ICCI: In-Cache Coherence Information

Antonio García-Guirado, Ricardo Fernández-Pascual, and José M. García

Abstract—In this paper we introduce ICCI, a new cache organization that leverages shared cache resources and flat coherence protocols to provide inexpensive hardware cache coherence for large core counts (e.g., 512), achieving execution times close to a non-scalable sparse directory while noticeably reducing the energy consumption of the memory system. Very simple changes in the system with respect to traditional bit-vector directories are enough to implement ICCI. Moreover, ICCI does not introduce any storage overhead with respect to a broadcast-based protocol, yet it provides large storage space for coherence information. ICCI makes smarter use of cache resources by dynamically allowing last-level cache entries to store blocks or sharing codes. This way, just the minimum number of directory entries required at runtime are allocated. Besides, ICCI suffers a negligible amount of directory-induced invalidations. Results for a 512-core CMP show that ICCI reduces the energy consumption of the memory system by up to 48 percent compared to a tag-embedded directory, up to 15 percent compared to a sparse directory, and up to 8 percent compared to the state-of-the-art Scalable Coherence Directory which ICCI also outperforms in execution time. In addition, ICCI can be used in combination with elaborated sharing codes to apply it to extremely large core counts. We also show analytically that ICCI's dynamic allocation of entries makes it a suitable candidate to store coherence information efficiently for very large core counts (e.g., over 200K cores), based on the observation that data sharing makes fewer directory entries necessary per core as core count increases.

Q1 **Index Terms—**Cache coherence, cache organization, scalability, multi-core, energy-efficiency

1 Introduction

Cache coherence enables simple shared-memory programming models that facilitate writing efficient parallel applications. During the next years, hardware coherence will remain desirable for developing new non-structured parallel programs as well as for running legacy applications [1]. Chips designed for market segments ranging from high-performance computing to cloud computing will benefit from a coherent shared-memory model. To allow this, it is needed to integrate a scalable hardware cache coherence mechanism onto these future chips. A recent commercial example is the new SGI UV2 [2] machine developed by Silicon Graphics.

Scaling coherent cache hierarchies for the envisioned chip multiprocessors (CMPs) that integrate large numbers of cores (e.g., hundreds or thousands) is problematic. The lack of scalability of existing coherence mechanisms in terms of area, traffic and energy-efficiency, as well as the introduction of spurious coherence invalidations of active cache lines [3] (L1 cache line invalidations performed on directory evictions to maintain coherence), may limit the applicability of cache coherence to future CMPs.

Large systems typically use directory-based cache coherence protocols [4]. The directory keeps information about which cores are using each cache line, enabling efficient point-to-point coherency traffic that uses scalable networks, such as meshes or tori, at the expense of using extra memory to store the sharing information.

Unfortunately, designing a scalable directory for large core counts is not easy, especially if the directory stores exact sharing information, which is necessary for minimizing network traffic. The directory organization has to deal with two extreme cases that may arise during the execution of programs. These are the following:

- All blocks in the private caches are private to the cores (no additional sharers exist for any of the blocks). This determines the maximum number of entries that will ever be used in the directory at once, one per private-cache entry. We define the *coverage* of a directory as the percentage ratio of the number of directory entries to the number of private-cache entries. The worst case that the directory must cover to offer good performance is equivalent to a 100 percent coverage (one directory entry per private-cache entry).
- A block is shared by all cores. This determines the size of each entry, which must have enough room to store all the sharers of the block (i.e., all cores). Usually, a full-map bit-vector is used, containing 1 bit to indicate the presence (or absence) of a copy of the block in each of the private caches [3], [5], [6], [7].

Handling both extreme cases makes directories become eventually unaffordable as the number of nodes increases, with both a high number of entries and a large size for each entry (1 bit per core), resulting in a per-tile area overhead proportional to the number of cores.

- *A. García-Guirado is with the Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Murcia, Spain, and Intel Barcelona Research Center, Intel Labs, Universitat Politècnica de Catalunya, Barcelona, Spain. E-mail: toni@ditec.um.es, antoniox.garcia.guirado@intel.com.*
- *R. Fernández-Pascual and J.M. García are with the Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia,* Q2 *Murcia, Spain. E-mail: {rfernandez, jmgarcia}@ditec.um.es.*

Manuscript received 20 Feb. 2013; revised 21 Dec. 2013; accepted 30 Jan. 2014; date of publication xx xxx; date of current version xx xx xxxx.
Recommended for acceptance by R. Gupta.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2014.2308185

To make things worse, the set-associative arrays used to store the directory suffer from address conflicts that cause directory-induced invalidations in the private caches [3]. To reduce the amount of conflicts and invalidations, the directory must be *overprovisioned* (with a coverage over 100 percent), containing many more entries than those necessary for the worst case scenario [8].

A number of proposals try to reduce the size of the directory's exact sharing code and the frequency of directory-induced invalidations. These usually imply an increase in the complexity, latency and energy consumption of the directory circuitry and the cache coherence protocol. For instance, hierarchical directories [1] have been proposed as a scalable alternative to flat directories. By distributing the sharing information in several levels, their per-core overhead is proportional to the k-th root of the number of cores (where k is the number of levels of the hierarchy). However, the complex management of the distributed sharing information makes these protocols difficult and costly to validate and implement in real systems.

Inexact sharing codes are another way of reducing the directory overhead [7], [9], [10], [11]. However, code inexactitude causes superfluous coherence actions, increasing network traffic and execution time and compromising scalability.

In this paper we tackle the area scalability problems of directories from a novel perspective. We address an inefficiency that consists of the use of a fixed-size dedicated structure for the directory, typically made up of set-associative arrays which need to be oversized to cover even worst-case scenarios. We show that such structure is unnecessary when scaling up the number of cores, and it is actually harmful in a number of ways, as it introduces unaffordable area overhead and causes a high number of directory-induced invalidations. We discuss a novel way to store directory entries that we call in-cache coherence information (ICCI). ICCI uses storage structures already present in the chip (the last level cache (LLC)) to dynamically store directory entries with fine granularity, allocating just the strictly necessary number of these. ICCI is based on the following observation: as the number of cores rises, the number of directory entries required per-core does not increase, because the number of private-cache entries to track per core remains constant and, at the same time, data sharing implies that a single directory entry can track an increased number of private-cache entries (one per sharer, with potentially as many sharers as cores). This means that fewer and fewer directory entries per-core will be in effective use. Hence, dynamically allocating only the necessary entries, like ICCI does, results in scalable directory storage overhead.

The overhead of ICCI remains within acceptable limits for proper multi-core scalability regarding area, latency and energy. ICCI is orthogonal to the sharing code used (it is not our purpose to develop any new sharing codes). In our analysis, we will use full-map sharing codes for core counts up to 512, and a hierarchical code for larger core counts up to 256 K. The nature of coherence information makes ICCI very suitable to easily store it, rivaling in area with other scalable schemes (e.g., SCI [12], hierarchical protocols [1] or duplicate-tag directories [13]) without incurring their associated problems (e.g., slow invalidations, complexity, power-consuming lookups) and providing some particular advantages, such as suffering a negligible amount of directory-induced invalidations.

ICCI provides *dynamic directory coverage*, which never rises over 100 percent and can be as low as 0 percent depending on runtime workload characteristics. We have studied the range of coverages resulting from different application characteristics and it can be concluded that, under reasonable circumstances, the total amount of storage used for storing coherence dynamically is within scalable limits.

The rest of this paper is organized as follows: Section 2 gives the necessary background for ICCI. Section 3 describes ICCI. Section 4 compares a possible implementation of ICCI against both traditional and novel directory schemes from performance and energy-consumption points of view. Section 5 provides an analytical study of the storage overhead introduced by ICCI and its scalability in comparison to other coherence schemes. A survey of other alternative directory schemes can be found in Section 6. Finally, our conclusions are presented in Section 7.

2 Background

Chip multiprocessors containing several processing cores are the reigning architecture nowadays [8], [14], [15], [16], [17]. They are expected to remain so for the foreseeable future, integrating more cores to make cost-effective use of the extra transistors provided by shrinking feature sizes. CMPs usually employ a tiled design with NUCA caches [18], distributing the storage capacity across the chip, each tile containing a core and its associated bank of the shared NUCA last level cache. Through this paper we assume a cache hierarchy composed by a shared LLC and private L1 caches, without loss of generality. The cache coherence protocols discussed in this paper maintain coherence among the private L1 caches, and use exact sharing information stored on-chip unless otherwise noted. A scalable network (typically a mesh) connects all the cores. Next, we describe some cache coherence schemes necessary for understanding ICCI.

2.1 Inclusive Cache with Tag-Embedded Sharing Information

Inclusive cache hierarchies, in which each level of the hierarchy contains all the blocks stored in the levels closer to the cores, provide a natural directory when sharing codes are co-located with the LLC tags [5], [6]. However, this option gets less interesting as the number of cores grows. Embedding the sharing code in the tags makes the entry size increase linearly with the number of cores. With 512 cores and a full-map bit-vector, the size of the sharing bit-vector would match the size of its associated 64-byte cache line, and the total LLC area would almost double. This is an unaffordable overhead, and for this reason tag-embedded sharing information remains used only for low core counts. Some recent Intel microarchitectures use this scheme to keep coherence among four private L2 caches, being the L3 cache shared and

inclusive, with its tags containing core valid bits (CVB) to indicate which core private caches contain copies of the block [19].

In general, if the total number of entries in the LLC is t times larger than in all the private caches combined (e.g., t is equal to 8 in Intel's Ivy Bridge [19]), a tag-embedded directory can potentially track $t \times n$ times as many sharers as there are single entries in the private caches, where n is the number of cores. This is an exorbitant capacity compared to the real usage of these resources.

2.2 Sparse Directory

Sparse directories [7] are caches that store sharing codes (instead of memory blocks) to track all the contents of the private caches. Since the L1 entries to track are much fewer than the LLC entries, the utilization of tag-embedded directories is low (i.e., most directory entries track no sharers), and the sparse directory takes advantage of this fact to reduce the coherence information overhead by storing a smaller number of sharing codes separately from the LLC. The sparse directory is typically banked and distributed following the same pattern as the NUCA LLC.

Sparse directories also enable non-inclusive and exclusive caches naturally. Blocks stored in the L1 caches are tracked by the sparse directory, and they do not need to be stored in the LLC, leaving room for extra blocks compared to inclusive caches. If a block can be stored in the LLC while copies in the private caches exist (tracked by the sparse directory), the cache hierarchy is said to be non-inclusive. If the block can be on either the LLC or the sparse directory, but not both, the cache hierarchy is exclusive. Exclusive and non-inclusive caches provide a more efficient use of LLC resources, at the cost of introducing extra three-hop L1 misses (those in which, in addition to the request and response messages, a third message is needed to reach the L1 cache supplying the block), and performing data transmissions on clean L1 writebacks (when no copy of the block exists in the LLC, which is always the case in purely exclusive caches and can be the case for blocks in exclusive state in non-inclusive caches). The number of three-hop misses and data transmission on L1 writebacks in non-inclusive caches depends on the particular allocation (and bypass) policies used by the LLC to optimize the LLC-capacity/latency/bandwidth features of the design in question (examples are allocation upon an LLC miss, upon an L1 eviction or upon L1 sharing).

Directory-induced invalidations are another serious problem in sparse directories. Because directory-induced invalidations affect blocks actively used by the cores, they generate extra L1 cache misses with negative effects on performance. To reduce the number of such invalidations, overprovisioned sparse directories are used, and 200 percent coverages are not uncommon [8]—meaning that the sparse directory has twice as many entries as all the L1 caches that it tracks combined.

Even though smaller than a tag-embedded directory, a sparse directory has room for tracking $\frac{c}{100} \times n$ times the total number of entries in the private caches, where c is the coverage (e.g., 200 percent) and n is the number of cores. Most of this space, whose per-code size grows proportional to the core count, will be always unused.

2.3 Scalable Coherence Directory (SCD)

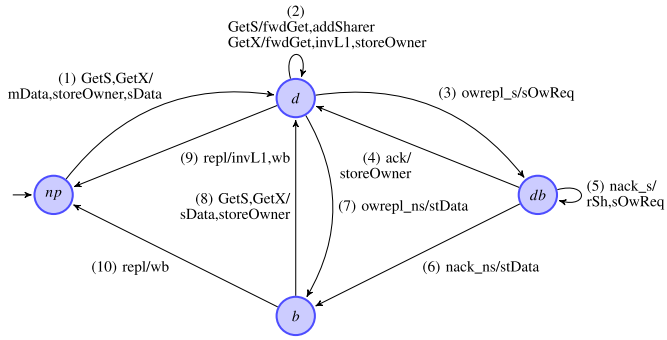
To date, the Scalable Coherence Directory [20] is arguably one of the most promising directory schemes for supporting coherence for high core counts (see Section 6 for other recent proposals). SCD is capable of storing exact sharing information in hierarchical entries within a single cache. This way, SCD has the same scalability properties as hierarchical protocols. SCD's per-core storage overhead is proportional to the k-th root of the number of cores, with k being the number of levels of the hierarchy, fixed at design time. The strong point of SCD is that a flat directory can be used instead of a hierarchical one, because all sharing information of a memory block can be found in the same directory cache. A flat directory is much simpler, easier to implement, and even to formally prove correct, than a hierarchical directory.

