 62–63 (2008)
16. Conway, P., Kalyanasundharam, N., Donley, G., Lepak, K., Hughes, B.: Cache hierarchy and memory subsystem of the AMD opteron processor. IEEE Micro **30**(2), 16–29 (2010)
17. Raghavan, A., Blundell, C., Martin, M.M.K.: Token tenure: PATCHing token counting using directory-based cache coherence. In: *41st IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 47–58 (2008)
18. Menezo, L.G., Puente, V., Gregorio, J.A.: The case for a scalable coherence protocol for complex on-chip cache hierarchies in many-core systems. In: 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 279–288 (2013)
19. Gupta, A., Weber, W., Mowry, T.: Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In: International Conference on Parallel Processing, pp. 167–192 (1990)
20. Martin, M.M.K., Hill, M.D.D., Wood, D.A.: Token coherence: decoupling performance and correctness. In: 30th International Symposium on Computer Architecture (ISCA), pp. 182–193 (2003)
21. Baer, J.-L., Wang, W.-H.: On the inclusion properties for multi-level cache hierarchies. ACM SIGARCH Comput. Archit. News **16**(2), 73–80 (1988)
22. Jaleel, A., Borch, E., Bhandaru, M., Steely Jr., S.C., Emer, J.: Achieving non-inclusive cache performance with inclusive caches: temporal locality aware (TLA) cache management policies. In: 43rd IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 151–162 (2010)
23. Huh, J., Kim, C., Shafi, H., Zhang, L., Burger, D., Keckler, S.W.: A NUCA substrate for flexible CMP cache sharing. IEEE Trans. Parallel Distrib. Syst. **18**(8), 1028–1040 (2007)
24. Lee, K., Lee, S.J., Yoo, H.J.: Low-power network-on-chip for high-performance SoC design. IEEE Trans. Very Large Scale Integr. Syst. **14**(2), 148–160 (2006)
25. Agarwal, N., Peh, L., Jha, N.K.: In-network snoop ordering (INSO): snoopy coherence on unordered interconnects. In: 15th International Symposium on High Performance Computer Architecture (HPCA), pp. 67–78 (2009)
26. Jerger, N.E., Peh, L.S., Lipasti, M.: Virtual circuit tree multicasting: a case for on-chip hardware multicast support. In: International Symposium on Computer Architecture (ISCA), pp. 229–240 (2008)
27. Abad, P., Puente, V., Menezo, L.G., Gregorio, J.A.: Adaptive-Tree Multicast: Efficient Multidestination Support for CMP Communication Substrate. IEEE Trans. Parallel Distrib. Syst. **23**(11), 2010–2023 (2012)
28. Zebchuk, J., Srinivasan, V., Qureshi, M.K.M.K., Moshovos, A.: A tagless coherence directory. In: International Symposium on Microarchitecture (MICRO), pp. 423–434 (2009)
29. OpenSPARC T2 Core Microarchitecture Specification. Santa Clara, CA (2007)

30. Sorin, D.J., Plakal, M., Condon, A.E., Hill, M.D., Martin, M.M.K., Wood, D.A.: Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. IEEE Trans. Parallel Distrib. Syst. **13**(6), 556–578 (2002)
31. Sanchez, D., Kozyrakis, C.: SCD: a scalable coherence directory with flexible sharer set encoding. In: 18th IEEE International Symposium on High Performance Computer Architecture, pp. 1–12 (2012)
32. Sanchez, D., Kozyrakis, C.: The ZCache: decoupling ways and associativity. In: International Symposium on Microarchitecture (MICRO), pp. 187–198 (2010)
33. Menezo, L.G.: Mosaic coherence protocol specification. State Table (sparse design) (2016). https://www.ce.unican.es/docs/coherence_protocols/mosaic_sparse/index.html
34. Menezo, L.G.: Mosaic coherence protocol specification. State Table (in-cache design) (2016). https://www.ce.unican.es/docs/coherence_protocols/mosaic_incache/index.html
35. Shin, J.L., Park, H., Li, H., Smith, A., Choi, Y., Sathianathan, H., Dash, S., Turullols, S., Kim, S., Masleid, R., Konstadinidis, G., Golla, R., Doherty, M.J., Grohoski, G., McAllister, C.: The next-generation 64b SPARC core in a T4 SoC processor. IEEE J. Solid-State Circuits **48**(1), 82–90 (2013)
36. Pinkston, T.M., Duato, J.: Appendix F: interconnection networks. In: Computer Architecture: A Quantitative Approach, 5th ed. Morgan Kaufmann, Burlington (2012)
37. Menezo, L.G., Puente, V., Abad, P., Gregorio, J.A.: Improving coherence protocol reactiveness by trading bandwidth for latency. In: ACM International Conference on Computing Frontiers (CF'12), pp. 143–152 (2012)
38. Martin, M.M.K., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. ACM SIGARCH Comput. Archit. News **33**(4), 92–99 (2005)
39. Muralimanohar, N., Balasubramonian, R., Jouppi, N.: Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In: 40th IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 3–14 (2007)
40. Sun, C., Chen, C.-H.O., Kurian, G., Wei, L., Miller, J., Agarwal, A., Peh, L.-S., Stojanovic, V.: DSENT—a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In: International Symposium on Networks-on-Chip (NOCS), pp. 201–210 (2012)
41. Jin, H., Frumkin, M., Yan, J.: The OpenMP implementation of NAS parallel benchmarks and its performance. Natl. Aeronaut. Sp. Adm. (NASA), Tech. Rep. NAS-99-011, Moffett Field, USA, no. October (1999)
42. Alameldeen, A.R., Martin, M.M.K., Mauer, C.J., Moore, K.E., Hill, M.D., Wood, D.A., Sorin, D.J.: Simulating a $2M commercial server on a $2K PC. Computer (Long. Beach. Calif) **36**(2), 50–57 (2003)
43. Loh, G.H., Hill, M.D.: Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In: International Symposium on Microarchitecture (MICRO), p. 454 (2011)
44. Demetriades, S., Cho, S.: Stash directory: a scalable directory for many-core coherence. In: International Symposium for High-Performance, Computer Architecture (HPCA) (2014)
45. Cuesta, B.A., Ros, A., Gómez, M.E., Robles, A., Duato, J.F.: Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In International Symposium on Computer Architecture (ISCA), p. 93 (2011)