Int J Parallel Prog (2015) 43:1054–1077
DOI 10.1007/s10766-014-0339-0

Cluster Cache Monitor: Leveraging the Proximity Data in CMP

Guohong Li · Olivier Temam · Zhenyu Liu · Sanchuan Guo · Dongsheng Wang

Received: 5 January 2014 / Accepted: 13 October 2014 / Published online: 4 November 2014

Abstract As the number of cores and the working sets of parallel workloads increase, shared L2 caches exhibit fewer misses than private L2 caches by making a better use of the total available cache capacity, but they also induce higher overall L1 miss latencies because of the longer average distance between two nodes, and the potential congestions at certain nodes. One of the main causes of the long L1 miss latencies are accesses to home nodes of the directory. However, we have observed that there is a high probability that the target data of an L1 miss resides in the L1 cache of a neighbor node. In such cases, these long-distance accesses to the home nodes can be potentially avoided. We organize the multi-core into clusters of 2×2 nodes, and in order to leverage the aforementioned property, we introduce the Cluster Cache Monitor (CCM). The CCM is a hardware structure in charge of detecting whether an L1 miss can be served by one of the cluster L1 caches, and two cluster-related states are added in the coherence protocol in order to avoid long-distance accesses to home nodes upon hits in the cluster L1 caches. We evaluate this approach on a 64-node multi-core using SPLASH-2 and PARSEC benchmarks, and we find that the CCM can reduce the execution time by 15 % and reduce the energy by 14 %, while saving

This work is supported by Nature Science Foundation of China under Grant No. 61373025, the National 863 High-Tech Program of China under Grant No. 2012AA010905.

G. Li · Z. Liu · D. Wang (✉)
Tsinghua National Laboratory for Information Science and Technology, Beijing, China
e-mail: wds@tsinghua.edu.cn

S. Guo
National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing, China

O. Temam
INRIA Saclay, Orsay, France

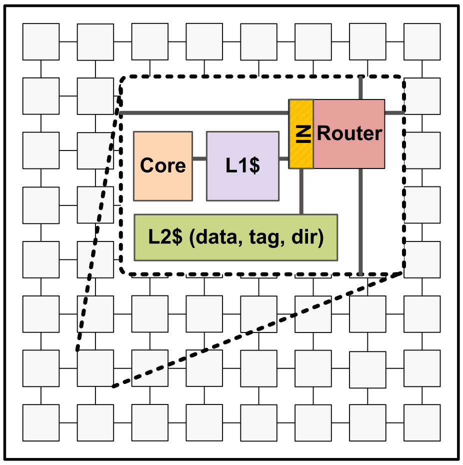

Fig. 1 Baseline architecture

28 % of the directory storage area compared to a standard multi-core with a shared L2. We also show that the CCM outperforms recent mechanisms, such as ASR, DCC and RNUCA.

1 Introduction and Motivation

Each node of a multi-core usually contains a core, a private L1 cache, and L2 storage [4,7,28], see Fig. 1. This L2 storage can take two forms: a private L2 cache, or part of a distributed L2 cache shared by all nodes. For a large number of cores and large workloads, a shared L2 outperforms private L2s, where too much cache capacity is used for storing redundant data. We illustrate that point by comparing, in Fig. 2, the performance of two 64-core architectures with respectively a distributed shared L2 and private L2s of the same total storage, for different benchmark suites; we show the ratio of execution time (private over shared) and the memory footprint of the benchmarks.[1] Beyond performance, there is also an increasingly stringent yield issue: because foundries have difficulties producing chips containing a too high amount of SRAM cells with a low enough total number of faults [2,27,30], the total SRAM capacity available on-chip may not increase as fast as the number of cores. In that context, it will become more efficient to share among cores all the available SRAM capacity rather than to use small private L2 caches. It is interesting to note, for instance, that the SPARC T3 architecture is composed of 16 cores and 6MB L2 storage distributed among private caches [28], while the recent Intel® Xeon Phi™ Coprocessor 5110P has 60 cores and 30 MB distributed L2 cache.

However, as the number of cores increases, the increased latency of L1 misses cancancel out the capacity advantage of the shared L2 [32]. With standard private L2 caches, an L1 miss is serviced by the local L2. With a shared L2, the L1 miss can be

[1] The methodology for these experiments is described in Sect. 4.

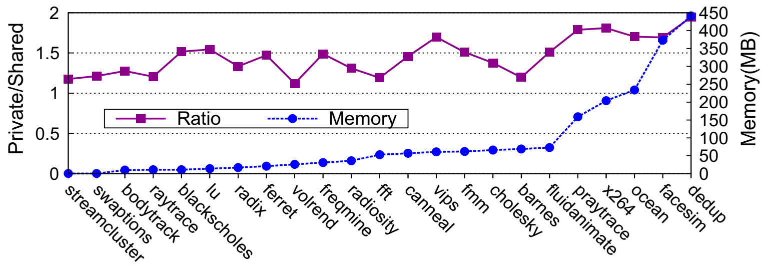

Fig. 2 Shared versus private L2s

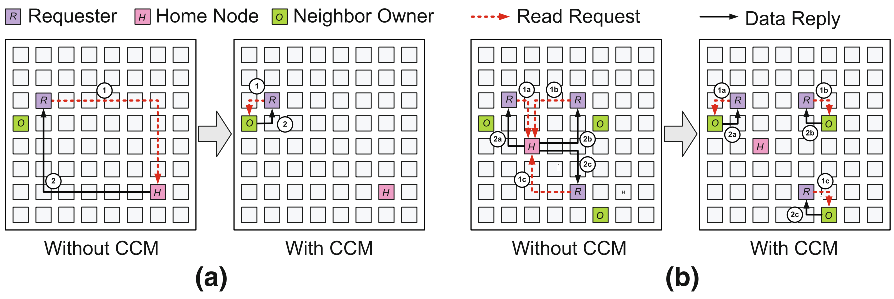

Fig. 3 **a** *Left* long distance access to the home node. **b** *Right* congestion at the home node

serviced by any of the L2 cache banks spread among all nodes; thus, as the number of nodes increases, the average L1 miss latency increases.

The root cause of long latency is usually access to the memory block home node. Multi-cores with a large number of cores favor distributed directory coherence, where each memory block has a home node. Upon an L1 miss, this home node must be accessed by the requesting node, which updates the coherence state of the memory block, and directs the node currently owning a copy of the data to send it back to the requesting node. As the number of nodes increases, the average latency to access the home nodes can increase for two reasons: (1) the increased average distance between the requesting node and the home node, see Fig. 3a, and (2) the congestion at the home node, see Fig. 3b. This long access latency to the home node is all the more wasteful, both time-wise and energy-wise, if the requested data itself is located in a nearby node, as illustrated in Fig. 3, see `Owner` node.

We observe that this case is actually frequent: upon an L1 miss in one node, there is a high probability that the requested data is located in the L1 cache of a nearby node, a form of node-level spatial locality. In Fig. 4, we show the fraction of all L1 misses which can be serviced by nodes from the same cluster (the benchmarks and experimental setup are later described in Sect. 4). On average, 59 % of L1 misses can be serviced by neighbor cluster nodes, and up to 89 % for `radiosity`. This behavior is inherent to the pattern of access to shared data among threads of parallel programs. Consider for instance the parallel FFT: threads first process row-wise transforms, then followed by column-wise transforms, spreading input data among cores; moreover, the same FFT coefficients are used by all nodes, so that if some of these coefficients

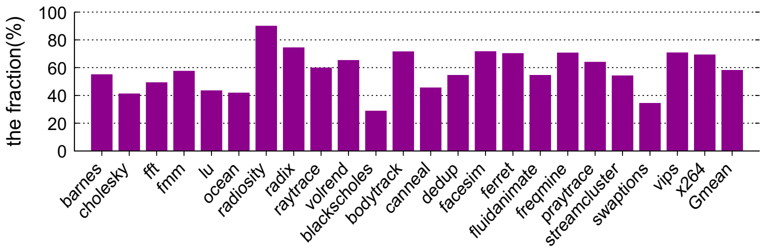

Fig. 4 Fraction of L1 misses which can be serviced by nodes from the same cluster

are evicted from the L1 of one core, it is very likely that the L1 of a neighbor core will contain a copy. Similar cases are found in many applications.