The smart sharing information encoding of SCD enables a memory block shared by few cores to use just one directory entry. Additional entries are allocated as the number of sharers increases, creating a tree structure, with entries pointing to child-entries containing further sharers.

The storage of sharing codes in multiple entries makes SCD's lookup mechanism more complex. Multiple lookups are needed to retrieve all the SCD cache entries containing the sharing information of highly shared blocks. On such a common event as a block invalidation, this multiple-lookup process is carried out to retrieve the sharers to be invalidated. In general, up to $\sum_{a=1}^{k} \frac{n}{(\sqrt[k]{n})^a}$ lookups are necessary to read all the entries of a k-level hierarchy, assuming n cores. For instance, a 512-core two-level hierarchy requires 23 sequential lookups in the same SCD cache, in the critical path of the invalidation process. Additional hierarchy levels reduce SCD's entry size but increase the number of lookups. SCD's entries for a nine-level 512-core hierarchy require just a 2-bit vector, but 511 of these entries make up the hierarchy. Reading so many entries sequentially for an invalidation is not affordable.

As long as the lookup latency overlaps with others, it should not affect performance. Prior work tested SCD with an unicast network, and the results showed that the look-up latency for a 1,024-core two-level hierarchy was overshadowed by unicast message sending. However, unicast is very inefficient when sending messages to large core counts. Mechanisms such as multicast or cruise-missile invalidates [13] are interesting then, as suggested by SCD's authors, for both the sending of requests and the recollection of responses, to improve the performance and energy efficiency of the system. When using these mechanisms, the effect of the look-up latency on the execution time would become more important.

To reduce the number directory-induced invalidations, SCD relies on the high-associative properties of ZCaches [21]. ZCaches provide high associativity by considering



states	events	actions
np: block not present	**GetS**: a core's load request	**mData**: get data from main memory
d: directory info present, block in L1 cache(s)	**GetX**: a core's write request	**storeOwner**: store owner in the directory information
b: block in LLC only	**owrepl_ns**: replacement from L1 (no additional sharers exist)	**sData**: send data to requesting core
db: seeking new owner	**owrepl_s**: replacement from L1 (additional sharers exist)	**fwdGet**: forward request to owner
	repl: LLC block replacement	**addSharer**: store new sharer in the directory information
	nack_ns: L1 cannot accept ownership (no more sharers)	**invL1**: invalidate L1 copies
	nack_s: L1 cannot accept ownership (sharers exist)	**sOwReq**: send request for a sharer to accept ownership
	ack: L1 accepted ownership	**rSh**: remove stale sharer
		stData: store data in the LLC
		wb: send data to main memory

Fig. 1. ICCI's state diagram for the LLC.

many replacement candidates on evictions. This allows SCD to outperform traditional hierarchical protocols in which directory-induced invalidations are more frequent. Further details on SCD can be found in [20].

3 ICCI: In-Cache Coherence Information

ICCI is a new cache organization that provides natural support for storing cache coherence information. This support is derived from a novel usage of LLC entries to store either a cache line or sharing information about the copies of a memory block stored in the L1 caches.

The LLC is dynamically filled with cache lines and sharing codes, taking up just the strictly necessary number of entries for storing cache coherence information. A flat cache coherence protocol can be used to maintain coherence. No specific sharing code or inclusion properties of the cache hierarchy are enforced by ICCI. We use a full-map bit vector along most of this paper to make use of the large storage capabilities of ICCI, but other sharing codes can be used. More efficient compressed codes for larger core counts can be enabled by the large size of cache lines (or several LLC entries may be combined to create even larger *composable* codes, similarly to SCD).

To understand ICCI, it is illustrative to compare it with tag-embedded directories. We can think of it this way: ICCI moves the tag-embedded sharing code from the tag into the data field of the cache entry. The tag no longer needs to grow in size, while the data field has plenty of space to store large sharing codes (e.g., 512 bits with 64-byte cache lines). Because a cache entry cannot store a memory block when the data field is occupied by the sharing code, the system must be adapted to work with this kind of cache organization. In the most straightforward implementation, ICCI could work along with an exclusive cache hierarchy just by applying minor modifications to the LLC management. Section 3.1 explains that implementation of ICCI in detail.

ICCI provides a huge directory compared to sparse directories. However overprovisioned, a reasonably sized sparse directory will always be smaller than the LLC, which means that directory-induced invalidations will be much less frequent in ICCI.

In addition, contrary to tag-embedded information and sparse directories, ICCI introduces no dedicated structures and no fixed directory overhead. Traditional directory schemes are designed to cover worst-case scenarios in both terms of number of entries and entry size, which makes them scale poorly. The fundamental idea behind ICCI is that the storage of directory information can be performed more efficiently (and more simply) if, rather than using a dedicated structure, the system takes up just the minimum amount of entries required from the LLC. As the number of cores increases, the number of directory entries really needed per core typically decreases (Section 5.1), making ICCI's approach scalable.

To conclude with the description of ICCI, we must note that the characteristics of ICCI can be seen from two complementary points of view, depending on how resources are assigned:

- Assuming a fixed LLC capacity. Contrary to tag-embedded or sparse directories, which in this case introduce large overheads for storing the sharing codes, ICCI introduces no extra overhead on the storage capacity of the chip.
- Assuming a fixed amount of resources that are shared between cache blocks and coherence information. In traditional schemes, these resources are split statically between the LLC and the structures storing coherence information at design time, resulting in a smaller LLC (and increasingly smaller as the number of cores rises). This is not the case in ICCI, where the LLC will be assigned all the resources and they will be used dynamically to store either data or coherence information. ICCI will adapt at runtime to the characteristics of the applications running, changing the percentage of the resources used for sharing information depending on the sharing patterns of the workload at each moment.

In both cases, the most appropriate scheme will be determined by the most efficient global usage of resources. We use the first point of view for the detailed evaluation of an implementation of ICCI against other proposals in Section 4. In Section 5, we use the second point of view to analytically show ICCI's favorable scalability properties compared to other coherence schemes, depending on the particular characteristics of the running applications.

3.1 ICCI LLC Management

Fig. 1 shows the state diagram for the LLC operation in a possible implementation of ICCI (assuming an exclusive cache hierarchy). An ordinary MOESI cache coherence protocol is assumed (others are possible). The states shown in the diagram correspond to the possible configurations of a

TABLE 1
Comparative of Directory Schemes

	Type	Directory Features: Dedicated per-tile overhead	Directory Features: Lookup latency	Directory Features: Directory-induced invalidations	Influence in the complexity of: Coherence protocol	Influence in the complexity of: Cache design
Tag-embedded	Flat	O(n)	O(1)	Low	Low	Low
Sparse directory	Flat	O(n)	O(1)	Medium	Low	Low
Hierarchical	Hierarchical	$O(\sqrt[k]{n})$	$O(\log n)$	High	High	Low
SCD	Flat	$O(\sqrt[k]{n})$	$O(\frac{n}{\sqrt[k]{n}})$	Low	Low	High (hierarchy on ZCache)
SCI	List-based	$O(\log n)$	O(n)	Medium	Medium	Low
Pointer Tree	Tree-based	$O(\log n)$	$O(\log n)$	High	High	Low
ICCI-full-map	Flat	none	O(1)	Low	Low	Low

block in the LLC: not present (*np* state), directory information stored in the LLC (*d* state), and block stored in the LLC (*b* state). These states are codified in the tag array of the LLC. Next, we give an explanation of these states.

When a block is fetched from main memory, an LLC entry is allocated for the sharing code (*d* state), and the block is only stored in the requesting core's L1 cache, which becomes the block owner (1). Other cores can get a shared copy of the block by sending a request to the LLC, which forwards the request to the owner L1 cache (2), which answers with the shared copy.

Only the owner core writes back the block data to the LLC upon eviction, while other cores' shared copies can be optionally replaced silently. When the owner replaces the block, the LLC asks another sharer (if any exists) to accept the ownership (3/4). Other sharers are known to the LLC thanks to the sharing code stored in the LLC. However, if a sharer has silently replaced its shared copy of the block, it will reject the ownership. In such a case, the LLC removes the former sharer from the sharing code and probes another sharer (5). This process is out of the critical path of L1 cache misses. If there are no sharers left, the sharing code stored in the LLC entry is not necessary anymore, and the evicted memory block reuses that LLC entry (6/7), transitioning from *d* to *b* state. Reusing the LLC entry also prevents LLC evictions upon L1 cache replacements.

When a core requests a block stored in the LLC, the block is sent to the requesting core (in exclusive state), and the LLC entry that contained the block is reused to store the newly generated sharing code (8), transitioning from *b* to *d* state. Again, the entry reuse mechanism prevents any LLC evictions. This reuse is important, since directory entry evictions are the cause of directory-induced invalidations.

In this implementation of ICCI, only main memory accesses (due to LLC misses, which are hopefully infrequent) cause LLC evictions of either a directory entry (9) or a block (10) in order to allocate a directory entry for the newly fetched block.

ICCI's LLC uses an ordinary pseudo-LRU replacement policy. LLC blocks never get their pseudo-LRU information updated in ICCI, because LLC accesses cause blocks to be substituted by directory entries (8). Only entries containing directory information have their pseudo-LRU information updated. Hence, data blocks are commonly evicted from the LLC before directory entries naturally. In addition, sharing codes are not evicted as long as there are candidate blocks for eviction. This makes ICCI work implicitly as the mechanisms proposed by Jaleel et al. [22] to bridge the performance gap between inclusive and non-inclusive caches, which in practice are meant to reduce directory-induced invalidations.

Alternatively to the operation just explained, the LLC could store both a memory block and its directory information in different cache lines (ways) of the same LLC set to enable non-inclusive or inclusive caches with ICCI. The changes required in the cache array and controller to carry out the management necessary for this are comparable in complexity to the management of a tag-embedded directory or the dedicated structure of sparse directories.

3.2 Contextualizing ICCI's Directory Scheme

Table 1 compares ICCI with other shared-memory organizations based on cache coherence directory protocols, showing that ICCI has the best features among them. Especially important is ICCI's ability to store directory information with no dedicated area overhead, unlike the rest of schemes. Also, ICCI's negligible number of directory-induced invalidations contrasts with most schemes, that at some degree suffer performance degradation due to these invalidations. Other good features of ICCI are the use of a simple flat protocol and ordinary caches, as well as its constant lookup latency, especially if compared with the closest alternative, SCD.

Finally, note the differences between ICCI and an apparently similar published proposal: AMD Magny-Cours' cache coherence [8]. When operating in non-coherent mode, the Magny-Cours uses all the ways of the cache to store blocks. When operating in coherent mode, the Magny-Cours allocates some ways of all cache sets to work as an ordinary directory cache. This is different from ICCI because while the Magny-Cours reserves resources to create a separate directory cache, ICCI selectively uses the minimum number of LLC entries possible to store sharing information.

3.2.1 Directory-Induced Invalidations

ICCI incurs a negligible amount of directory-induced invalidations. In ICCI, the eviction of directory entries from the LLC, causing L1 cache invalidations, is a rare phenomenon due to the much larger size of the LLC compared to the tracked L1 caches and the fact that data blocks are evicted from the LLC before directory entries. For instance, ICCI working on an LLC with eight times as many entries as the aggregate L1 caches works logically as an 800 percent coverage directory cache. Applying the analytic model proposed by Sanchez et al. [20] to ICCI, the maximum probability of evicting a cache entry with sharing information upon an LLC insertion is $(\frac{size_{L1}}{size_{LLC}})^{assoc_{LLC}}$. Assuming the previous

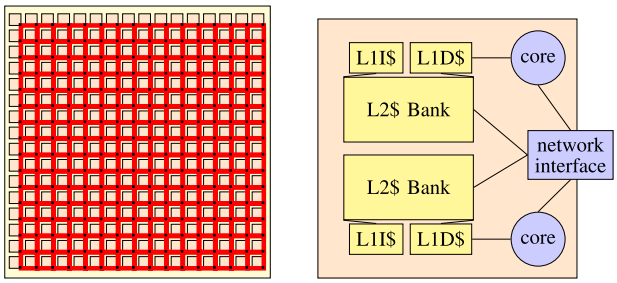


Fig. 2. 512-Core CMP. Two tiles (each containing a core, private L1 caches and a shared L2 cache bank) share each router of the 16×16 mesh network.

ratio between LLC and private caches and 8-way associativity in the LLC, the eviction probability is 6×10^{-8} for ICCI. As a comparison, SCD using overprovisioned 110 percent coverage 64-replacement-candidate ZCaches has a much larger 10^{-3} eviction probability, which is considered negligible by SCD's authors. Moreover, ICCI's eviction probability is applicable only upon memory accesses, because it is then that LLC insertions take place (see how entry reuse works in Section 3.1), while for SCD evictions take place upon more frequent LLC accesses (SCD entries are allocated for the directory information of the LLC blocks accessed by the cores, causing evictions). Note that one of the main contributions of SCD is its ability to bound the eviction probability by means of controlled overprovisioning thanks to ZCache's high associativity. We have shown that ICCI can do as good a job with no need for the (overprovisioned) complex ZCache-based SCD cache.

