Runtime Home Mapping for Effective Memory Resource Usage

Mario Lodde and José Flich

Universitat Politècnica de València, Spain

Abstract

In tiled Chip Multiprocessors (CMPs) last-level cache (LLC) banks are usually shared but distributed among the tiles. A static mapping of cache blocks to the LLC banks leads to poor efficiency since a block may be mapped away from the tiles actually accessing it. Dynamic policies either rely on the static mapping of blocks to a set of banks (D-NUCA) or rely on the OS to dynamically load pages to statically mapped addresses (first-touch).

In this paper, we propose Runtime Home Mapping (RHM), a new dynamic approach where the LLC home bank is determined at runtime by the memory controller when the block is fetched from main memory, trying to map each block as close as possible to the requestor thus speeding up execution time and lowering message latencies. Block migration and replication provide further improvements to basic RHM. Also, in a further optimization we eliminate the directory structure. All these optimizations involve specific NoC optimizations and co-designs. Results with PARSEC and SPLASH-2 applications show, when compared with alternative solutions, that RHM achieves a 41% and 35% average reduction in load and store latencies respectively compared to static mapping. This leads to an average reduction of 28% in applications execution.

Keywords:
chip multiprocessors, network-on-chip, cache hierarchy, coherence protocols

1. Introduction

Chip multiprocessor systems (CMPs) usually employ a shared memory programming model, thus requiring a cache coherence protocol to keep data

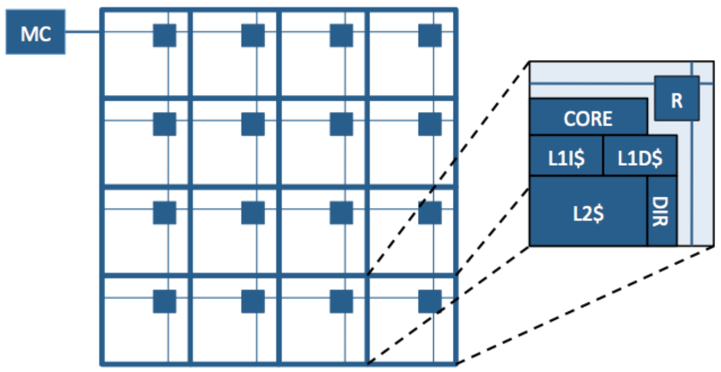


Figure 1: Tiled CMP system.

consistency along the cache hierarchy. The on-chip cache is organized hierarchically, with small low-latency caches at the highest level and larger caches with higher access times at the lower levels. This provides high on-chip storage capacity without the high access latency a single, large cache would have. Without losing generality, in this work we assume the tiled CMP system shown in Figure 1 with a two-level cache. Each tile includes a core, separate L1 caches for instructions and data, a bank of L2 cache and a switch to connect the tiles through a 2D mesh.

L1 caches are private to the core in the tile but different policies can be implemented for the L2 cache. The first option is to use each L2 cache bank as a private cache to the tile, extending its private cache capacity. This is the best option if the working set of the application fits in the L2 cache bank, since all cached data can be accessed without sending requests over the NoC. If the working set does not fit, this policy generates many L2 cache line replacements, and therefore, off-chip requests. Furthermore, shared blocks end up being replicated in different L2 cache banks. The second option is to consider the sum of L2 banks as a shared but distributed L2 cache. Data replication is avoided and cache resources are efficiently used. However, the latency of retrieving a cached data in case of L1 miss will be higher and variable depending on the distance between the L1 cache and the accessed L2 bank. Thus, the mapping of blocks to L2 banks is a crucial design parameter for this approach. In this paper we follow this policy.

The L2 bank that hosts a block is called the *home* bank. There are two main design options when deciding which L2 bank is the *home* for a block. On the one hand, block mapping can be done statically (S-NUCA): the address