In this article, we leverage this property by grouping nodes into 2 × 2 clusters, and by introducing a novel hardware structure, the Cluster Cache Monitor (CCM), which can detect whether an L1 miss can be serviced by one of the caches of the cluster. By coupling the CCM with the introduction of cluster-related states in the coherence protocol, we show that it is possible to avoid long-distance L2 accesses in a significant number of cases, thereby improving performance and saving energy. Moreover, in spite of the addition of the CCM, the overall directory storage cost is significantly reduced, because node-based sharer information is replaced with cluster-based sharer information in the directory. We run SPLASH-2 and PARSEC benchmarks on a 64-core architecture, and we observe an average execution time reduction of 15 %, an average energy reduction of 14 %, while saving 28 % of the area of the directory.

This paper makes the following contributions:

- We propose an approach to reduce the long latency drawback of shared L2 caches, and thus take advantage of their lower miss rate advantage over private L2s.
- For that purpose, we introduce the CCM, a mechanism for taking advantage of the property that many L1 misses can be serviced by neighbor nodes, thus reducing the L1 miss latency.
- Thanks to a coherence protocol modification, the CCM can also remove all long-distance accesses when the L1 miss can be serviced by a neighbor node, reducing network hotspots, and further contributing to reducing the overall L1 miss latency.
- We compare CCM against three state-of-the-art private or shared L2 mechanisms (ASR, DCC, RNUCA), and we show that, on average, CCM outperforms all three mechanisms.

The remainder of the paper is organized as follows. We compare the CCM to the related work in Sect. 2, we present the CCM in Sect. 3, the methodology in Sect. 4, and the experimental evaluation in Sect. 5.

2 Related Work

The research works most closely related to CCM are victim or replication strategies [6,12,16,20,26,31,32], Cooperative Caching strategies [1,5,10,11,13,17,18], and to a lesser extent, hierarchical directory coherence [8,14,15,22,33,34].

2.1 Replication

Replication and migration strategies [6,12,16,20,26,31,32] share the goal of CCM to reduce accesses to distant L2 caches, even though the approach is entirely different. The common intuition of these mechanisms is the replication or the migration of L2 cache blocks, at the cost of a modified directory structure and coherence protocol, in order to keep track of the copied or moved cache blocks. For example, Victim Replication [32] moves data evicted from the L1 to the private L2 cache, a scheme later improved with Victim Migration [31]. Beckmann et al. [6] propose Adaptive Selective Replication (ASR) which dynamically monitors workload behavior in order to better adjust replication. Like CCM, replication or migration schemes trade long-distance accesses for more local accesses, and both approaches require the addition of a few coherence protocol states. However, unlike CCM, which only checks neighbor L1s, these schemes require to access the local L2, which is more costly, both time-wise and energy-wise. Another weakness of such schemes is that they sacrifice available L2 capacity due to migration or replication. RNUCA [16] improves over replication schemes by forming overlapping clusters of nodes which enable to virtually replicate data among several cores at no extra capacity. It also eliminates the need for hardware coherence at the L2 cache, though hardware coherence at the L1 cache is still required. Even though RNUCA is based on shared L2s, each node can access addresses within a cluster of neighbor private L2s. Unlike CCM, RNUCA still requires to go through the L2 for any L1-to-L1 transfer. In Sect. 5, we empirically compare CCM against both ASR and RNUCA which are among the most advanced replication/migration schemes.

2.2 Migration

Chang et al. [11] proposed Cooperative Caching which consists in managing private caches in such a way that they achieve almost the same capacity benefits as a shared L2 of the same size. In addition to aforementioned replication strategies, Cooperative Caching relies on original mechanisms such as storing evicted L2 data into neighbor L2 caches. This notion of taking advantage of neighbor data bears some resemblance with the concept of CCM. However, in the case of Cooperative Caching, the data is actively placed in neighbor nodes, while CCM takes advantage of an observed property of parallel programs. Moreover, Cooperative Caching has two limitations: it does not avoid long distance accesses to the home nodes for coherence management purposes, and its scalability is limited by the presence of a Central Coherence Engine. However, Herrero et al. [17] addressed the latter issue in Distributed Cooperative Caching (DCC) by using a Distributed Coherence Engine.

2.3 Leveraging Data Proximity

Proximity-Aware directory coherence (PADC) [10] shares many intuitions with DCC: both are based on private L2s and take advantage of neighbor data again. The main difference is that DCC is more aggressive by moving data to neighbor nodes, while

PADC takes advantage of existing proximity. As a result, we compare CCM against DCC in Sect. 5.

Eisley et al. [13] also takes advantage of data proximity by moving much of the coherence-related control and data storage into the network, in order to reduce the miss latency for parallel workloads. However, the tree cache introduced in routers and virtual links constructed from home node to requestors result in costly cache line duplication, which reduces the overall storage capacity at a constant hardware cost. The design of routers is also significantly more complex because of their involvement in coherence decisions. CCM requires only marginal modifications of the network interface. Barrow [5] leverages the proximity data by sending requesting messages to all neighbors, which complicates the coherence protocol. Acacio [1] and Hossain [18] also introduces the concept of using the nearby data.

2.4 Hierarchical Coherence

Like hierarchical coherence mechanisms, CCM reduces the sparse directory storage requirement by replacing the node location with a cluster location; since the cluster is 2×2, i.e., 4 nodes, the directory storage size is divided by 4. While this storage reduction is similar to what hierarchical coherence protocols achieve, this is only a side-effect of CCM, the main goal and benefit is the reduction of the L1 miss latency and NoC traffic, which most hierarchical coherence protocols do not aim at nor achieve. There are numerous examples of hierarchical coherence organizations [34]. For instance, Fractal Coherence [33] is a hierarchical scheme which aims at facilitating the validation of coherence protocols using self-similarity. Manager-Client Pairing [8] also proposes a framework for implementing coherence hierarchies, but it focuses on the easier integration of layers, and design-space exploration; moreover, MCP voluntarily uses a simplified network model, where virtual channel allocation, switch allocation and link traversal are contention free. Moreover, many of the existing directory size reduction approaches, such as Cuckoo [14], are orthogonal, and thus complementary, to CCM.

3 Cluster Cache Architecture

We first provide an overview of the CCM operations, then detail its hardware structure, and finally, present the modified cache coherence protocol.

3.1 Overview of the CCM Operations

In a standard multi-core with a distributed shared L2 cache and distributed directory coherence, upon an L1 read miss, the request is sent to the L2 cache controller of the home node of the target cache block. If the requesting node is not the home node, the request is sent via the NoC to the home node. Then, the L2 cache controller of the home node looks up the directory to determine the coherence state of the desired cache block. If the cache block is shared or clean, the data is read from the L2 cache,

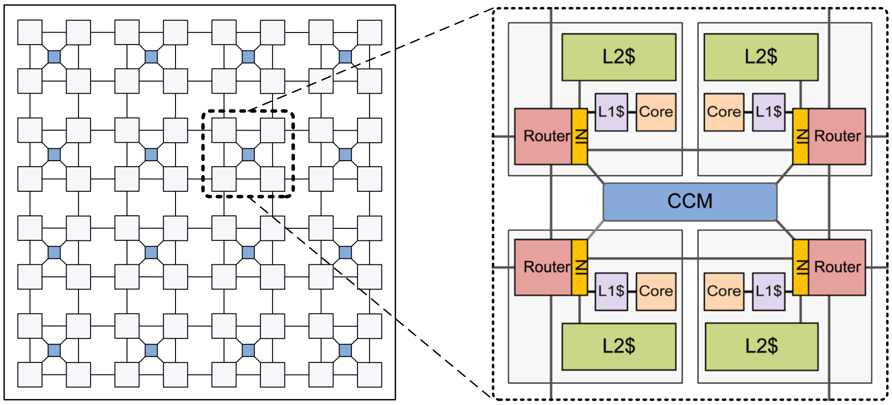

Fig. 5 Block diagram of cluster based CMP, and detailed structure of one 2 × 2 cluster

the requesting node id is recorded in the directory, and the cache block is marked as shared. If the cache block is dirty, and thus exclusively held by another L1 cache, the request is forwarded to that L1 cache node. Upon receiving the request, that L1 cache sends a copy to the requesting node, and a sharing write-back message to the home node, which then updates the cache block directory state.