4 Evaluation

We used a simulator based on Pin [23] and GEMS [24] to perform the tests shown in this section. The chip components of GEMS were attached to a Pin tool to enable fast simulation of large numbers of cores. The methodology explained by Monchiero et al. [25] was used to obtain performance numbers.

4.1 Parameter Settings

We simulated a 512-core CMP (shown in Fig. 2) running at 2 GHz with a shared eight-way associative L2 cache based on a NUCA design (one 10-cycle access latency 256 KB L2 bank per core, 128 MB total) on a mesh network (every two tiles share a router), and four-way associative 16 KB data and instruction L1 caches (1-cycle access latency). The cache block size is 64 bytes. We call the capacity ratio between the shared cache and the aggregate private caches as the S/P ratio. The S/P ratio of our simulations was 8×. For main memory we assumed DDR4 technology [26], [27].

For reference, the cache sizes were set to those of Intel's SCC [14], which was designed to scale out to hundreds of cores. Intel's SCC measures 567 mm^2 at 45 nm with a 125 W TDP. Our assumed 512-core could be realized in 585 mm^2 at 14 nm with similar power consumption.

TABLE 2
Simulated Machine

Processors	512 x86 cores @ 2 GHz, 2-ways, in-order
L1 Cache	Split I&D. Size: 16 KB, 4-ways, 64 bytes/block Access latency: 1 cycle MOESI coherence protocol
L2 Cache	Size: 256/128/64 KB per bank (NUCA) 16-ways, 64 bytes/block Access latency: 10 cycles Directory cache lookup: 2 cycles, SCD cache lookup: 1 cycle
RAM	16 GB DDR4 DRAM 16 3D-stacked memory controllers
Interconnection - Mesh	2 GHz, 2D mesh: 16×16. Express links every 4 routers 16 byte links Latency: 1 cycle/link, 3 cycles/express-link 4-cycle pipelined routers Flit Size: 16 bytes Control/Data packet size: 8/72 bytes (1/5 flits)

For energy calculations, we used McPAT [28] assuming a 22 nm process and scaled the resulting figures down to 14 nm. Simulations were configured with the values generated by McPAT. Table 2 summarizes the characteristics of the simulated machine.

We evaluated four different schemes. Two of them are common area-consuming (non-scalable) directory coherence schemes to use as baselines: a tag-embedded directory (TAG) and a 200 percent coverage sparse directory (SPARSE). The other two are the scalable directory coherence proposals we intended to compare: a 110 percent coverage SCD and ICCI assuming an exclusive cache hierarchy. Table 3 shows the simulated directory schemes and the overhead of their associated extra resources. Note that due to the difficulty in simulating and implementing arbitrary cache sizes (i.e., sizes not power of 2), we did not fix the overall amount of area resources and derive the sizes of the LLC and directory storage. Instead, we fixed the LLC size and added the extra resources needed by each directory scheme. Notice that ICCI is the only alternative that requires no additional hardware resources, while the tag-embedded directory requires the most extra resources.

The 110 percent coverage SCD uses 52-replacement-candidate ZCaches. We used ordinary 200 and 110 percent coverages for the sparse directory and SCD to reduce directory-induced invalidations to a negligible number [8], [20]. The sparse directory needs a higher coverage to achieve a similar number of directory-induced invalidations to SCD, as SCD takes advantage of the higher associativity of ZCaches. The tag-embedded directory uses an inclusive cache hierarchy, while the sparse directory cache and SCD allow for a non-inclusive hierarchy. The implementation of ICCI used is the one described in Section 3.1. Both TAG and ICCI implement a suitable LLC replacement algorithm to minimize the performance loss of inclusive caches with respect to non-inclusive caches [22]. Evictions of shared blocks are notified to the directory in SPARSE and SCD to prevent stale sharers from polluting the directory caches and generating a large number of directory-induced invalidations

TABLE 3
Directory Size Requirements for the Schemes Tested

S/P ratio	Embedded-tags	200%-coverage sparse directory	110%-coverage SCD	ICCI
8×	729%	200%	15%	0%
4×	364%	200%	15%	0%
2×	182%	200%	15%	0%

Size is given as a percentage of the aggregate capacity of the tracked caches, assuming a 512-core CMP and 64-byte lines.

TABLE 4
SPLASH-2 Program Sizes

Benchmark	Original problem size (maximum 64 cores)	Scaled problem size (512 cores)
Barnes	16K particles	256K particles
Ocean_cp	258×258 grid	2048×2048 grid
Ocean_ncp	258×258 grid	2048×2048 grid
Volrend	ROTATE_STEPS=4	ROTATE_STEPS=100
Water_ns	512 molecules	8K molecules
Water_s	512 molecules	32K molecules
Cholesky	tk29.O	tk29.O
FFT	64K points	1M points
LU_cb	512×512 matrix	2048×2048 matrix
LU_ncb	512×512 matrix	2048×2048 matrix
Radix	256K integers	64M integers

that would harm performance noticeably, as our observations confirmed.

To understand the results of the evaluation, the differences between the cache hierarchies used must be taken into account. The implementation of ICCI incurs the most three-hop misses (as many as an ordinary exclusive cache), because the memory blocks that are already present in the L1 caches are not stored in the LLC. Also, clean blocks evicted from L1 caches are written back to the LLC (transitions 6 and 7 in Fig. 1), like in the non-inclusive cache (SPARSE and SCD), in which we do not store blocks in the LLC while exclusively owned by an L1 cache to increase the LLC effective capacity. Finally, in practice, regarding the number of memory accesses this implementation of ICCI works as an inclusive cache with the same LLC size (TAG), because blocks in L1 caches require an LLC entry for the sharing code, while the non-inclusive cache provides a higher effective total cache capacity for the same LLC size (thanks to storing the directory separately in additional area resources). Remember that ICCI can be implemented along with a non-inclusive cache at the cost of some extra complexity, resulting in different tradeoffs.

This implementation of ICCI removes the need to access a directory cache in addition to the L2 tags. Note that the directory cache causes extra energy consumption if accessed in parallel to the LLC or extra latency if accessed sequentially after the LLC. We have considered parallel accesses to maximize performance.

We ran benchmarks from the SPLASH-2 suite appropriately scaled up for 512 cores, making them able to stress the 128 MB LLC. Table 4 compares the original SPLASH-2 input sizes recommended for up to 64 cores and the scaled-up input sizes used in our experiments for 512 cores. In Section 4.3.1, we also show results for the PARSEC 3.0 benchmarks.

We considered the use of unicast or multicast networks in the simulated 512-core chip. Our preliminary results showed that unicast communication causes performance to drop in all benchmarks compared to using efficient one-to-many and many-to-one communication, as noted by Ma et al. [29]. Seven out of 11 evaluated benchmarks increased their execution by 50 percent at least when using unicast communication. The least affected benchmark was FFT, which still showed a 5 percent increase in execution time. The main cause is the slow invalidation of highly shared blocks, which becomes a bottleneck and increases the pressure on the network creating hot spots. While multicast can gracefully deal with invalidations to many cores, which are especially important for efficient thread synchronization (e.g., barriers and locks), unicast requires the origin of invalidations to send up to 511 unicast messages and process up to 511 response messages, becoming a fatal bottleneck for performance, as evidenced by our results. For its superior performance, we chose a network with efficient multicast request sending and response collection to evaluate the four directory schemes [29].

4.2 Results for 8 × S/P Ratio in SPLASH-2

4.2.1 Execution Time

The top graph of Fig. 3 shows the execution time of the SPLASH-2 benchmarks. The results are normalized to ICCI. The central graph of the figure gives insight into how time is spent on L1 cache misses, showing the main differences between the four evaluated directory organizations.

In general, ICCI works similarly to the slower cache hierarchy in each benchmark: inclusive (TAG) or non-inclusive (SPARSE). ICCI suffers as many extra memory accesses as the inclusive cache used by TAG (because ICCI uses the least resources, see Table 3), increasing execution time in Ocean, FFT and Radix. ICCI suffers even more extra three-hop accesses than the non-inclusive hierarchy used by SPARSE, increasing the execution time of Barnes, Volrend and Water. Volrend is the best benchmark for TAG, with 7 and 8 percent faster execution than SPARSE and ICCI at an 8× S/P ratio, due to the difference in number of three-hop misses. At an 8× S/P ratio, ICCI performs less than 2 percent worse than the fastest coherence scheme in eight out of 11 benchmarks.

As for SCD, it performs similar to SPARSE since both use a non-inclusive hierarchy. This is an advantage over ICCI when memory accesses make up most of the execution time, like in Radix, as SPARSE and SCD reduce the LLC miss rate compared to TAG and ICCI.

SCD's weak point is the multiple sequential directory lookups required to reconstruct the sharing vector that take place in the critical path of cache misses. The effect of these accesses is especially harmful when they take place in critical events such as barriers or contended locks, affecting the performance of many cores, increasing the inefficiency of thread synchronization. This makes SCD results deviate from those of SPARSE in several benchmarks. This is the case in Ocean, Volrend and LU, in which SCD shows degraded performance, with up to a 10 percent slowdown in Volrend compared to ICCI.

In broad terms, both ICCI and SCD perform reasonably close to the non-scalable tag-embedded directory and sparse directory, with the performance differences just described. Their worse performance in some cases can be justified because they use far fewer resources than the non-scalable organizations (see Table 3). SCD's degraded performance in some benchmarks (ICCI beats SCD in eight out of 11 benchmarks) as well as ICCI's simplicity and smaller area are the main arguments in favor of ICCI in this comparison.

These results also suggest that the cache hierarchy used with ICCI does not affect execution time very negatively compared to other hierarchies, obtaining better results than other scalable alternatives like SCD. We went into detail to investigate why this is so, and summarize our findings next.

First, in ICCI as well as in inclusive caches (TAG), most of the LLC resources are used to store memory blocks not

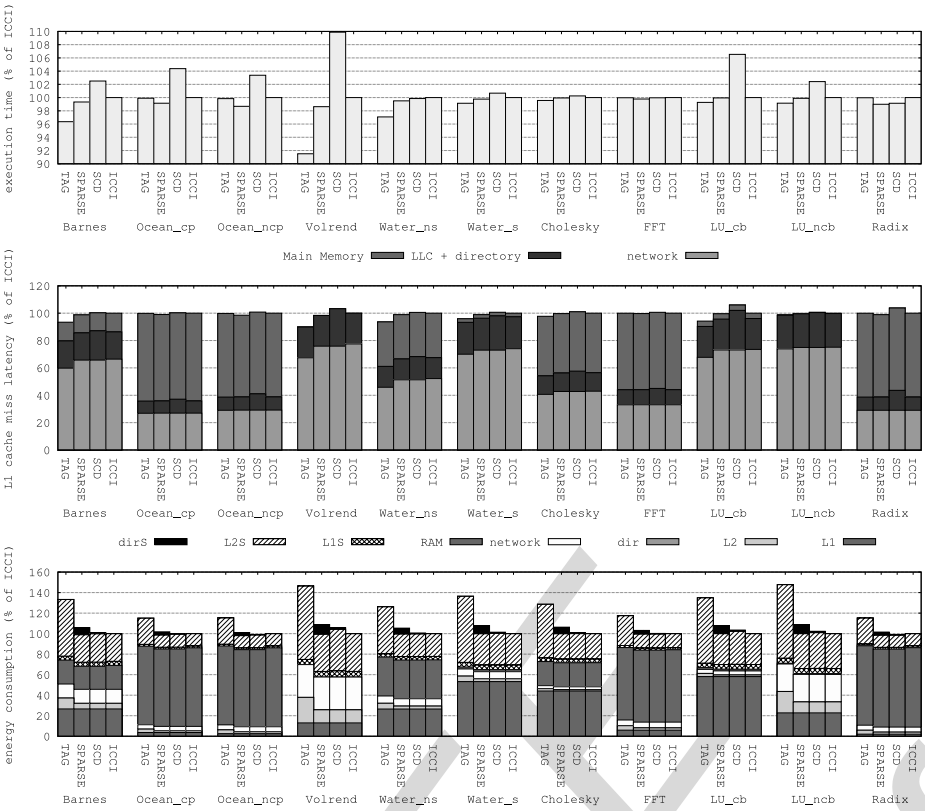


Fig. 3. Results for 8× S/P ratio in SPLASH-2. From top to bottom: execution time, average L1 cache miss latency and energy consumption.

present in the L1 caches. In general, the total amount of L1 cache entries is much smaller than that of LLC entries, hence only a small percentage of the LLC is used for blocks present in the L1 caches (or for sharing information in ICCI). In addition, the least re-referenced blocks of the exclusive LLC are the ones not present in the inclusive LLC, and these cause few extra memory accesses. Hence, the LLC miss ratio increase is small. Table 5 compares the LLC miss rate of the exclusive cache used with ICCI and the non-inclusive cache used with SCD. In general, some benchmarks have a working set that fits in both the non-inclusive LLC and ICCI, both yielding the same miss rate (Volrend, Water, LU). Some benchmarks have large working sets and the miss rate of ICCI is higher. The rest of results agree with the commonly accepted empirical observation that LLC miss rate is approximately inversely proportional to the square root of the effective size of the LLC cache [30].

In addition, prior work showed that the performance gap between inclusive and non-inclusive caches is caused by inappropriate LLC replacement-information in inclusive caches, rather than by the difference in effective LLC capacity, and appropriate replacement policies can bridge that gap almost completely [22]. ICCI was designed in such a way that an appropriate update mechanism is part of its operation (as was shown in Section 3.1).

Second, most three-hop misses are caused by accesses to blocks currently owned by L1 caches, and take place also in the non-inclusive caches. As a result, the difference in the amount of three-hop misses between ICCI and the non-inclusive caches is small, as the results in Table 6 show. When three-hop misses are abundant, the performance of the non-inclusive caches and ICCI can degrade compared to the inclusive one (TAG). This becomes evident in Barnes and Volrend when comparing the execution time and L1 miss latency shown in Fig. 3.

Third, ICCI reduces the amount of data writebacks compared to the non-inclusive caches, as can be seen in Table 7. The explanation to this lies in ICCI's replacement mechanism, which transfers the ownership to another sharer upon an owner replacement. This means that data is not written

TABLE 5
LLC Miss Rate for SPLASH-2

	8x S/P ratio			4x S/P ratio			2x S/P ratio		
Benchmark	SCD (non-Inclusive)	ICCI (exclusive)	Increase	SCD (non-Inclusive)	ICCI (exclusive)	Increase	SCD (non-Inclusive)	ICCI (exclusive)	Increase
Barnes	3.3%	3.4%	3.8%	4.1%	4.2%	4%	5.2%	5.5%	4.1%
Ocean_cp	35.3%	35.8%	1.5%	49.6%	51.7%	4.1%	66%	70.3%	6.6%
Ocean_ncp	30.9%	31.6%	2.4%	43%	45%	4.8%	52.4%	55.9%	6.6%
Volrend	0%	0%	0%	0%	0%	0%	0%	0%	0%
Water_ns	10.5%	10.5%	0.5%	10.9%	11%	0.7%	11%	11.2%	1.2%
Water_s	0.6%	0.6%	0%	0.6%	0.6%	0%	0.6%	0.6%	0%
Cholesky	16%	16%	0.3%	18.4%	18.6%	1.1%	19.6%	20.4%	3.8%
FFT	25.3%	25.4%	0.5%	28.7%	29.1%	1.5%	50.6%	52.5%	3.9%
LU_cb	0.9%	0.9%	0%	0.9%	0.9%	0%	2.3%	2.4%	3.3%
LU_ncb	0.1%	0.1%	0%	0.5%	0.5%	0%	0.8%	0.8%	2%
Radix	31.1%	31.7%	1,8%	41.8%	43.7%	4.6%	54.7%	59.9%	9.4%

For TAG (inclusive) and SPARSE, it is approximately equal to ICCI and SCD, respectively.

TABLE 6
Three-Hop Misses for SPLASH-2 as a Percentage of All L1 Cache Misses

Benchmark	Non-Inclusive	ICCI (exclusive)	Increase
Barnes	59.1%	66.2%	11.9%
Ocean_cp	2.4%	2.5%	4.8%
Ocean_ncp	2.8%	3.5%	26.3%
Volrend	75.5%	89.9%	19.2%
Water_ns	71.5%	83.6%	16.8%
Water_s	26.4%	35.2%	33.3%
Cholesky	31.4%	34.4%	9.5%
FFT	0.3%	0.3%	0.4%
LU_cb	47.6%	51.2%	7.7%
LU_ncb	8.5%	9.8%	15.3%
Radix	0.5%	0.6%	13%

TABLE 7
Percentage of L1 Evictions That Cause a Data Writeback in SPLASH-2

Benchmark	Non-Inclusive	ICCI (exclusive)	Decrease
Barnes	40.8%	33.8%	17.3%
Ocean_cp	97.6%	97.4%	0.2%
Ocean_ncp	97.1%	96.4%	0.8%
Volrend	24.5%	9.9%	59.2%
Water_ns	28.4%	16.4%	42.4%
Water_s	74.4%	65.2%	12.3%
Cholesky	68.9%	65.7%	4.6%
FFT	99.6%	99.6%	0%
LU_cb	52.3%	48.7%	7%
LU_ncb	90.4%	89.1%	1.5%
Radix	99.5%	99.4%	0.1%

back as long as there are sharers remaining in the chip. We found that ICCI generates traffic closer to the inclusive cache in this regard.

4.2.2 Energy Consumption

The bottom graph of Fig. 3 breaks down the energy consumption of the memory system (including the interconnection network) normalized to ICCI. Static and dynamic energy consumption is taken into account for the caches, network-on-chip and RAM. Cache energy is broken down into static and dynamic energy to analyze the detailed effects of the directory area overhead, while RAM and network energy are not broken down for clarity.

These results show the inefficiency of embedded-tag directories and sparse directory caches. TAG approximately doubles the static energy of the LLC compared to ICCI, due to the directory information that is as large as the associated LLC block. The L2 cache dynamic energy of TAG also increases due to the larger tags, raising its total energy consumption even more. This results in increases in the overall energy of the memory system of up to 48 percent with respect to ICCI. SPARSE reduces the static energy compared to TAG thanks to the smaller area overhead of the directory cache. However, SPARSE still increases energy consumption in general when compared to ICCI, and does so by up to 15 percent in Volrend. SCD reduces the static energy further compared to SPARSE, but its area overhead still makes it more energy consuming than ICCI.

ICCI is the least energy consuming alternative at an 8× S/P ratio, outperforming SCD in most benchmarks, and doing so clearly on those with moderate RAM usage. ICCI's lower execution time when SCD suffers from directory serialization causes the largest energy differences. This is specially noticeable in Volrend, where ICCI reduces energy by 8 percent compared to SCD. ICCI consumes as much RAM energy as TAG, which is more than SPARSE and SCD, but the absence of area overhead clearly makes up for the increased RAM energy consumption. ICCI never consumes more energy than SCD at this S/P ratio.

4.3 Results for Lower S/P Ratios in SPLASH-2

To explore the effects of ICCI's increased pressure on the LLC for different S/P ratios, we simulated two smaller LLC sizes, maintaining the L1 cache size. In particular, we tested per-core L2 cache sizes of 128 KB (S/P ratio of 4×) and 64 KB (S/P ratio of 2×). For these ratios, the overhead of the coherence schemes tested can be found in Table 3. As a comparison, Intel's Sandy/Ivy Bridge microarchitectures have an inclusive shared L3 cache (8MB total) and private non-inclusive L2 and L1 caches (256 KB plus 32+32 KB per core, with four cores), resulting in an S/P ratio that varies from 8× to around 6.4×, depending on the degree of inclusivity at runtime between L2 and L1 caches. Note that a ratio of 2× should be rare and it is included just as a worst case scenario for this implementation of ICCI.

Results for 2× S/P ratio are shown in Fig. 4. They are in line with the results for the 8× S/P ratio previously

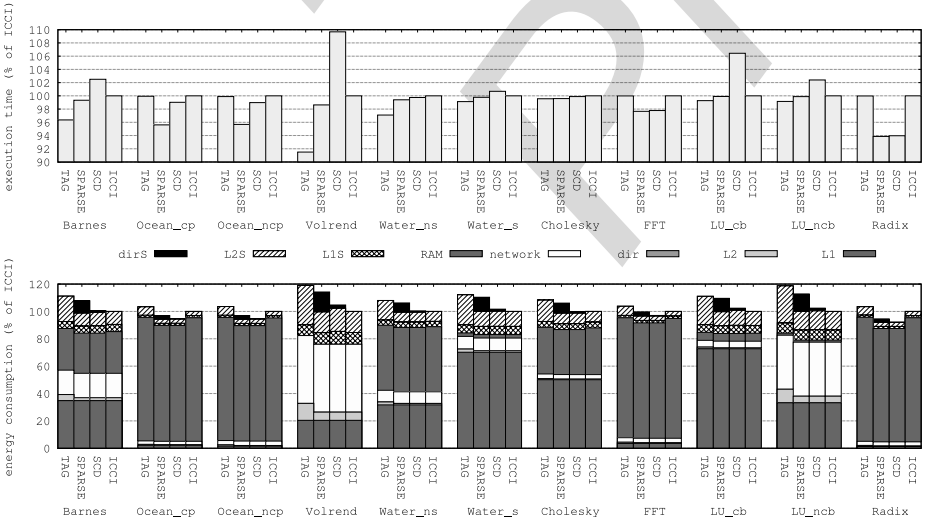


Fig. 4. Results for 2× S/P ratio in SPLASH-2. From top to bottom: execution time and energy consumption.

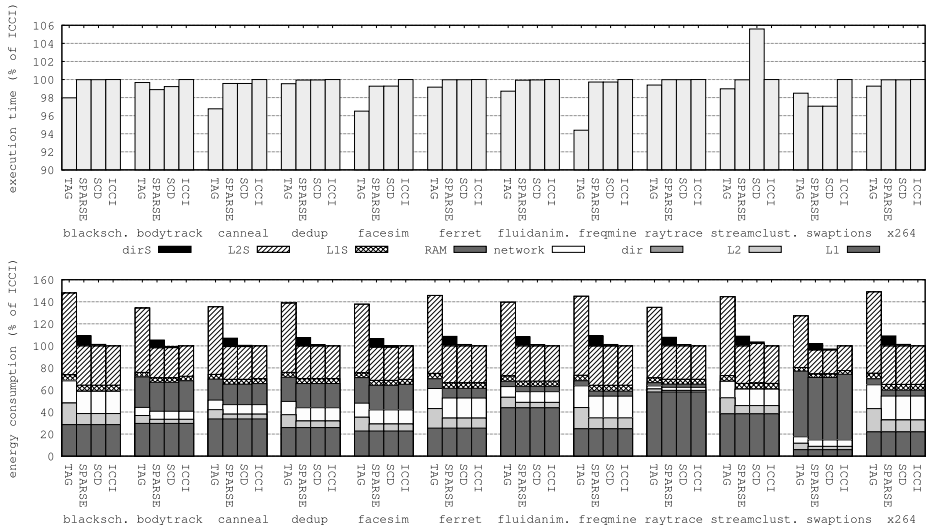


Fig. 5. Results for 8× S/P ratio and PARSEC 3.0. From top to bottom: execution time and energy consumption.

discussed, and they also show the increased effect in LLC pressure of TAG and ICCI. The worst benchmark for ICCI (and TAG) is Radix, which executes 6 percent more slowly than with SPARSE and SCD at a 2× S/P ratio, due to extra memory misses.

Table 5 shows that, as the S/P ratio decreases, the miss ratio difference increases between the non-inclusive caches (SPARSE and SCD) and ICCI, as expected. But even with a 2× S/P ratio, the largest miss ratio increase is 9.4 percent in Radix. Those benchmarks with the largest increases are those in which the inclusive cache performs worse, with ICCI also suffering from higher LLC miss ratios.

In terms of energy, as the S/P ratio goes down, ICCI struggles to keep up in memory demanding benchmarks, with a clear 7 percent energy increase over SCD in Radix at a ratio of 2× as the worst result for ICCI. Nevertheless, even with a 2× S/P ratio, ICCI reduces energy consumption in six out of 11 benchmarks compared to SCD. These results for such an unusually small S/P ratio and RAM demanding benchmarks show that ICCI makes a good job in containing energy consumption in very adverse conditions. Also, SPARSE power consumption gets closer to TAG as the S/P ratio goes down due to the shrinking difference in overhead between both schemes.

4.3.1 Results for PARSEC Benchmarks

As said before, even though ICCI is not tied to a particular inclusivity policy in the cache hierarchy, exclusive caches were chosen to exemplify an implementation of ICCI because they seem the most natural match for ICCI. Any other cache hierarchy, such as non-inclusive or inclusive, could be used along with ICCI to trade off between three-hop misses and memory accesses, but the analysis of such implementations is left for future work.

In this section, we evaluate the directory schemes with the PARSEC 3.0 benchmark suite. This suite presents significantly more data sharing than SPLASH-2, resulting in more three-hop misses in exclusive cache hierarchies like the one used along with ICCI in this article, possibly degrading performance compared to non-exclusive hierarchies. For these experiments, due to limitations in the available input data sets of the PARSEC benchmarks, we had to limit the configuration setting for these programs to 64 threads with the SIMLARGE input size, even though we simulate a 512-core chip, which means that only 64 cores in the chip do useful work.

We observed that, in our machine configuration, the percentage of L1 cache misses that are three-hop misses increases over 40 percent on average in PARSEC as compared to SPLASH (although the benchmark with the most three-hop misses is Water_ns from SPLASH-2). Also, when compared to the non-inclusive cache hierarchy used along with SPARSE and SCD, the average increase of three-hop misses caused by the exclusive hierarchy used in this ICCI implementation (measured in percentage points) almost doubles in PARSEC.