As explained in the introduction, in a CCM-based multi-core, we group nodes into 2×2 clusters, as shown in Fig. 5. When the network interface of one router receives an L1 miss request, it holds the requests in its buffer and firstly dispatches the request to the CCM for lookup before injecting it in the NoC. The CCM contains a copy of all L1 tags in the cluster, and as a result, it can decide if the L1 miss request can be satisfied by one of the cluster L1 caches. The CCM will return the lookup result to the network interface. Based on information provided by the CCM, the network interface decides canceling the L1 miss request from its buffer or injecting it in the NoC. In theory, the requesting node should still send a message to the home node so that the cache block directory state is set to shared. In order to avoid that long-distance communication, we add two new states in the coherence protocol called *Cluster Exclusive* (*CE*) and *Cluster Modified* (CM). In Sect. 3.3, we detail the modification of the coherence protocol.

3.2 Hardware Structure of CCM

The architecture block diagram of the CCM is shown in Fig. 6. The CCM processes all messages related to the L1 cache misses of its cluster, i.e., all messages to/from the four network interfaces of the cluster, see Fig. 5. The outgoing messages correspond to read or write L1 miss requests. The incoming messages correspond to data or invalidation messages from remote nodes.

3.2.1 CTA

The main component of the CCM is the Cluster Tag Array (CTA), see Fig. 6a, which contains a copy of all cluster L1 tags and their coherence states, implemented as one

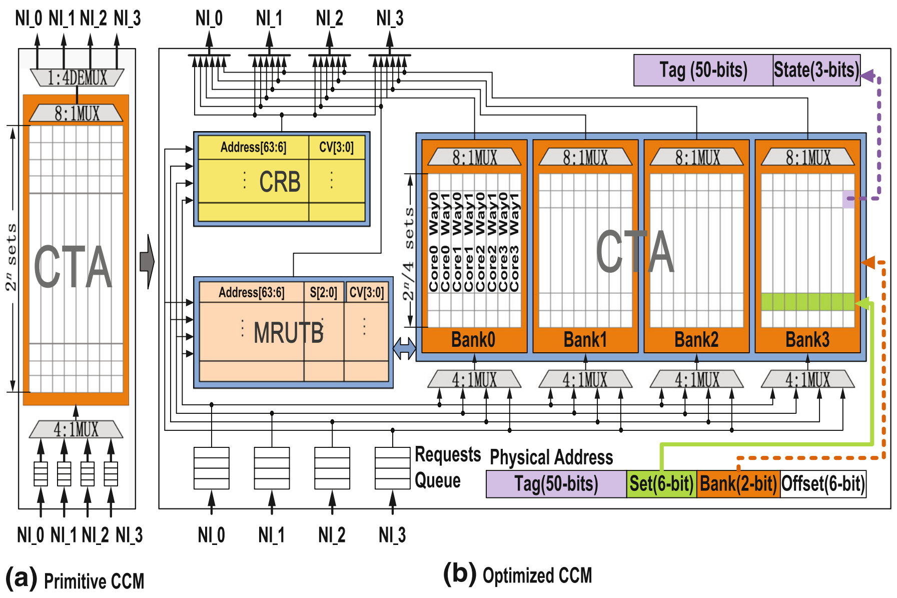

Fig. 6 Block diagram of CCM (CTA, CRB, MRUTB and network interfaces NI_i)

single-port SRAM bank. The CTA associativity is 8-way, which is equal to the sum of four L1 associativity. The CTA is indexed using the L1 miss request physical address, and an access returns the corresponding entry of the L1 tags (address and coherence vector) of all four cores. The baseline design of the CCM, shown in Fig. 6a, is merely composed of the CTA. Since the CCM must process the requests of the four network interfaces, and since this simple CTA implementation can process only one request at a time, we buffer incoming requests in order to cope with L1 miss requests from two or more network interfaces issued at the same cycle, see the bottom of Fig. 6a. In order to reduce the impact of such simultaneous CCM requests on the overall L1 miss latency, we have partitioned the CTA into four single-port banks, as shown in Fig. 6b. The addresses are interleaved according to the least significant bits of the index. The multiplexers under CTA indicate the bank in which to find the line. Requests which do not result in bank conflicts are processed in parallel.

3.2.2 MRUTB and CRB

The CTA is the key component for realizing the functionality of the CCM. However, we can further improve the CCM performance in two ways: by speeding up simultaneous accesses to the same CCM entries, and by eliminating redundant requests to remote nodes (requests for the same address in a remote L2 cache bank). For that purpose, we respectively introduce two new hardware structures, the MRUTB and the CRB.

MRUTB We have empirically observed that there is a high probability that when one L1 cache misses on an address, some of the three other L1 caches of the cluster will simultaneously or soon miss on the same address. As a result, we can reduce conflicts in the CTA and speed up L1 miss requests by adding a small multi-ported buffer, conceptually acting as a CTA cache, and capable of simultaneously serving multiple requests for the same entry. This buffer is called the *Most Recently Used Tag Buffer* (MRUTB), which is a 16-entry register array. As shown in Fig. 6b, each MRUTB entry is composed of three fields: the physical address tag (58 bits for our configuration), the cache coherence state (3 bits), and the sharer information (4 bits, one per cluster node). Because the MRUTB is small, we can use a high number of ports (exactly 4 read/write ports) in order to simultaneously serve the requests from all four network interfaces without significantly increasing the overall size of the CCM. Naturally, concurrent write requests to the same register are not allowed. With the MRUTB, a CCM look up proceeds as follows: the network interface first looks up the MRUTB; upon a miss in the MRUTB, it then accesses the CTA.

CRB The same way all four L1 caches of a cluster tend to request the same data more or less at the same time, if none of the L1 caches contains the requested data, they are likely to issue simultaneous, and thus redundant, requests to a remote L2 cache bank. In order to avoid these costly redundant requests, we introduce a hardware structure, called the *Cluster Request Buffer* (CRB), which is a 4-entry 4-port buffer, and which realizes a functionality akin to a miss address file, in order to capture such hits-on-miss.

Each entry of the CRB contains the physical address tag, and a 4-bit mask indicating which of the four network interfaces request that address. The CRB is checked in parallel with the MRUTB. Upon a miss in the MRUTB, but a hit in the CRB, the bit for the corresponding network interface is set in the mask, and the miss request is not sent to the network. When a remote request comes back, the CRB is checked, and the data is forwarded to the corresponding network interfaces.

Latencies For the L1 (and thus CTA) dimensions used in this article (32 KB, 64B-line, 2-way, see Sect. 4), we evaluated by CACAT6.0 [25] that the CTA access time is 2 cycles. A hit in the CCM can have a latency as low as 3 cycles: 1 cycle to send the request from the network interface to the CCM, 1 cycle to look up the MRUTB (and CRB), and 1 cycle to send the response from the CCM to the network interface. Upon an MRUTB miss, the CCM looks up the CTA. Since the latency of the CTA is 2 cycles, the best case for processing a CCM request with an MRUTB miss is 5 cycles $(3+2)$. However, in spite of the CTA bank partitioning, bank conflicts can occur, with a worst-case of 8 cycles to access the CTA, corresponding to an overall worst-case CCM latency of 11 cycles $(3 + 8)$.

CTA Updates We distinguish three CTA update cases.(1) If an L1 miss request can be served by one of the cluster nodes, then the cache block coherence state (sharing information) is updated.(2) If the L1 miss request cannot be served, it is temporarily stored in the CRB. When the reply comes back from the L2, the CTA is updated, and a CRB look-up indicates which requests are waiting for that reply; the corresponding L1 cache blocks are then updated as well. (3) Upon receiving an invalidation or data transfer message, the CTA states are updated accordingly, as well as the L1 cache blocks. Note that when a CTA entry needs to be updated after a miss is served, and that entry resides in the MRUTB, it is updated in the MRUTB, not the CTA. Upon eviction from the MRUTB, the modified entry is written back to the CTA.

3.2.3 Modifications of Network Interface

Implementing CCM does not require to modify the router internal structure, only its interface. We add one port to the network interface,[2] which connects to the CCM, see Fig. 5. The network interface holds requests from a cluster node during the CCM lookup. If the lookup is successful, that request is canceled. On the contrary, if the lookup is missing, the network interface injects the request in NoC.

3.3 CCM Coherence Protocol

The CCM coherence protocol is based on the MESI (M: Modified, E: Exclusive, S: Shared, I: Invalid) protocol [21]. The CCM is meant to save accesses to remote nodes upon L1 misses. However, in theory, the home node must still be accessed whenever the state of one cache block changes in order to update it. If these home nodes are located outside the cluster, these state updates also correspond to costly remote accesses. However, it is possible to significantly reduce such home node accesses provided the coherence protocol factors in the multi-core organization into clusters.