Fig. 5 shows the execution time and energy consumption results of these experiments. In general, we can see that our previous findings also hold for this scenario, suggesting that the increase in three-hop misses does not degrade performance excessively. It is important to note that the effect of three-hop misses is reduced in this scenario due to the smaller number of retransmissions required by the messages involved because of the limited number of threads used, which are placed closer together. As in the previous sets of experiments, ICCI does not need the energy consumed by the extra resources required by other schemes, while providing similar execution times.

Table 8 shows the storage overhead due to directory information measured for these benchmarks with ICCI, as the average of samples taken every million instructions, measured for those 64 cores doing useful work. These low

TABLE 8
Size of the Directory Information Stored in the LLC in ICCI Normalized to L1 Capacity, for the PARSEC 3.0 Benchmarks

Benchmark	Directory size	Benchmark	Directory size
Blackscholes	2.1%	Fluidanimate	2.7%
Bodytrack	2.5%	Freqmine	2.3%
Canneal	2.7%	Raytrace	2.9%
Dedup	2.3%	Streamcluster	1.3%
Facesim	2.2%	Swaptions	2.4%
Ferret	1.7%	x264	1.1%

values are a result of the amount of shared data, which we have measured to be noticeably higher in PARSEC than in SPLASH, and the number of sharers per shared block. The average resource consumption of ICCI for directory entries (around 2 percent) is far lower than that of the dedicated resources used by TAG (729 percent), SPARSE (200 percent) or SCD (15 percent), showing that it makes a more efficient usage of on-chip storage resources.

Another important result follows from these experiments: when a subset of the cores (or even a single core) uses all the LLC resources, the entries used by ICCI for directory information are just those required for the private caches actually in use, which contrasts with TAG, SPARSE and SCD, where the directory needs to be sized statically for the worst case, i.e., assuming that the all the cores are doing useful work.

These results suggest that an exclusive implementation of ICCI with the same amount of resources and the same cache coherence protocol as the other directory schemes would not have significant problems due to any increases in three-hop misses, and would use far less resources for directory information, resulting in a noticeably larger effective LLC capacity.

5 Exploratory Analysis of ICCI's Resource Usage

The efficiency in the usage of resources by different directory schemes can be directly compared by measuring the storage resources taken up by each of them, in a way that the variations in the rest of the storage-dependent characteristics of the memory organization are eliminated (e.g., LLC miss rate, three-hop misses). For instance, separate directories (such as SPARSE) take up a part of the resources, at design time, reducing the available space for the LLC. In contrast, ICCI takes up entries from the LLC on demand. The scheme using the least resources for directory information in practice will be the most beneficial from a point of view of storage, as in the end it will leave more resources available for other elements of the chip.

To do this, first we need to distinguish between two types of directory coverage. We define the term *effective coverage* as the percentage ratio of the minimum number of directory entries required for tracking all the data in the private caches to the total number of tracked private-cache entries (e.g., a 100 percent effective coverage means that each private-cache entry requires one directory entry, and a 50 percent effective coverage means that, on average, every two private-cache entries contain the same block and are tracked by the same directory entry, hence requiring half as many directory entries as a 100 percent coverage). By definition, the effective coverage can never rise over 100 percent. On the other hand, we define the *physical coverage* of a directory as the ratio of the number of entries allocated for storing sharing information to the number of private-cache entries, whether the directory entries are currently in use or empty. The physical coverage of separate directories typically rises over 100 percent, like in 200 percent coverage sparse directories [8].

The flexible allocation of entries in ICCI makes its physical coverage dynamically match the effective coverage at all times. For instance, if all blocks are widely shared by all the cores of a 512-core chip, ICCI's physical coverage becomes just $\frac{1}{512}$ of its physical coverage when all blocks are private. This also implies that the resources taken up by ICCI to store directory information vary dynamically. In other words, ICCI resizes the directory at runtime to match the minimum necessary coverage required by the current workload, while the rest of resources are used to store data blocks. This is not possible in separate directories such as SPARSE or SCD, where the physical coverage is fixed at design time, sizing the directory to fit every possible worst-case scenario. Fixed-size directories result in large amounts of resources assigned to a directory that will rarely be highly used at runtime, when effective coverages over 100 percent can never take place, and (much) smaller effective coverages often take place, especially as core count rises. In these rigid schemes, varying effective coverages simply translate into varying occupation rates of the fixed-size directory (i.e., the amount of directory entries that actually store sharing information). And despite whether the entries of the directory store sharing information or not at runtime, they cannot be used for other purposes (to our knowledge, no proposals do such thing yet), ending up wasted. Maybe the biggest contrast is that, while in ICCI the physical coverage never rises over 100 percent, in fixed-size directories physical coverage never goes under 100 percent (and a 100 percent coverage directory is optimistically small and will in all probability yield bad performance due to conflicts that cause directory-induced invalidations).

On the other hand, it can be argued that ICCI wastes more space inside each directory entry compared to SPARSE. This is especially true for low core counts, when an LLC entry is obviously much bigger than a sharing vector (more on this and how to fix it in Section 5.2). First, this is not a particular problem of ICCI. In SPARSE, space is also wasted inside entries (e.g., if a large full-map bit-vector stores just one sharer, when a pointer would suffice). ICCI just accentuates this for low core counts, which does not imply that ICCI performs badly with small core counts. In fact, additional detailed simulations show that ICCI routinely outperforms a 200 percent coverage sparse directory even for just 16 cores in both execution time and energy consumption (even though the sparse directory uses extra resources for the separated directory). This is so thanks to ICCI's better directory-specific features such as lower number of directory-induced invalidations. Second, ICCI and SPARSE become more and more similar in entry size as the core count rises and the sharing code size approaches the LLC entry size.

Orthogonally to the detailed simulation perspective provided earlier, in this section we perform a wide exploratory survey of possible runtime characteristics to evaluate the real performance of ICCI in terms of area (i.e., the number of entries dynamically taken up for directory information), and we base our final assessments on typical effective coverages found in the literature. We carry out a theoretical analysis of the characteristics of ICCI, focusing on a system with a fixed amount of storage resources and evaluating the distribution of these resources between data and sharing information by a number of coherence schemes, with special emphasis on their scalability when scaling out CMPs to

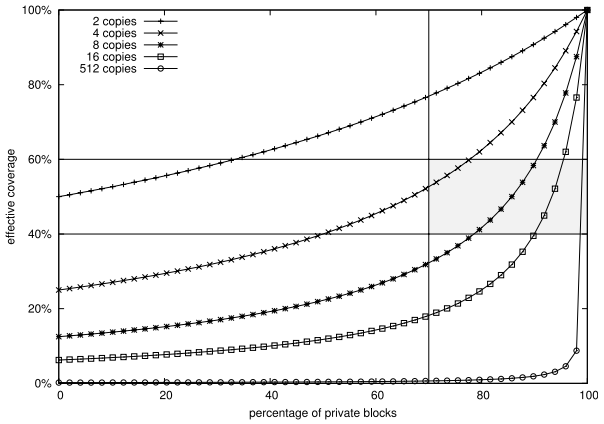


Fig. 6. Effective coverage depending on memory block sharing characteristics.

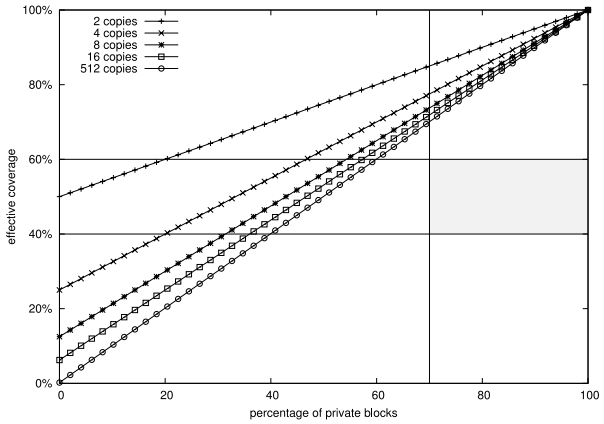


Fig. 7. Effective coverage depending on private-cache block sharing characteristics.

large core counts. The particular results will be very dependent on effective coverages and directory entry size.

This analysis only takes into account the area overhead of the evaluated directories. It does not measure other characteristics of the directory scheme such as energy consumption (e.g., huge in duplicated-tag directories for large core counts), latency (e.g., huge in SCI to go through the list of sharers upon invalidations), or other performance considerations (e.g., directory-induced invalidations, in whose prevention ICCI easily beats the rest). As ICCI forces no particular sharing code, we use a full-map bit-vector in the analysis for small core counts (up to 512 cores), and more elaborate sharing codes, in particular SCD's hierarchical code, for very large core counts (up to 256K cores).

After this evaluation, we conclude that ICCI has good scalability properties in terms of area compared to other schemes. With feasible small coverages, ICCI's directory storage space is comparable to schemes such as SCI or duplicate-tag directories. In addition, in the worst case scenario for ICCI, when the effective coverage is 100 percent, it is still much more area-efficient than SPARSE for large core counts, using the same full-map bit-vector sharing code. Also, we will explore ways to turn this worst-case scenario into a best-case scenario (with potentially 0 percent area overhead) thanks to ICCI's dynamic coverage that can leverage complementary coherence mechanisms.

5.1 Effective Coverage Analysis in Typical Scenarios

Effective coverages are completely workload-dependent, and they vary with the number of unique memory blocks stored in the private caches of the chip. Each of these memory blocks requires (at least) one directory entry to track all the copies of the block stored in the private caches (it might require more entries in composable schemes such as SCI). In general, the instantaneous effective coverage in a CMP can be characterized by means of the percentage of blocks that are private (only one copy of the block exists in the private caches), and the average number of copies of each of the remaining (shared) blocks (which by definition will be two or more). The higher the percentage of shared blocks and number of copies, the smaller the effective coverage. Smaller coverages will benefit ICCI, as its storage overhead is proportional to the effective coverage.

When calculating effective coverages, it is important to distinguish between memory blocks (of which several copies may exist) and private-cache blocks (several of which may be copies of the same memory block). This differentiation results in two possible methodologies when characterizing the private/shared block percentages, depending on whether the percentages refer to memory blocks or to private-cache blocks. Figs. 6 and 7 show the effective coverages resulting when using each of these two alternative methodologies, respectively, for varying average number of copies for shared blocks. Even though the usual way to calculate these percentages is by considering memory blocks [3], [31], [32], [33], [34], [35], we also show the calculations when considering private-cache blocks for completeness and to avoid ambiguity, as our graphical analysis (using conservative sharing values) will provide similar insights regardless of the methodology used.

To illustrate the differences between these methodologies, consider that, when counting private-cache blocks, each shared block is counted repeatedly, once for each copy of the block. With this methodology, the same runtime scenario would yield much smaller percentages of private blocks than if counting memory blocks (where shared blocks are counted just once), altering many of the values reported in the literature. Conversely, the same percentage of private blocks has different meanings depending on the methodology used. When counting private-cache blocks, having 90 percent of private blocks involve an effective coverage over 90 percent. This is not the case when counting memory blocks, as the remaining 10 percent of shared blocks may have many copies, resulting in a very small effective coverage. Take a combination of 90 percent of private blocks and 16 copies for shared blocks. When considering private-cache blocks, the effective coverage is over 90 percent. When considering memory blocks, the effective coverage is under 40 percent. Even though using the same percentages and number of copies, the two methodologies result in two very different execution scenarios.

It is generally agreed that, especially in scientific applications, just a small percentage of memory blocks are shared, part of which are typically widely shared. For many workloads, it has been empirically observed that a

majority of memory blocks are private to particular cores, with typical percentages ranging between 70 and 100 percent [36], [37], [38], [39]. It has also been reported that on average just around 16 percent of the memory blocks stored in private caches are shared in the PARSEC benchmark suite [40]. Nevertheless, applications exist with a myriad of footprints, including high percentages of shared blocks. It is also widely agreed that this is typically the case of commercial workloads (in contrast to scientific workloads), as reflected by the following percentages of shared memory blocks that have been reported for some commercial workloads: 49 percent [34], 34 percent [33], 50 percent [39], 58 percent [41] in apache; 49 percent [34], 48 percent [39], 62 percent [41] in oltp; 29 percent [34], 38 percent [33] in zeus.

In addition, effective coverages around 40-60 percent are commonly reported in the literature [3], [20] for scientific applications, some of them as low as 20 percent [20]. These small coverages have been leveraged by the use of complex hashing functions to reduce the amount of directory-induced invalidations with low physical coverages (close to or as low as 100 percent) [3], [20]. The combination of typical percentages (private block percentages over 70 percent and effective coverages between 40-60 percent) results in the highlighted rectangle on Fig. 6. We can see in the figure that those typical percentages are compatible with many possible values of average number of copies for shared blocks. As a curiosity, note that these typical private block percentages (over 70 percent) and effective coverages (many under 70 percent) do not match with the values in Fig. 7 calculated counting private-cache blocks (i.e., the highlighted rectangle is empty). If the same execution scenarios of the highlighted rectangle of Fig. 6 were plotted in Fig. 7, the percentage of private blocks would be obviously smaller, as pointed out earlier.

In general, as the core count rises, more opportunities for (wide) sharing arise [42]. This observation has been taken into account in several recent works that evaluate the effects of data sharing in multiprocessor design [43], [44]. In particular, as data sharing rises along with core count, it has increasingly critical impact on the miss rate differences between private caches that replicate shared data and shared caches that only store one copy of each memory block. Likewise, increases in data sharing make effective coverages decrease notably, as one directory entry is enough to track all the copies of a memory block. Notice that no sharing is possible in a chip containing a single core, which would always present a 100 percent effective coverage; for two cores, sharing may exist and as a result the effective coverage can range between 50 and 100 percent; as soon as we move up to 16 cores or 64 cores, effective coverages can be as low as 6.3 and 1.6 percent, respectively. Note in Fig. 6 that, even assuming that 70-90 percent of blocks are private, if shared blocks have 16 copies on average, the effective coverage already ranges between 20 and 40 percent. If private data is less than 70 percent (as is typical in commercial workloads) or if the average number of copies of shared blocks is higher (which can be common in any parallel application with high levels of sharing), the effective coverage will be much lower. With such small effective coverages, ICCI will take up very little per-core storage space, while the overhead of separate directories is insensitive to any of these circumstances.

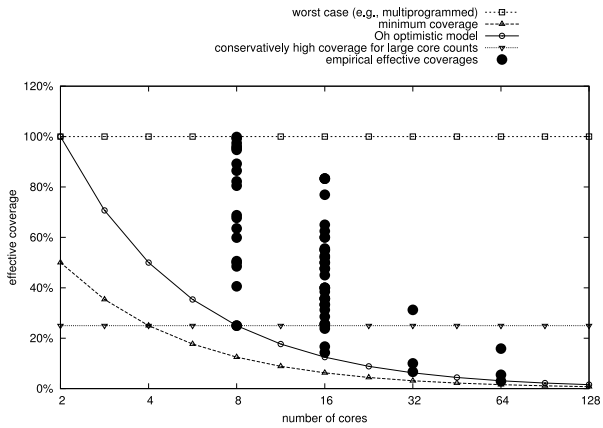


Fig. 8. Survey of effective coverages.

In fact, even small numbers of widely shared blocks result in low effective coverages. One of the elements that can reduce effective coverages notably is program code widely shared in parallel applications. In Fig. 6 we show, just as a very optimistic reference, the coverage required when shared data is widely shared by 512 cores on average. In this case, even with 90 percent of blocks being private, the effective coverage is just 1.9 percent.

In Fig. 8, we show a survey of effective coverages, including a considerable amount of empirical effective coverages found in the literature. In this figure, we show in semi-logarithmic scale the highest possible coverage (i.e., 100 percent, one directory entry per private-cache entry), that remains constant regardless of core count. We also show the minimum possible coverage, which takes place when each directory entry tracks as many sharers as there are cores in the chip (e.g., under 1 percent for 128 cores, with each directory entry tracking 128 sharers). In addition, the following relevant information is plotted:

- *Optimistic coverages.* Oh et al. [45] explore the optimal area breakdown between caches and cores for CMPs. In their analytical model, they optimistically assume that half of the cores share every memory block on average, based on their experience. In this model, the effective coverage at 512 cores would be just 0.4 percent. With this value, ICCI would take up just one-twelfth as much area as a duplicate-tag directory. In fact, ICCI would take up less area than duplicate-tag directories with just 40 cores. Even though possible (and confirmed by some empirical values, as we will see next), we consider this model too optimistic to be considered of general applicability.
- *Empirical coverages.* We have reviewed the literature and collected data from previous studies where the necessary information to calculate effective coverages was available [3], [31], [32], [33], [34], [35]. All these data (61 values in total) are plotted in Fig. 8, showing the effective coverages observed in a wide range of scientific and commercial workloads, from 8 to 64 cores. For eight cores, effective coverages near 100 percent are common. Nevertheless, effective

coverages as low as 25 percent have been reported [32] with just eight cores. Note that this value is surprisingly close to the minimum possible effective coverage for eight cores, 12.5 percent, and matches the optimistic analytical model of Oh et al. [45] previously discussed. As the core count increases, a decreasing trend can be observed, with effective coverages down to 3 percent for 64 cores. Oh [31] provides empirical data on data sharing in the PARSEC benchmark suite. This data indicates that, for 64 cores, the highest effective coverage in these benchmarks is below 16 percent. Note that some empirical effective coverages are even smaller than the values predicted by Oh's optimistic model.

- *Conservative estimated coverages.* Rather than optimistically evaluating ICCI, we assume conservatively high effective coverages to compare ICCI against other directory schemes, in order to prevent an overestimation of its low-overhead features. We consider reasonable that typical effective coverages for parallel applications running on hundreds of cores can go up to 25 percent (even with hundreds of potential sharers). This is a conservative value taking into account that the empirical results previously discussed suggest smaller coverages and the fact that effective coverages for such core counts can potentially be well under 1 percent. In addition, we must never lose sight of multiprogramming and virtualization, which may potentially raise effective coverages up to 100 percent, regardless of the core count, by running independent applications in all cores (Section 5.2).

Separate directories, whose sizes are fixed at design time, cannot take advantage of small effective coverages. They would just result in many unused directory entries. An option would be to reduce the physical coverage of these directories at design time, counting on the prevalence of small effective coverages. However, this would be a very risky practice, as applications with memory footprints dominated by private data would raise the effective coverage over the directory physical coverage, incurring huge amounts of directory-induced invalidations, and performance would drop dramatically. Even in applications with low effective coverages, specific phases of execution may raise the effective coverage temporarily and ruin performance. On the other hand, ICCI suffers none of these problems. Should the effective coverage go up, even to its highest (100 percent), ICCI would seamlessly allocate as many directory entries as necessary. Should the effective coverage be very low, ICCI would just take up the minimum required amount of directory entries and let almost all the storage resources be used for storing data.

5.2 Boosting the Scalability of ICCI

ICCI's overhead will typically be small for parallel applications with low effective coverages. However, scenarios such as multiprogrammed or virtualized workloads, in which independent applications run in different cores with potentially no data sharing at all, can make effective coverages be close to 100 percent. These represent ICCI's worst-case scenario.

We can use complementary techniques to improve the situation on these worst-case scenarios. Mechanisms that detect different kinds of memory blocks (e.g., private or shared) at runtime have been proposed with a number of purposes [36], [46], [47]. In particular, it is interesting to consider a mechanism that aims at reducing directory-induced invalidations on sparse directories recently proposed by Cuesta et al. [36]. These invalidations are not an issue in ICCI, but the same mechanism is very suitable for increasing ICCI efficiency in a completely different way.

First, we explain what the basics of this mechanism are. At a memory-page granularity, blocks are initially considered to be private to the core that first accesses them. The identifier of the accessing core and the private status of the page are stored in the page table of the process. No information for the blocks of private pages is stored in the directory. When another core attempts an access to a block belonging to a private page, it first receives the page table entry for the memory page (to carry out the virtual-to-physical address translation for the block) that also contains the private status of the page. This means that copies of blocks of that page may exist in the private-cache of the previous core, but no directory information exists for them. This triggers a procedure that creates directory entries for the former private blocks, which become shared, and retrieves the memory block from the private cache of the previous core if necessary. The page table entry is modified to indicate the new (shared) status of the page. This proposal has been reported to attain an average effectiveness over 75 percent, which means that it is able to deactivate the use of directory entries for more than 75 percent of private blocks. The remaining private blocks (less than 25 percent) belong to shared pages, and the page-level mechanism is unable to detect them. The effect of this mechanism was referred to as *deactivating coherence* for private blocks.

Originally, this mechanism was used to alleviate the pressure on sparse directories by reducing the number of blocks contending for the entries of the directory. This, in turn, reduced the amount of conflicts and evictions in the directory, preventing directory-induced invalidations and increasing system performance drastically.

Nothing prevents this mechanism from being directly applicable in combination with ICCI. Our observation is that, when coherence is deactivated for private blocks in a system implementing ICCI, no directory entries are allocated for them in the LLC. In contrast to sparse directories, where coherence deactivation just causes entries to be unused in the fixed-size directory, in ICCI this causes LLC entries to be used by data blocks instead of sharing codes. In both cases, the principle is the same: the effective coverage goes down; however, the side-effects are very different. Note that this mechanism will also reduce the effective coverages of parallel applications.

With this mechanism, the worst scenario for ICCI, 100 percent effective coverage, becomes a potentially perfect scenario. For instance, if many different single-threaded applications are running on the CMP, with all their memory pages being private (with a different table page per process), this mechanism should easily deactivate coherence

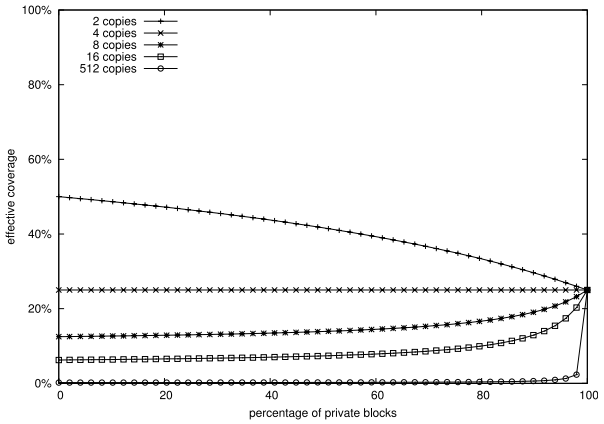


Fig. 9. Effective coverages with 75 percent effectiveness private-block coherence deactivation. Considering memory blocks.

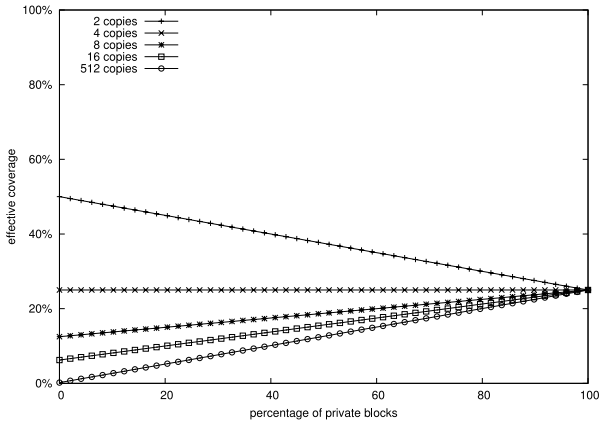


Fig. 10. Effective coverages with 75 percent effectiveness private-block coherence deactivation. Considering private-cache blocks.

for all private-cache blocks, and ICCI's dynamic allocation of directory entries would not allocate any LLC entries for directory information (instead of one entry per private-cache block), introducing no area overhead for coherence in practice (except for shared OS data and code). To our knowledge, no other coherence scheme has such storage adaptability to runtime characteristics.

Figs. 9 and 10 show the effective coverages resulting when applying this mechanism, assuming an effectiveness of 75 percent (smaller than the reported average effectiveness for the mechanism). Note how, for common percentages of private blocks (over 70 percent), the required coverage is always under 40 percent in the worst case (just two copies per shared block), and can be easily under 20 percent as soon as shared memory blocks have more than four copies on average. The typical cases that were highlighted in a rectangle in Fig. 6 always result in effective coverages equal to or below 25 percent after deactivating coherence for private blocks with 75 percent effectiveness. These results lead us to assume 25 percent again as a conservatively high upper bound for effective coverages with large core counts (the same percentage assumed for parallel applications in Section 5.1). We also choose this value because it corresponds to four copies per shared block regardless of the amount of private blocks and the methodology used to count blocks (see the flat line of Figs. 9 and 10) at 25 percent effective coverage, which is a conservative scenario for large core counts. In conclusion, the possibility of deactivating coherence for private data makes the scalability of ICCI benefit from low effective coverages regardless of the private/shared footprint of the particular workload in execution.

As discussed for wide data sharing scenarios, separate directories could be scaled down to physical coverages under 100 percent, in the hope that the coherence-deactivation mechanism will always attain low effective coverages. However, this mechanism does not give any guarantees on the deactivation of coherence for a single block (Figs. 6 and 7 still represent the case in which the mechanism has an effectiveness of 0 percent in addition to the case in which the mechanism is not used). As explained previously, using low physical coverages in the fixed-size directory would be very risky (and very-low ones, matching expected effective coverages under 25 percent, are out of the question). On the other hand, ICCI dynamically benefits from low effective coverages at runtime, potentially requiring 0 percent of storage resources for cache coherence.

The only advantage remaining in favor of using a fixed-size directory is their smaller entry size for low core counts. This may make ICCI take up more storage resources even when benefiting from small effective coverages. However, as the core count rises and sharing code size approaches LLC entry size, fixed-size directories lose this advantage and fail as a scalable scheme, while effective coverages go down benefiting ICCI's scalability.