Let us consider one example where an L1 cache does a read miss on an address and fetches the data from a remote L2 bank. The L1 controller will fetch the data, and, in the process, the home node must also be accessed in order to update the memory block state, i.e., a remote access if the home node is not located in the cluster. Let us assume this L1 will be the only one having a copy, and the state is exclusive. If another L1 of the same cluster later misses on the same memory block, thanks to the CCM, there is no need to fetch the data again remotely; however, the home node of the memory block must still be accessed to mark it as shared, inducing a remote access. It is possible to avoid this second remote access by introducing a state CE. CE means that one or several clean copies of the memory block reside in the cluster, albeit exclusively in that cluster. The first time an L1 cache of a cluster misses on a given memory block, a long-distance request is sent to the home node to fetch the memory block and to set its state to E. All subsequent L1 misses on that memory block, coming from other L1 caches within that cluster, require no long-distance communication, and as a result, the state is converted to CE. Note that if a node has a cache block with state E, no read or write operation needs to access the CCM. However, when writing a block with state CE, the CCM sends invalidation messages to other owners in the cluster. Similarly, we introduce a second state Cluster Modified (CM) which means that a memory block is modified and held by several L1 caches of a cluster.

While it is not possible to exhaustively describe all possible state transitions induced by the introduction of CE and CM, we indicate the main state transitions in the diagram of Fig. 7, where we distinguish between read/write and local/cluster/remote accesses ($R/W_{l/c/r}$). And, we detail several typical coherence scenarios in Fig. 8, with Fig. 8a

[2] One interface has 3 ports, one for L1 cache, one for L2 cache, and one for CCM.

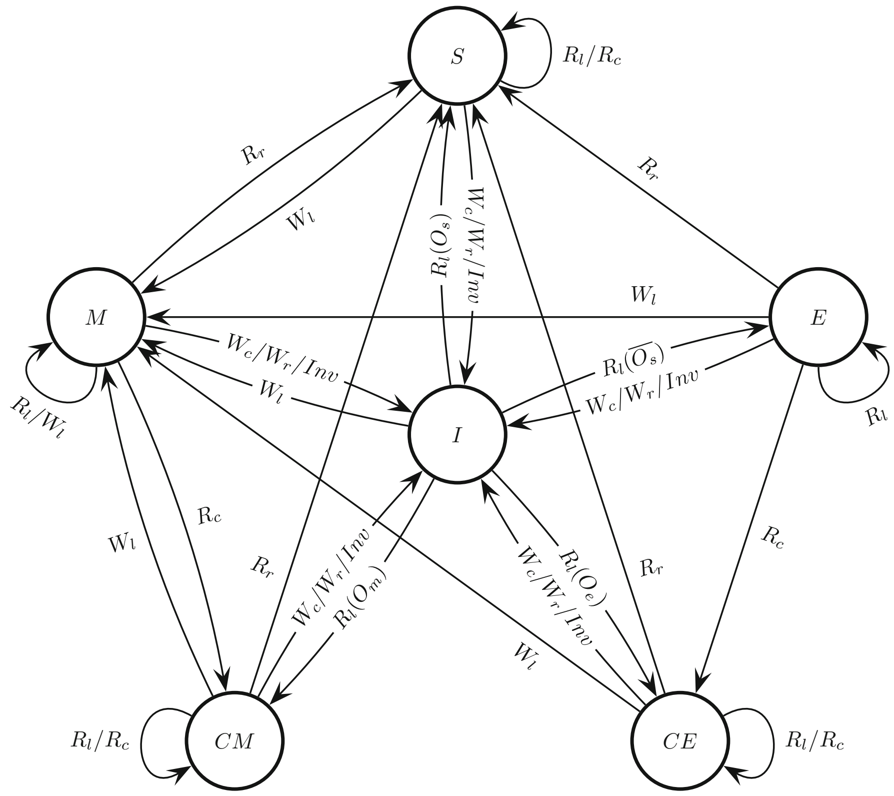

Fig. 7 L1 cache state transition diagram (*Inv*: Invalidate, O_s: shared, $\overline{O}_s$: non-shared, O_e: state is *E* or *CE*, O_m: state is *M* or *CM*)

–e corresponding to the cases where a cluster sends a request, and Fig. 8f–h to the cases where a cluster receives a request.

Read Hit in Cluster As shown in Fig. 8a, when the required memory block can be served by the intra cluster nodes, the request is forwarded to the closest node O_1, which will dispatch the memory block to requester R.

Write Hit in Cluster There are two possible cases corresponding to Fig. 8b, c. We first consider the case where the memory block is exclusively owned (e.g., by O_1), i.e., its state is E, M, CE or CM, As shown in Fig. 8b, the CCM then forwards the request from R to O_1, and the invalidation message to O_2. Then R receives the data from O_1 and acknowledges the message of O_2. If the memory block state is shared, i.e., by cluster cores (e.g., O_1 and O_2) and non-cluster cores, then, as shown in Fig. 8c, a write request is issued to the home node, which sends an invalidation message to all clusters having a copy of the memory block. Moreover, the CCM invalidates the copies in the cores of the cluster, and R must wait for the acknowledgment of all these invalidation messages.

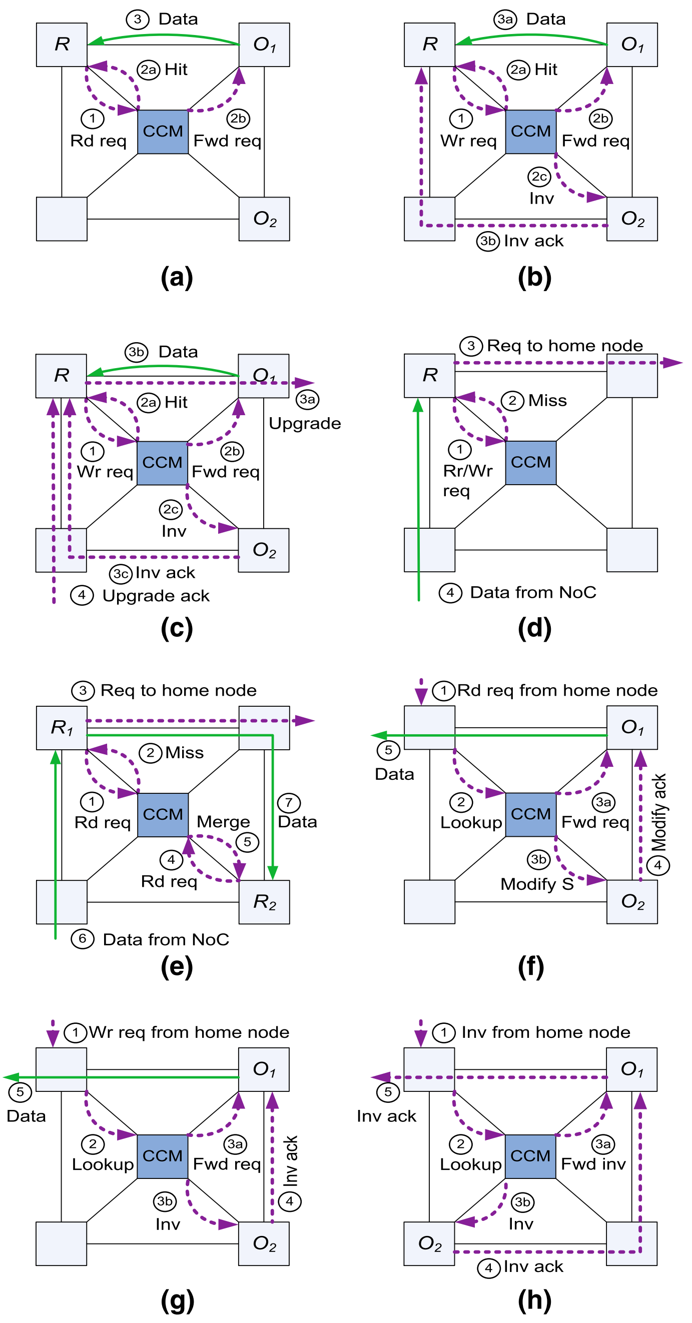

Fig. 8 Different coherence protocol scenarios (*purple dashed arrows* are control messages, *green solid arrows* are data transfers). **a** Read hit in cluster, **b** write hit in cluster, exclusively owned by cluster, **c** write hit in cluster, shared, **d** read/write miss in cluster, **e** Merging two miss requests, **f** incoming read request, exclusively owned by cluster, **g** incoming write request, exclusively owned by cluster, **h** incoming invalidation message, shared (Color figure online)

Incoming Request for an Exclusively Owned Data If the cluster exclusively owns a memory block, i.e., its states are E, M, CE or CM, and a network interface receives a read request from an external node, it first looks up the CCM in order to find all L1 caches having a copy of the memory block (e.g., O_1 and O_2). In the example of Fig. 8f, the CCM would forward the request to O_1 because it is closest to the core/interface which received the request. O_1 converts the cache block state to S, and the CCM also requests O_2 to do the same. When O_1 receives the state modification acknowledgment from O_2, O_1 sends the data to the external node which requested it. A similar case with an incoming write request is shown in Fig. 8g; the main difference is that all L1 caches containing a copy change the cache block state to I instead of S.