Once sharing code size surpasses LLC entry size, composable sharing codes (similar to SCD) specific for ICCI can be used, with the possibility to store many sharers and point to other entries that make up the sharing code at the same time thanks to the large LLC entry size. We find this possibility especially appealing. For instance, SCD's two-level hierarchy for 1,024 cores can be built with just three LLC entries in ICCI, instead of the original 33 SCD cache entries. This means a much higher entry-efficiency and faster lookups than in SCD. 512-bit LLC entries can support a hypothetical two-level ICCI-SCD coherent system containing 256K cores, with no dedicated storage overhead for coherence. In addition, 512-bit entries can store up to 28 pointers (to any of the 256K cores), before needing to use multiple entries, ensuring a high entry-efficiency. Note that when several LLC entries are needed for tracking one memory block, many private-cache blocks share each of these entries, resulting in a reduction of the effective coverage. Increasing the cache line size to 128 bytes would potentially enable cache coherence for a 1M-core machine. Also, the number of levels in the hierarchy can be increased to support more cores. In addition, ICCI's large directory (with potential effective coverage equal to the percentage ratio of the number of LLC entries to the number of private-cache entries) prevents conflicts and directory-induced invalidations naturally, removing altogether the need to use ZCaches to simulate large associativity in small coverage directories.

In addition, techniques similar to Amoeba Caches [48] can be used to enable different entry sizes in the LLC, to accommodate (small) directory entries and (large) cache entries. However, as the sharing code size approaches the

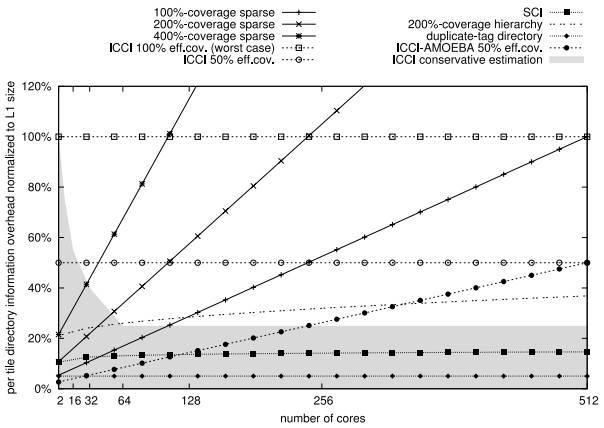


Fig. 11. Cache coherence storage overhead of several schemes depending on core count (from 2 to 512 cores).

LLC entry size, this results in an unnecessary complication. Nevertheless, taking this path one step further may be interesting, as different sharing codes could be used to minimize the total space taken up by coherence information in ICCI (e.g., pointers for few sharers or a bit vector relative to an area for a block shared in that area of the chip).

5.3 ICCI Compared to Other Coherence Schemes

After the discussion on typical coverages, now we can put ICCI's storage overhead into perspective with other coherence schemes. Fig. 11 shows the overhead of several proposals, ranging from 2 to 512 cores. The overhead introduced by these coherence schemes is measured as the percentage of storage space used for coherence information relative to the aggregate private-cache capacity. This figure shows sparse directories of several coverages (solid lines), ICCI for several effective coverages, a hierarchical directory (with two levels of sharing information and a 200 percent coverage in each level), SCI and a duplicate-tag directory. We assume an SCI version adapted to CMPs, in which a 200 percent coverage directory cache storing pointers is NUCA-distributed and each L1 cache entry contains two pointers to create the double-linked list characteristic of SCI. In this figure, ICCI uses the same encoding for sharers as the sparse directory (full-map bit-vector).

The characteristics of ICCI are a bit unusual. Its overhead depends on effective coverage rather than on core count. It remains constant for a given effective coverage because the number and size of the LLC entries used as directory entries is the same regardless of the number of cores. In addition, the entry size of the LLC is the same as the size of a private-cache entry, making the overhead of ICCI on private-cache capacity approximately equal to the value of the effective coverage. For instance, the worst-case effective coverage of 100 percent, with one LLC entry tracking each private-cache block, results in (approximately) 100 percent storage overhead on L1 cache capacity regardless of the number of cores. The only difference introduced by changing the core count is that, as the core count and the number of banks in a NUCA cache increase, more address bits are used to select the home LLC bank, and the number of remaining bits used in the tags of the LLC goes down. This is such a subtle difference that its impact on the overhead of ICCI cannot be appreciated in the graph.

Note that, contrary to traditional directories, ICCI will typically take up less storage per tile for directory information as core count increases. Higher core counts can potentially experience higher data sharing, generating smaller effective coverages and making the area overhead of ICCI go down as the core count goes up.

The shadowed area of Fig. 11 represents the expected range of overheads for ICCI, assuming the conservatively high upper bound for effective coverages for large core counts (25 percent) that was calculated in Sections 5.1 and 5.2. Also, an estimated overhead for ICCI in combination with Amoeba Caches is shown, for a 50 percent effective coverage, which beats the sparse directory even with small core counts. However, the higher the core count, the less savings this alternative provides and the less sense it makes to use it.

Several facts stand out in this graph:

- A 200 percent coverage sparse directory uses twice as many resources as ICCI in its worst-case scenario (i.e., 100 percent effective coverage) for 512 cores. ICCI's area overhead for typical effective coverages range between 12.5 and 0 percent that of the sparse directory. The sparse directory uses more resources than ICCI with 25 percent effective coverage as soon as the core count rises over 50 cores, and its overhead keeps rising with core count while in ICCI it goes down as more cores increase data sharing.
- The 100 percent coverage sparse directory uses as many resources as ICCI's worst-case with 512 cores and four times as many as ICCI with 25 percent effective coverage. In addition, a sparse directory with just 100 percent coverage would produce unacceptably large numbers of directory-induced invalidations due to conflicts.
- ICCI provides reasonable overhead for up to 512 cores based on expected effective coverages, without dramatic changes and without the complexity of more intricate alternatives such as hierarchical protocols. Note that with just 64 cores and a 25 percent effective coverage, ICCI already takes fewer resources than the hierarchical directory.
- ICCI has lower overhead than SCI when the effective coverage is below 16 percent and lower than duplicate-tag directories when the effective coverage is below 5 percent. Some empirical effective coverages observed are under these values, even for low core counts (see Section 5.1).

Fig. 12 shows the storage overhead on the L1 caches for core counts between 512 and 256 K, in semi-logarithmic scale. In this case, ICCI uses the same composable codification as SCD. When ICCI uses composable sharing codes (taking up several entries when many sharers exist, like in SCD), the relation between effective coverage and overhead may vary ever so slightly (because the one-to-one relationship between directory entries and memory blocks is broken), although they remain roughly equivalent. Nevertheless, this has no effect on the fixed coverages shown in the figures (in which each LLC entry used for directory

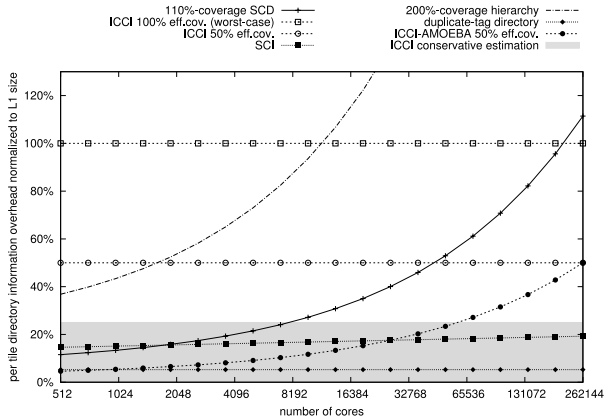


Fig. 12. Cache coherence storage overhead of several schemes depending on core count (from 512 to 256K cores).

tracks one or two private-cache blocks) and on the conservative upper bound for effective coverages. In this case, sparse directories are not shown, as their overhead is well out the charts (e.g., over 50,000 percent for 256K cores in a 200 percent coverage sparse directory).

Interesting results are the following:

- SCD goes over 100 percent overhead eventually, while ICCI with SCD's sharing code remains under 25 percent. In addition, ICCI does not require the use of ZCaches, as explained in Section 5.2.
- Again, due to the particular characteristics of ICCI, it has lower overhead than SCI when the effective coverage is below 16 percent and lower overhead than duplicate-tag directories when the effective coverage is below 5 percent.

As the size of the sharing code approaches the LLC entry size with increasing core counts, ICCI has obviously superior scalability properties than a separate directory using the same sharing code (e.g., full-map in a sparse directory for up to 512 cores and a composable hierarchical code like SCD's for up to 256K cores).

5.4 Discussion on ICCI's Restrictions

Among the main drawbacks of ICCI, ICCI results inconvenient when different treatment is required for the structures storing data and directory information. An example of this is if we want to apply compression techniques to the LLC but not to the directory, or conversely. On the other hand, the usage of a single structure may simplify the application of such techniques globally to data and directory and avoid hardware replication, although the different nature of directory entries and memory blocks may reduce the effectiveness of techniques such as compression.

The fact that the entry size of the directory is constrained to that of an LLC data line can also be seen as a restriction, especially for chips containing only a few cores, because in that case smaller entries would be preferable. However, this problem disappears in practice as we move to large core counts, which precisely is the scenario whose problems ICCI aims at solving. ICCI also requires the use of either inexact of composable sharing codes for very large numbers of cores, whose management is more complicated, but this is also the case for any other directory-based coherence scheme.

Finally, when working in combination with non-inclusive or inclusive cache hierarchies, ICCI would require a slightly more complex cache able to deal with two simultaneous tag hits (as both a memory block and its associated directory entry could be stored in the LLC, in different ways of the same set) and to serialize the processing of both read blocks in the particular order desired. Nevertheless, such LCC cache is easily realizable and its effective capacity with ICCI, for the same global amount of resources than in other directory schemes, would be higher thanks to ICCI's dynamic coverage that prevents the existence of unused directory entries, as discussed in Section 5. When an exclusive hierarchy is chosen instead, like in the example implementation of ICCI evaluated in Section 4, such hierarchy may cause many three-hop cache misses that in some scenarios may degrade performance (this happens with ICCI or any other directory scheme), but our results suggest that this can still be a good design.

6 Related Work

Novel coherence schemes appear periodically in the literature, and the complexity of the most recent ones shows that it is becoming increasingly difficult to improve the scalability of cache coherence.

For instance, the Tagless Coherence Directory (TL) [49] uses multiple-hash bloom filters to store directory information. In essence, TL works as an inexact duplicate-tag directory (inexactitude due to bloom filter aliasing, which creates spurious invalidation messages). Ideally, TL has constant per-core overhead. In practice, the bloom filter size has to be increased with the core count to prevent excessive levels of aliasing, in a trade-off between extra network traffic and area overhead. In addition, although more energy-efficient than a duplicate-tag directory, TL is less energy-efficient than ICCI. In ICCI, an LLC lookup is enough to find the block or the sharing vector. TL requires an additional directory lookup whose energy consumption is proportional to the number of cores. In our 512-core CMP, a TL access requires looking up 512 multiple-hash bloom filters in parallel to generate the 512-bit sharing vector. Overall, TL introduces the difficulty of managing bloom filters in hardware, extra resources for the directory, and the inefficiency of spurious invalidation messages compared to ICCI.

SPACE [50] is based on the observation that many cache blocks have the same or similar sharing patterns. SPACE stores these sharing patterns in a table. The directory cache stores pointers to positions of the pattern table, one pointer per tracked block, with many directory entries pointing to the same patterns. As long as the pointer directory cache dominates the overhead, SPACE can scale up gracefully with core count. However, as the number of cores grows, the pattern table starts to dominate (note that the table is distributed and patterns need be repeated at every tile) resulting in a per-core overhead proportional to the number of cores. SPACE assumes that few different sharing patterns exist at any given time, hence a small sharing pattern table is needed, resulting in a smaller overhead than an ordinary sparse directory in any case. However, as the number of

cores grows, the number of possible sharing patterns increases exponentially; hence, the possibilities of pattern repetition diminish. SPACE also introduces the complexity of managing the sharing pattern table, which requires nontrivial actions such as pattern coalescing. ICCI suffers none of these problems.

SPATL [51] combines both the Tagless Coherence Directory and SPACE, storing the pointers to the sharing pattern table inexactly in bloom filters, reducing the overhead further at the cost of the aggregate complexity of both proposals. Unfortunately, for large core counts, SPATL faces the same scalability problems as SPACE due to the size of the pattern table.

The Cuckoo Directory [3] uses a different hash function per directory way so as to prevent directory conflicts. This reduces the need for overprovisioning cache directories, but does not change their per-core overhead growth.

The SGI UV2 [2] uses directory-based cache coherence to maintain 512-processor-socket coherent domains. The full directory is stored in DRAM, typically consuming approximately 3 percent of the 64 TB memory space, and an on-chip directory cache allows for fast access to information about reused addresses. Similarly, WayPoint [52] uses small, low-associativity directory caches. Evicted directory entries are inserted in main memory to prevent costly directory-induced invalidations. ICCI allows for 512-processor coherence domains without the need for the slow-access DRAM directory nor the directory cache used by the SGI UV2 and WayPoint.

The idea of using cache entries to store information other than memory blocks has already been used by other proposals like DeNovo [53], in which cache coherence is simplified by assuming some properties in the software, and cache entries store the identity of the owner L1 caches. Nevertheless, ICCI's idea of providing scalable ordinary hardware cache coherence by leveraging the use of the minimum amount of LLC entries to store regular sharing information is different to any previous approaches.

We have already discussed the Scalable Coherence Directory [20] and used it to compare ICCI effectiveness.

7 Conclusion

In this paper, we have introduced ICCI, a new cache organization that leverages shared cache resources and flat coherence protocols to provide inexpensive hardware cache coherence for large core counts (e.g., 512), without degrading the performance and energy consumption of the system as other proposals do (e.g., coarse bit vectors, SCI) and without the need of complex cache structures (like SCD's ZCaches). Simple changes in the system are needed to implement ICCI with respect to a traditional full-map directory. ICCI does not introduce any dedicated storage overhead, yet it provides large storage space for coherence information. ICCI takes up entries of the LLC as directory entries. ICCI incurs a negligible number of directory-induced invalidations and outperforms complex state-of-the-art proposals such as SCD, especially in terms of energy. Moreover, ICCI can be used in combination with more elaborated sharing codes to apply it to extremely large core counts. By combining ICCI and SCD, we can provide hardware cache coherence for massively parallel machines at no extra chip area cost. This can be done by storing SCD's hierarchical entries in the LLC and using ICCI's operation. ICCI removes the need to orchestrate two different array structures for data and directory information and the use of complex coherence protocols (e.g., hierarchical, list based or tree based).

We have carried out an analytical survey of the characteristics of workloads, concluding that low effective coverages typical at runtime ensure high scalability for ICCI in terms of storage taken up for directory information in the LLC. In the presence of data sharing, effective coverages typically below 25 percent make ICCI take up few directory entries and add little overhead. In the absence of data sharing, deactivating coherence for private blocks also enables low effective coverages (typically under 25 percent), making ICCI take up few directory entries under any circumstances. In comparison, a fixed-size directory always takes up the same amount of entries, determined at design time to fit worst-case scenarios, and leaves them unused if there is wide data sharing or if coherence for private blocks is deactivated.

ICCI's logical directory size (number of sets and associativity) is huge compared to dedicated storage directories. Directory-induced invalidations are never an issue in ICCI. This prevents the need for complex caches (e.g., ZCaches) or hash procedures (e.g., Cuckoo hashing) to emulate large associativity in small fixed-coverage separate directories to prevent directory-induced invalidations.

Finally, contrary to any other directory scheme, ICCI's usage of resources for directory information typically decreases as the number of cores rises, because more opportunities for data sharing appear, reducing the effective coverage and the number of allocated directory entries. In addition, reported effective coverages make ICCI take up less area for directory information than SCI or duplicate-tag directories with as few as 64 cores.

Acknowledgments

The authors would like to thank the anonymous reviewers for their thorough work in revising this paper. Their insights enabled them to unveil some of the features of ICCI whose description and analysis make up an important part of the paper. This work was supported by the Spanish MEC and European Commission FEDER funds under grants "TIN2009-14475-C04-02" and "TIN2012-31345". A. García-Guirado was supported by a research grant from the Spanish MEC under the FPU National Plan (AP2008-04387).

References

[1] M.M.K. Martin et al., "Why On-Chip Cache Coherence Is Here to Stay," *Comm. ACM*, vol. 55, no. 7, pp. 78-89, July 2012.

[2] G. Thorson et al., "SGI UV2: A Fused Computation and Data Analysis Machine," *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1-9, 2012.

[3] M. Ferdman et al., "Cuckoo Directory: A Scalable Directory for Many-Core Systems," *Proc. 17th IEEE Int'l Symp. High Performance Computer Architecture (HPCA)*, pp. 169-180, 2011.

[4] L. Censier et al., "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Computers*, vol. C-27, no. 12, pp. 1112-1118, Dec. 1978.

[5] J.L. Baer et al., "On the Inclusion Properties for Multi-Level Cache Hierarchies," *Proc. 15th Ann. Int'l Symp. Computer Architecture (ISCA)*, pp. 73-80, 1988.
[6] M. Chaudhuri et al., "Introducing Hierarchy-Awareness in Replacement and Bypass Algorithms for Last-Level Caches," *Proc. 21st Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, pp. 293-304, 2012.
[7] A. Gupta et al., "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," *Proc. Int'l Conf. Parallel Processing (ICPP)*, pp. 312-321, 1990.
[8] P. Conway et al., "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *IEEE Micro*, vol. 30, no. 2, pp. 16-29, Mar./Apr. 2010.
[9] A. Agarwal et al., "An Evaluation of Directory Schemes for Cache Coherence," *Proc. 15th Ann. Int'l Symp. Computer Architecture (ISCA)*, pp. 280-298, 1988.
[10] J. Laudon et al., "The SGI Origin: A ccNUMA Highly Scalable Server," *Proc. 24th Ann. Int'l Symp. Computer Architecture (ISCA)*, pp. 241-251, 1997.
[11] M.E. Acacio et al., "A Two-Level Directory Architecture for Highly Scalable cc-NUMA Multiprocessors," *IEEE Trans. Parallel and Distributed Systems (TPDS)*, vol. 16, pp. 67-79, Nov. 2005.
[12] D.V. James et al., "Distributed-Directory Scheme: Scalable Coherent Interface," *Computer*, vol. 23, no. 6, pp. 74-77, June 1990.
[13] L.A. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proc. 27th Ann. Int'l Symp. Computer Architecture (ISCA)*, pp. 282-293, 2000.
[14] J. Held et al., "Introducing the Single-Chip Cloud Computer," Intel White Paper, 2010.
[15] T. Piazza et al., "Technology Insight: Intel Next Generation Microarchitecture Code Name Haswell," *Proc Intel Developer Forum*, Sept. 2012.
[16] T. Fischer et al., "Design Solutions for the Bulldozer 32nm SOI 2-Core Processor Module in an 8-Core CPU," *IEEE Int'l Solid-State Circuits Conf. Digest of Technical Papers (ISSCC)*, pp. 78-80, 2011.
[17] C. Ramey, "TILE-Gx100 ManyCore Processor: Acceleration Interfaces and Architecture," *Proc. Hot Chips*, Aug. 2011.
[18] C. Kim et al., "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 211-222, 2002.
[19] D. Molka et al., "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," *Proc. 18th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, pp. 261-270, 2009.
[20] D. Sanchez et al., "SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding," *Proc. 18th IEEE Int'l Symp. High-Performance Computer Architecture (HPCA)*, pp. 129-140, 2012.
[21] D. Sanchez et al., "The Zcache: Decoupling Ways and Associativity," *Proc. 43rd Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, pp. 187-198, 2010.
[22] A. Jaleel et al., "Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies," *Proc. 43rd Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, pp. 151-162, 2010.
[23] C.-K. Luk et al., "Pin: Building Customized Program Analysis tools with Dynamic Instrumentation," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pp. 190-200, 2005.
[24] M.M.K. Martin et al., "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset," *SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92-99, 2005.
[25] M. Monchiero et al., "How to Simulate 1000 Cores," *SIGARCH Computer Architecture News*, vol. 37, no. 2, pp. 10-19, July 2009.
[26] K. Sohn et al., "A 1.2V 30nm 3.2Gb/s/pin 4Gb DDR4 SDRAM with Dual-Error Detection and PVT-Tolerant Data-Fetch Scheme," *IEEE Int'l Solid-State Circuits Conf. (ISSCC)*, pp. 38-40, 2012.
[27] T. Legler, "Choosing the DRAM with Complex System Considerations," *Proc. Embedded Systems Conf.*, Mar. 2012.
[28] S. Li et al., "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," *Proc. 42nd Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, pp. 469-480, 2009.
[29] S. Ma et al., "Supporting Efficient Collective Communication in NoCs," *Proc. IEEE 18th Int'l Symp. High-Performance Computer Architecture (HPCA)*, pp. 1-12, 2012.
[30] A. Hartstein et al., "Cache Miss Behavior: Is It $\sqrt{2}$," *Proc. Third Conf. Computing Frontiers (CF)*, pp. 313-320, 2006.
[31] T.C. Oh, "Analytical Models for Chip Multiprocessor Memory Hierarchy Design and Management," PhD dissertation, Univ. of Pittsburgh, 2010.
[32] N. Agarwal et al., "In-Network Coherence Filtering: Snoopy Coherence without Broadcasts," *Proc. 42nd Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, pp. 232-243, 2009.
[33] Z. Guz et al., "Nahalal: Memory Organization for Chip Multiprocessors," technical report, Dept. of Electrical Eng., Technion-IIT, 2006.
[34] B.M. Beckmann et al., "ASR: Adaptive Selective Replication for CMP Caches," *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, pp. 443-454, 2006.
[35] N.D. Enright Jerger et al., "Virtual Tree Coherence: Leveraging Regions and In-Network Multicast Trees for Scalable Cache Coherence," *Proc. 41st IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, pp. 35-46, 2008.
[36] B. Cuesta et al., "Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks," *Proc. 38th Ann. Int'l Symp. Computer Architecture (ISCA)*, pp. 93-104, 2011.
[37] S.H. Pugsley et al., "SWEL: Hardware Cache Coherence Protocols to Map Shared Data onto Shared Caches," *Proc. 19th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, pp. 465-476, 2010.
[38] Z. Guz et al., "Utilizing Shared Data in Chip Multiprocessors with the Nahalal Architecture," *Proc. 20th Ann. Symp. Parallelism in Algorithms and Architectures (SPAA)*, pp. 1-10, 2008.
[39] B.M. Beckmann et al., "Managing Wire Delay in Large Chip-Multiprocessor Caches," *Proc. 37th Int'l Symp. Microarchitecture (MICRO)*, pp. 319-330, 2004.
[40] B.M. Rogers et al., "Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling," *Proc. 36th Ann. Int'l Symp. Computer Architecture (ISCA)*, pp. 371-382, 2009.
[41] M.M. Martin et al., "Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors," *Proc. 30th Ann. Int'l Symp. Computer Architecture (ISCA)*, pp. 206-217.
[42] G. Kurian et al., "ATAC: A 1000-Core Cache-Coherent Processor with On-Chip Optical Network," *Proc. 19th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, pp. 477-488, 2010.
[43] A. Krishna et al., "Data Sharing in Multi-Threaded Applications and Its Impact on Chip Design," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS)*, pp. 125-134, 2012.
[44] E.Z. Zhang et al., "Does Cache Sharing on Modern CMP Matter to the Performance of Contemporary Multithreaded Programs," *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 203-212, 2010.
[45] T. Oh et al., "An Analytical Model to Study Optimal Area Breakdown between Cores and Caches in a Chip Multiprocessor," *Proc. IEEE CS Ann. Symp. VLSI (ISVLSI)*, pp. 181-186, 2009.
[46] Y. Li et al., "Practically Private: Enabling High Performance CMPs through Compiler-Assisted Data Classification," *Proc. 21st Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, pp. 231-240, 2012.
[47] N. Hardavellas et al., "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," *Proc. 36th Ann. Int'l Symp. Computer Architecture (ISCA)*, pp. 184-195, 2009.
[48] S. Kumar et al., "Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy," *Proc. 45th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, pp. 376-388, 2012.
[49] J. Zebchuk et al., "A Tagless Coherence Directory," *Proc. 42nd Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, pp. 423-434, 2009.
[50] H. Zhao et al., "SPACE: Sharing Pattern-Based Directory Coherence for Multicore Scalability," *Proc. 19th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, pp. 135-146, 2010.
[51] H. Zhao et al., "SPATL: Honey, I Shrunk the Coherence Directory," *Proc. 2011 Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, pp. 33-44, 2011.
[52] J.H. Kelm et al., "WAYPOINT: Scaling Coherence to Thousand-Core Architectures," *Proc. 19th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, pp. 99-110, 2010.
[53] B. Choi et al., "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, pp. 155-166, 2011.

Antonio García-Guirado received the BSc and MSc degrees (with honors) in computer science and engineering from the University of Murcia, Spain, in 2008 and 2009, respectively, where he is currently working toward the PhD degree in computer science. He works as a research scientist at Intel Labs, Universitat Politècnica de Catalunya, Barcelona, Spain. His research interests include cache coherence and memory consistency, cache organizations, virtualization, and interconnection networks.

Ricardo Fernández-Pascual received the MS and PhD degrees in computer science from the Universidad de Murcia, Spain, in 2004 and 2009, respectively. In 2004, he joined the Computer Engineering Department as a PhD student with a fellowship from the regional government. In 2006, he joined the Computer Engineering Department of the Universidad de Murcia where he is currently an associate professor. His research interests include general computer architecture, fault tolerance, memory hierarchies for chip multiprocessors, and performance simulation.

José M. García received the MS degree in electrical engineering and the PhD degree in computer engineering, both from the Technical University of Valencia. He is currently a professor of computer architecture at the Department of Computer Engineering in the University of Murcia, Spain, and also the head of the Research Group on Parallel Computer Architecture. He specializes in computer architecture, parallel processing, and interconnection networks. His current research interests include the design of power-efficient heterogeneous systems, and the development of data-intensive applications for those systems. He has published more than 140 refereed papers in different journals and conferences in these fields.

Queries to the Author

Q1. Index terms were given in the LaTeX source file but missing in the input pdf. Source file has been followed. Please check.
Q2. Please check whether the affiliations of the authors are okay as set.