Incoming Invalidation Message for a Shared Data As shown in Fig. 8h, when the data in a cluster is in the shared state, and the cluster receives an invalidation message, all copies within the cluster are invalidated, and an acknowledgment message is then sent back.

4 Experimental Methodology

We use the full system simulation environment Simics [23] to implement a 64-core Sparc V9 CMP, and the GEMS toolset [24] to implement the cycle-accurate memory hierarchy models. We model the NoC using Garnet [3], which is a detailed cycle-accurate network simulator included in GEMS. We derive the power model of the interconnection network from Orion 2.0 [19], which describes the power of the buffers, crossbars, arbiters and links. We measure energy consumption of the memory hierarchy (on-chip caches and off-chip memory) using CACTI6.0 [25]. We have estimated the capacitances of all these components using the technology parameters of CACTI.

The default simulation configuration parameters are shown in Table 1. It should be noted that, in order to accurately evaluate the impact of the NoC, we applied the fixed model of Garnet, which cycle-accurately simulates the buffers, crossbars, arbiters and links. For the energy evaluation, we assumed a 45 nm CMOS NVT technology, and optimally-buffered 2.5 mm inter-tile copper wiring.

We used 23 parallel workloads from two benchmark suites (SPLASH-2 [29] and PARSEC [9]), listed in Table 2. In order to distinguish the SPLASH `raytrace` from the PARSEC `raytrace`, we call the PARSEC version `praytrace`. The programs are run from beginning to end on Solaris 10 OS. The proposed CCM architecture was compared against the standard distributed shared L2 cache organization with a directory coherence protocol, Adaptive Selective Replication (ASR) [6], Distributed Cooperative Caching (DCC) [17] and Reactive NUCA(RNUCA) [16]. The configurations of the different mechanisms are the following.

4.1 Distributed Shared L2, Directory Coherence (Baseline)

L1 and L2 caches are inclusive. Addresses are interleaved across L2 cache banks in order to reduce hot spots. We use bits [19 : 14] of the physical address for inter-

Table 1 Architecture configuration

Component	Parameter
Cache Technology	45 nm CMOS NVT
NoC Technology	45 nm CMOS NVT
Memory Technology	68 nm
CMP Size	64 cores
Processor Model	Sparc V9, in-order
Frequency	2 GHz
Cache Line Size	64 B
L1 I-Cache Size/Associativity	32 KB/2-way
L1 D-Cache Size/Associativity	32 KB/2-way
L1 Load-to-Use Latency	2-cycle
L1 Replacement Policy	LRU
L2 cache Size(total)/Associativity	16-MB/8-way
L2 Load-to-Use Latency	15-cycle
L2 Replacement Policy	Pseudo-LRU
Network Topology	8×8 2D-Mesh
Garnet Configuration	Fixed
V-Net NO./V-Channel NO.	2/2
Buffers per V-Channel	4
Crossbar Model	MULTREE
Virtual Channel Arbiter Model	MATRIX
Switch Arbiter Model	MATRIX
Link Length	2.5 mm
Flit Width	32-bit
Hop Latency	5-cycle
External Memory Size	8-GB
External Memory Latency	300 cycles

leaving instead of the least significant bits for the sake of a fair comparison. We empirically found that interleaving using least significant bits induce many more hot spots, and thus greatly exaggerated the performance benefits of CCM. Consequently, we selected the address interleaving bits which, on average, induce the least hot spots.

4.2 Cluster Cache Monitor (CCM)

We considered two configurations. The first one corresponds to the baseline CCM design of Fig. 6a which only contains the CTA, and the second one to the more sophisticated design of Fig. 6b which contains a banked CTA, an MRUTB and a CRB. Unless otherwise specified, in figures, CCM stands for the latter version. We use a cluster of 2×2 tiles, and we explore in alternative cluster sizes in Sect. 5.6.

Table 2 Benchmarks

SPLASH2 Workloads	Inputs	PARSEC Workloads	Inputs
barnes	32768 particles	blackscholes	simlarge
cholesky	tk 29.0	bodytrack	simlarge
fft	-m 20	canneal	simlarge
fmm	32,768 particles	dedup	simsmall
lu	1,024 × 1,024	facesim	simsmall
ocean	514 × 514 grid	ferret	simsmall
radiosity	room model	fluidanimate	simlarge
radix	1M-keys	freqmine	simsmall
	1024-radix	(p)raytrace	simlarge
	2M-maxkey	streamcluster	simsmall
raytrace	teapot.env	swaptions	simlarge
volrend	head	vips	simlarge
		×264	simlarge

4.3 Adaptive Selective Replication (ASR)

We implement distributed ASR (private L2s) where each L2 comes with a 16K-entry 16-way set-associative Next Level Hit Buffers (NLHBs), and 1K-entry 16-way set-associative Victim Tag Buffers (VTBs), as in ASR[6].

4.4 Distributed Cooperative Caching (DCC)

We consider a DCC configuration (private L2s) where each core node comes with one DCE, as in DCC[17].

4.5 Reactive NUCA (RNUCA)

We implement Reactive NUCA (shared L2) with size-1 clusters for core-private data blocks, size-4 for instructions and size-64 for shared data blocks, see RNUCA [16].

5 Performance Evaluation

In this section, we first present the energy benefits of CCM and compare it against ASR , DCC and RNUCA. We then present its execution time benefits, and analyze how CCM improves the overall L1 miss latency, with a special emphasis on network traffic reduction. Finally, we compare the hardware cost of the different mechanisms, and discuss some details of CCM.

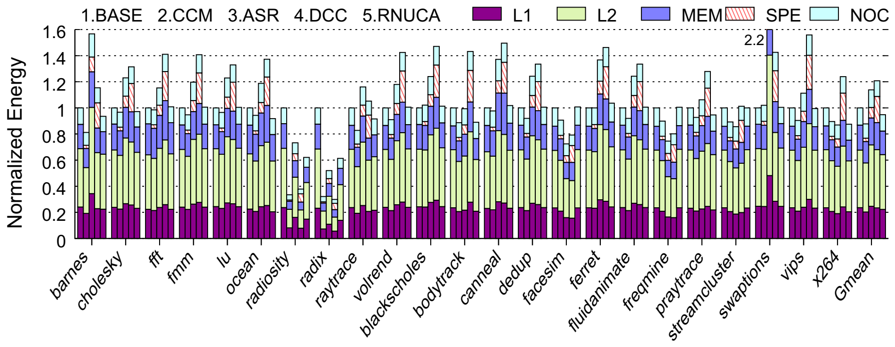

Fig. 9 Energy consumption

5.1 Energy

For all mechanisms, we break down the energy consumed in the memory system into L1 and L2 caches, memory accesses (MEM), network accesses (NoC), and mechanism-specific blocks (SPE). For CCM, SPE corresponds to the CCM itself (CTA, MRUTB, CRB), for ASR, it is the NHLB and the VTB, and for DCC, it is the DCE; RNUCA only requires one additional bit per TLB entry. The energy reported in Fig. 9 is the full energy (static + dynamic).

CCM requires less overall energy than ASR, DCC and RNUCA for two reasons: (1) because the CCM is small compared to the respective additional structures of ASR and DCC, so that the L1 + L2 static energy is lower, and (2) because the overall execution time of CCM is lower than all three other mechanisms, thanks in large part to the lower L1 miss latency. CCM also removes more L2 misses than ASR and DCC (55 and 59 %, respectively) thanks to the inherent assets of shared L2 caches over private L2 caches, even with sophisticated mechanisms such as ASR and DCC. CCM also reduces the NoC traffic more than ASR and RNUCA.

The MRUTB and the CRB further increase the energy consumption, but thanks to their small size, this energy overhead accounts for less than 1 % of the total energy spent in serving L1 misses.

5.2 Performance

In Fig. 10, we report the normalized execution time of the different mechanisms with respect to the baseline system. On average CCM reduces execution time by 15 % over the baseline. We can also note that the performance of none of the benchmarks is degraded, in spite of the additional cycles of the CCM look-up. The performance improvements are due to the reduction of the average L1 miss latency, especially the elimination of many home node congestions, as further analyzed in Sect. 5.4.

CCM also outperforms ASR, DCC and RNUCA on average. In a few cases, such as `radiosity`, `radix` or `x264`, DCC or ASR outperform CCM. For `x264`, more than 60 % of CCM accesses correspond to the merge case of Fig. 8e, which reduces

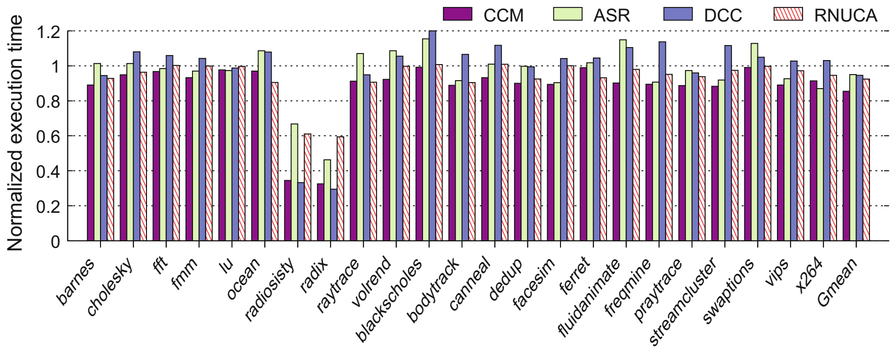

Fig. 10 Execution time with respect to baseline

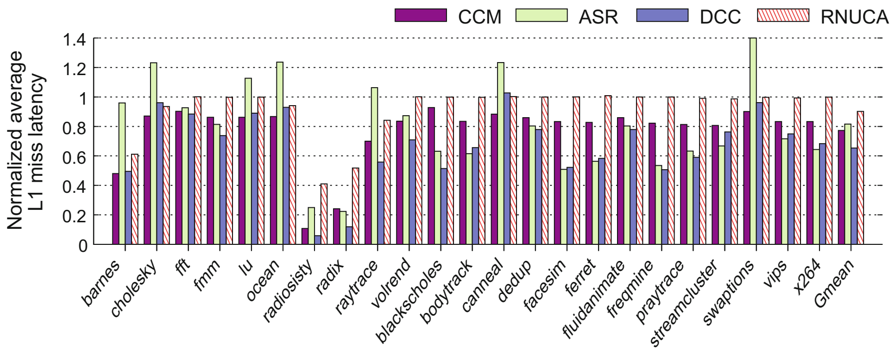

Fig. 11 Average L1 miss latency normalized to the baseline

the memory traffic, but not the latency. For `radix` and `radiosity`, their input set is small and fits in local private L2s, which then have a definite advantage over a shared L2. For instance, in DCC, respectively 82.5 and 94.6 % of the L1 misses of these two benchmarks are serviced by the local private L2.

5.3 L1 Miss Latency Analysis

One of the benefits of CCM is to reduce the average L1 miss latency. In Fig. 11, we report the normalized average L1 miss latency. On average, CCM reduces the latency by 22 %, and up to 89 % with respect to the baseline. On average, CCM is also consistently more efficient than ASR and RNUCA at reducing the miss latency. We can note that the average L1 miss latency of CCM is higher than that of DCC. This is due to the highly varying L1 miss latency of DCC across nodes, very low for some nodes, high for other nodes, resulting in a low average; this phenomenon is illustrated for `volrend` in Fig. 12. Still, the overall execution time of DCC will be worse than CCM because, in parallel workloads, the execution time will be determined by the slowest node.

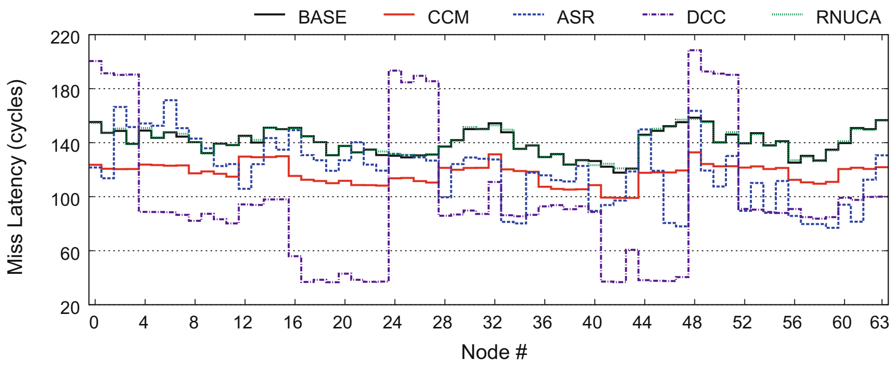

Fig. 12 Average L1 Miss latency of `volrend` across nodes

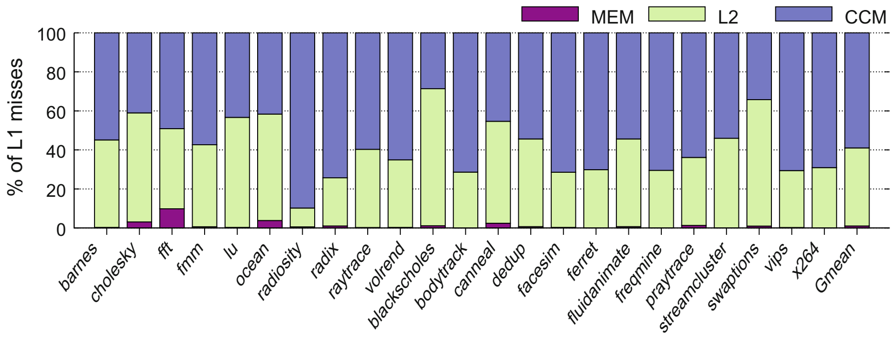

Fig. 13 Structure serving misses in CCM

In general, CCM reduces the L1 miss latency in two ways: (1) by serving L1 misses directly, within clusters, and (2) by avoiding L2 congestions. In Fig. 13, we indicate by which structure L1 misses are serviced (CCM, L2, Memory). On average, clusters can eliminate 59 % of all L1 misses, and up to 89 % in `radiosity`.

For instance, the baseline L1 miss latency of `fft` is 178 cycles, while the baseline latency of `radiosity` is 679 cycles. These drastic latency variations are due to the presence of multiple simultaneous requests at some nodes, which is itself due to either simultaneous misses or the unbalanced allocation of home nodes across cores by the operating system. By capturing many of the L1 misses within the cluster, CCM can considerably reduce this congestion e.g., latency is down to 147 cycles for `radiosity`, and162 cycles for `fft`. The overhead brought by CCM remains small across all benchmarks, as shown in Fig. 14.

5.4 Network Traffic

CCM can reduce the network traffic in three ways: (1) implicitly by serving L1 misses within the cluster, (2) by merging L1 misses using the CRB, (3) because the cluster organization transforms invalidation messages to individual nodes into invalidation

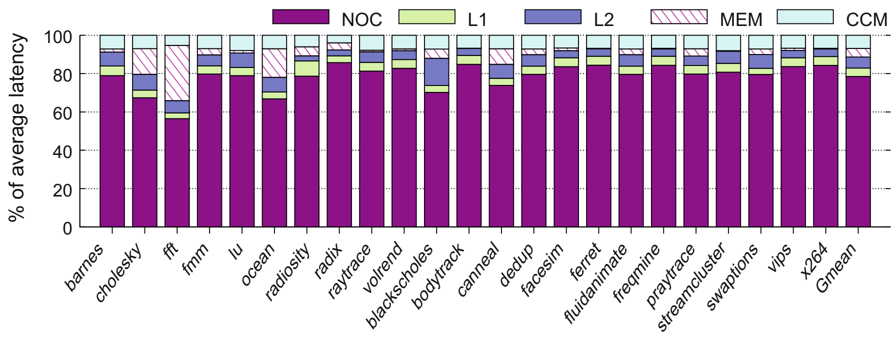

Fig. 14 CCM L1 miss latency breakdown

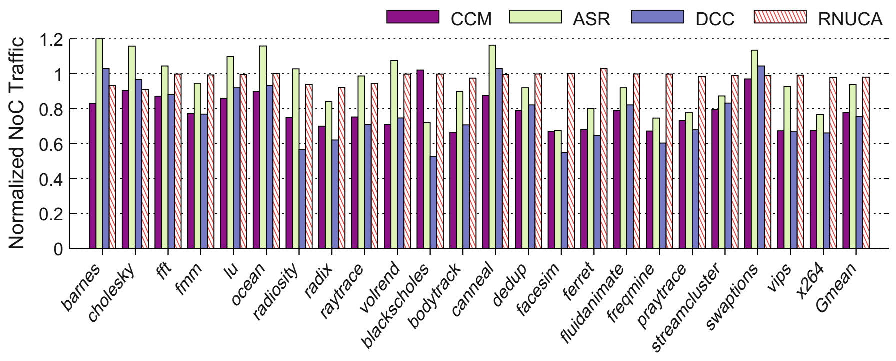

Fig. 15 Network traffic

messages to whole clusters; when multiple nodes to be invalidated belong to the same cluster, the number of invalidation messages is reduced accordingly. In Fig. 15, we show that CCM lowers the network traffic by 21 % on average compared to the baseline, and brings more network traffic reduction than ASR and RNUCA. For DCC, it should be noted that the traffic between the L1 and its private L2 was not factored in, while the cluster NoC traffic of the CCM was factored in. As a result, DCC appears to reduce the network traffic slightly more than CCM.

We further analyze the reduction of network traffic, and we especially focus on the reduction of congestions which have a strong impact on the L1 miss latency, see Sect. 5.3. For that purpose, we visualize the total number of flits passing through each router (one square is one router) for `radiosity` in Fig. 16, for the baseline architecture, CCM, ASR, DCC and RNUCA. Dark (blue) squares correspond to high flit activity, while light (green) or white squares correspond to little or no activity. The baseline system exhibits several hotspots, with one in particular at the node located on the 5th row and the 1st column. On the CCM graph, the hotspots have disappeared and the traffic at many nodes has become fairly low. While ASR and DCC also successfully remove hotspots, the overall traffic remains high.

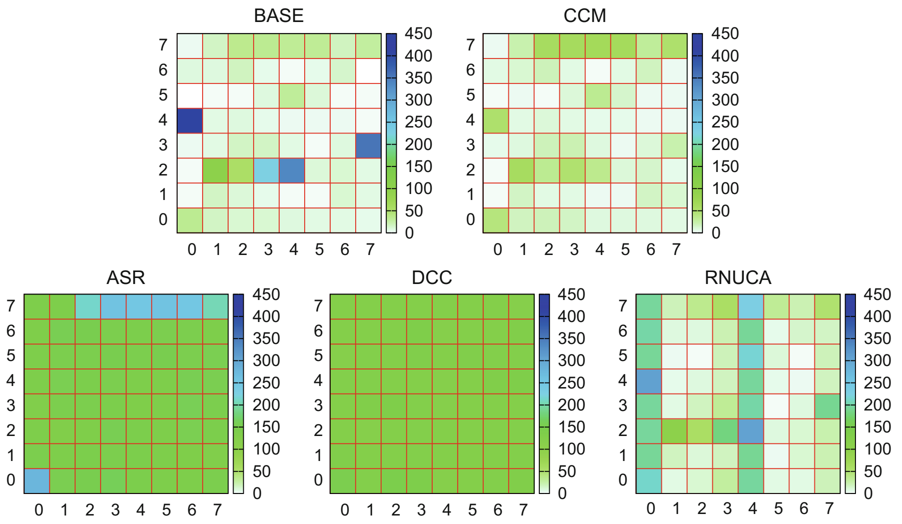

Fig. 16 Heat map of flit router activity (the color scales are in millions of flits) (Color figure online)

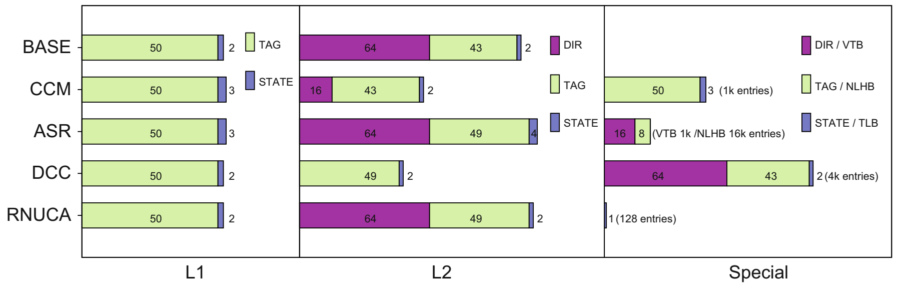

Fig. 17 Structure of tag, directory, and state bits

5.5 Hardware Cost

We evaluate the hardware cost in KBytes of the different hardware mechanisms. In Fig. 17, for each mechanism, we detail the structure of the tag, directory and state bits which largely account for the storage structures sizes. We also implemented the CCM in Verilog and synthesized it using Synopsys. The cost of the combinational logic (MUXes, comparators) is only 14.7 k gates, which is negligible compared to all other storage structures. In the comparison of Table 3 we include both the directory size and the mechanism-specific structures. In spite of the CTA, MRUTB and CRB, the total area required for implementing the CCM is actually 27.9 % smaller than for the baseline architecture, because CCM reduces the cost of the directory by replacing node-based sharer information with cluster-based sharer information, see Table 3. Because of the implementation of DCE in DCC and NLHB/VTB in ASR , the additional hardware overhead of DCC and ASR are 41.8 and 36.3 % , respectively.

Table 3 Hardware cost of one node

Design	Hardware cost (KB)				
	L1	L2	Special	Total	Area (%)
BASE	6.5	54.5	0	61	0
CCM	6.625	30.5	6.79	43.91	−28
ASR	6.625	58.5	18	83.125	36.3
DCC	6.5	25.5	54.5	86.5	41.8
RNUCA	6.5	57.5	0.015625	64.02	4.9

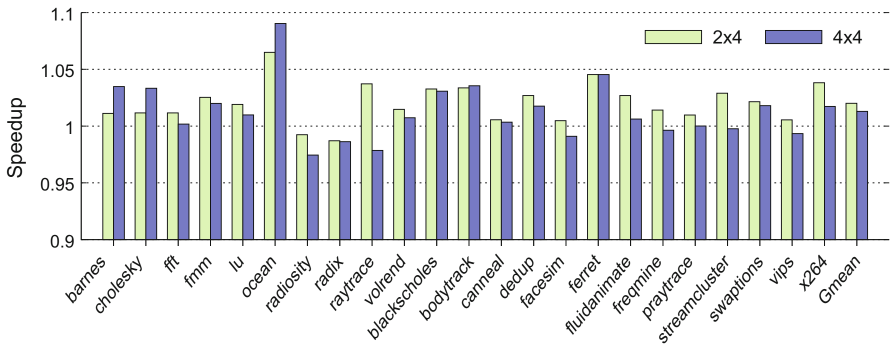

Fig. 18 Speedup with respect to 2 × 2 cluster

5.6 Cluster Size

In this section, we analyze the impact of the cluster size. We try two additional cluster configurations: 2 × 4 and 4 × 4, and we report the results in Figure 18. For 4 × 4, there is only a minor increase in performance with respect to the 2 × 2 cluster, i.e., a speedup of 1.01. The analysis shows that a larger cluster captures only a small additional fraction of L1 misses, while it increases the L1 latency, and contention within the CCM. Moreover, for about 1/3 of the benchmarks, the 4 × 4 configuration actually performs worse than the 2 × 2 configuration. We also experiment with an intermediate 2 × 4 cluster configuration, and we find that it is slightly better than the 4 × 4 configuration, and again only marginally better than the 2 × 2 configuration, because it is less hampered by the longer latency and higher contention compared to the 4 × 4 configuration. Overall, while a 2 × 4 configuration or even a 4 × 4 configuration might outperform a 2 × 2 configuration, the benefits are only marginal.

5.7 CTA Conflicts

In this section, we analyze the CCM optimization which consists in partitioning the CTA and integrate the MRUTB in order to allow parallel accesses to the CCM from the four different network interfaces. In Fig. 19, we compare the fraction of conflicts

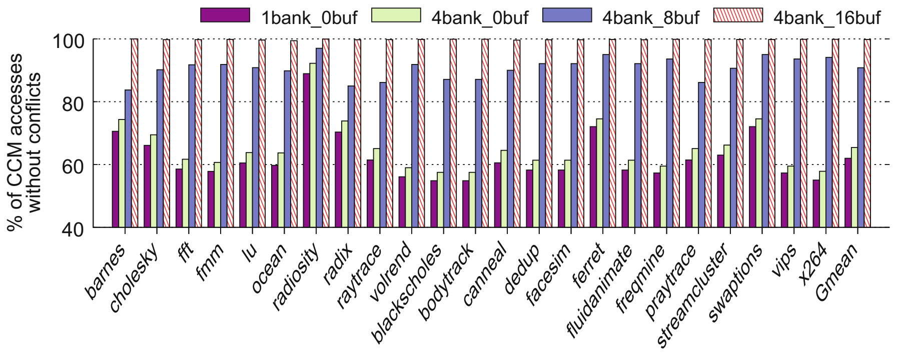

Fig. 19 % Of CCM accesses without conflicts, for different number of CTA banks and MRUTB sizes

of the primitive CCM (non-partitioned CTA, no MRUTB, see `1bank_0buf`) and the optimized CCM design with different configurations, including 4-partition without MRUTB, 4-partition with 8-entry MRUTB, and 4-partition with 16-entry MRUTB. We find that the proposed optimization reduces conflicts by 38 % on average. The 16-entry MRUTB resolved 34 % conflicts on average, while merely increasing the CCM hardware cost by 130 Bytes.

6 Conclusions

As the number of cores increases, recent research, such as ASR and DCC, have tilted the balance between shared and private L2 towards private L2, in spite of the capacity advantage of shared L2. However, we show that the same locality principles which have guided the design of memory systems at large can be once more leveraged to significantly improve the performance of shared L2. We empirically observe a form of node-based spatial locality for L1 caches, and we show that a clustered organization of the CMP, coupled with our CCM structure and a modification of the coherence protocol, can successfully leverage that property. As a result, the shared L2-based CCM significantly outperforms even recent private L2 mechanisms such as ASR or DCC, or alternative shared L2 mechanisms such as RNUCA, from both an energy and execution time standpoint. And because of the clustered structure (and modified coherence protocol) of CCM, the overall storage size is reduced compared to the baseline architecture.

References

1. Acacio, M.E., González, J., García, J.M., Duato, J.: Owner prediction for accelerating cache-to-cache transfer misses in a cc-NUMA architecture. In: Supercomputing, ACM/IEEE 2002 Conference, pp 49–49 (2002)

2. Agarwal, A., Paul, B., Mahmoodi, H., Datta, A., Roy, K.: A process-tolerant cache architecture for improved yield in nanoscale technologies. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **13**(1), 27–38 (2005)
3. Agarwal, N., Krishna, T., Peh, L., Jha, N.: Garnet: a detailed on-chip network model inside a full-system simulator. In: Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09), pp 33–42 (2009)
4. Barroso, L., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., Verghese, B.: Piranha: a scalable architecture based on single-chip multiprocessing. In: Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00), ACM, pp 282–293 (2000)
5. Barrow-Williams, N., Fensch, C., Moore, S.: Proximity coherence for chip multiprocessors. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10), ACM, pp 123–134 (2010)
6. Beckmann, B.M., Marty, M.R., Wood, D.A.: (2006) ASR: Adaptive Selective Replication for CMP Caches. In: Proceedings of 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), pp 443–454
7. Bell, S., Edwards, B., et al.: TILE64—Processor: A 64-Core SoC with Mesh Interconnect. In: Proceedings of 2008 IEEE International Solid-State Circuits Conference (ISSCC'08), pp 88–598 (2008)
8. Beu, J., Rosier, M., Conte, T.: Manager-client pairing: a framework for implementing coherence hierarchies. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11), ACM, pp 226–236 (2011)
9. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC Benchmark Suite: Characterization and Architectural Implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08), pp 72–81 (2008)
10. Brown, J., Kumar, R., Tullsen, D.: Proximity-aware directory-based coherence for multi-core processor architectures. In: Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07), ACM, pp 126–134 (2007)
11. Chang, J., Sohi, G.S.: Cooperative Caching for Chip Multiprocessors. In: Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA'06), pp 264–276 (2006)
12. Chishti, Z., Powell, M.D., Vijaykumar, T.N.: Optimizing replication, communication, and capacity allocation in CMPs. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05), pp 357–368 (2005). doi:10.1145/1080695.1070001
13. Eisley, N., Peh, L., Shang, L.: In-network cache coherence. In: Proceedings of 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), pp 321–332 (2006)
14. Ferdman, M., Lotfi-Kamran, P., Balet, K., Falsafi, B.: Cuckoo directory: a scalable directory for many-core systems. In: Proceedings 17th International Symposium on High Performance Computer Architecture (HPCA'11), pp 169–180 (2011)
15. Gupta, A., Weber, W.D., Mowry, T.C.: Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In: Proceedings of 1990 International Conference on Parallel Processing (ICPP'90), pp 312–321 (1990)
16. Hardavellas, N., Ferdman, M., Falsafi, B., Ailamaki, A.: Reactive NUCA: near-optimal block placement and replication in distributed caches. In: Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09), pp 184–195 (2009)
17. Herrero, E., González, J., Canal, R.: Distributed cooperative caching. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08) pp 134–143. ACM(2008)
18. Hossain, H., Dwarkadas, S., Huang, M.C.: POPS: coherence protocol optimization for both private and shared data. In: Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT'11), pp 45–55 (2011)
19. Kahng, A., Li, B., Peh, L.S., Samadi, K.: ORION 2.0: a fast and accurate noc power and area model for early-stage design space exploration. In: Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'09), pp 423–428 (2009)
20. Kim, C., Burger, D., Keckler, S.: An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In: Proceedings of the 10th Annual IEEE/ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02), pp 211–222. ACM (2002)

21. Laudon, J., Lenoski, D.: The SGI origin: a ccNUMA highly scalable server. In: Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97), pp 241–251 (1997)
22. Maa, Y.C., Pradhan, D.K., Thiebaut, D.: A hierarchical directory scheme for large-scale cache-coherent multiprocessors. In: Proceedings of 6th International Symposium on Parallel Processing (IPPS'92), pp 43–46 (1992)
23. Magnusson, P., Christensson, M., et al.: Simics: a full system simulation platform. Computer **35**(2), 50–58 (2002)
24. Martin, M.M., Sorin, D.J., et al.: Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. Comput. Archit. News (CAN) **33**(4), 92–99 (2005)
25. Muralimanohar, N., Balasubramonian, R., Jouppi, N.: Cacti 6.0: a tool to understand large caches. Technical report, HP Tech Report HPL-2009 (2009)
26. Noel, E., Li-Shiuan, P., Li, S.: Leveraging on-chip networks for data cache migration in chip multiprocessors. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08), pp 197–207 (2008)
27. Roberts, D., Kim, N., Mudge, T.: On-chip cache device scaling limits and effective fault repair techniques in future nanoscale technology. Microprocess. Microsyst. **32**(5), 244–253 (2008)
28. Shin, J.L., Tam, K., et al.: A 40 nm 16-Core 128-Thread CMT SPARC SoC Processor. In: Proceedings of 2010 IEEE International Solid-State Circuits Conference (ISSCC'10), pp 98–99 (2010)
29. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95), pp 24–36 (1995)
30. Yoon, D., Erez, M.: Memory mapped ecc: low-cost error protection for last level caches. In: Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'09), pp 116–127. ACM (2009)
31. Zhang, M., Asanovic, K.: Victim migration: dynamically adapting between private and shared cmp caches. Technical report, Massachusetts Inst Of Tech Cambridge Computer Science and Artificial Intelligence Lab (2005)
32. Zhang, M., Asanovic, K.: Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05), pp 336–345 (2005)
33. Zhang, M., Lebeck, A., Sorin, D.: Fractal coherence: scalably verifiable cache coherence. In: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10), pp 471–482 (2010)
34. Zhang, Y., Lu, Z., Jantsch, A., Li, L., Gao, M.: Towards hierarchical cluster based cache coherence for large-scale network-on-chip. In: Proceedings of the 4th International Conference on Design and Technology of Integrated Systems in Nanoscal Era (DTIS'09), pp 119–122 (2009)