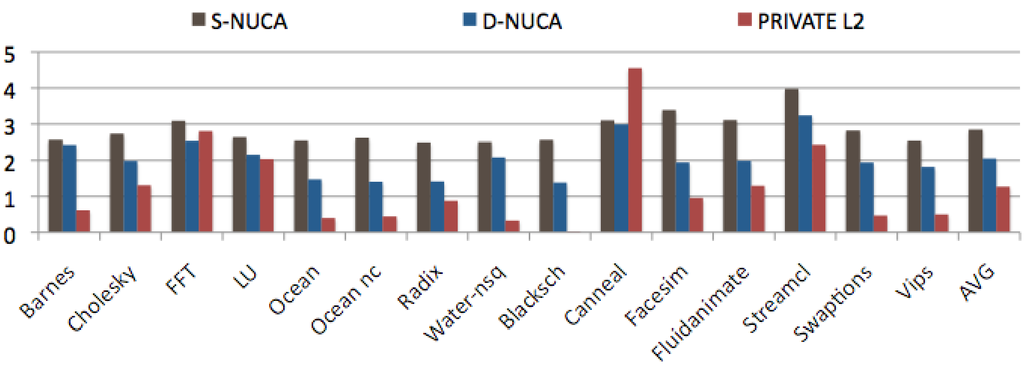
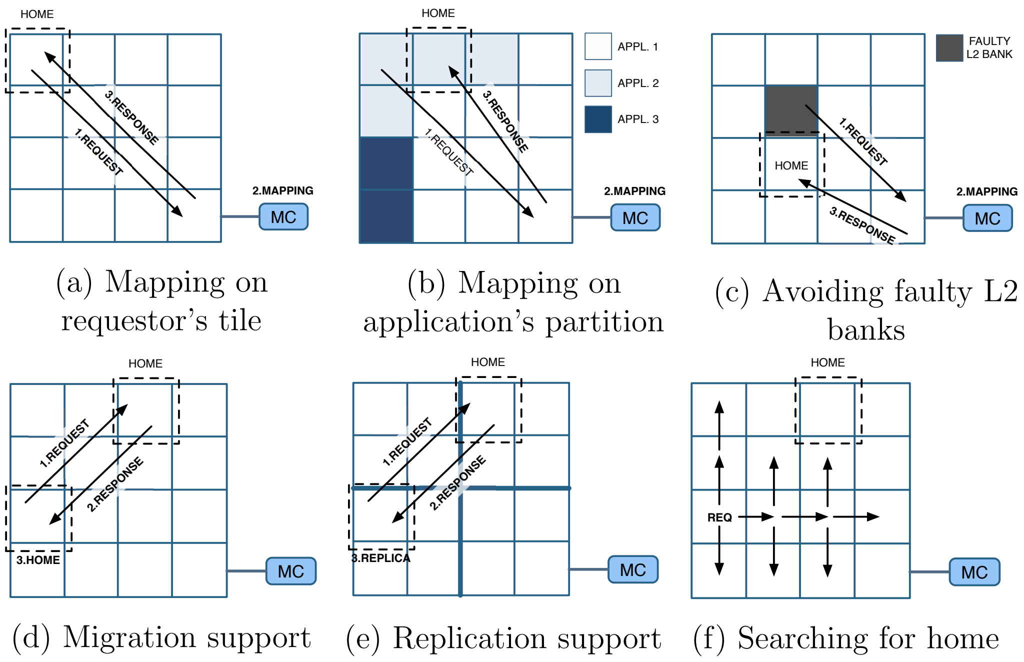


Figure 2: Average distance of L2 banks to their L1 requestors for different mapping policies.

space is divided in subsets and all the blocks of a subset are statically mapped to a bank. This policy is very simple to implement but can be inefficient as blocks may be mapped to banks which are far away from L1 requestors. The second option is to perform the mapping dynamically (D-NUCA) [1], where each subset of blocks is mapped to a group of banks, or bank set, and blocks can migrate within a bank set to move as close as possible to the requestor's tile. This policy has lower miss latencies but is more complex to implement. Furthermore, the process of finding a block within a bank set leads to a tradeoff between access time and NoC traffic since all the banks of a bank set must be accessed, leading to either high latency (sequential search) or more traffic (parallel search).

Figure 2 shows the average distance of accessed L2 banks by their L1 requestors in a 4×4 tiled configuration. The figure shows results when using private L2 caches, shared L2 caches with S-NUCA approach and shared L2 caches with D-NUCA approach with bank sets of four configured in columns. As expected, private caches achieve the lowest hop count, thus impacting in a reduced access latency. However, this is achieved by highly restricting the L2 cache capacity. In contrast, neither S-NUCA (due to its static nature) nor D-NUCA (due to its static bank set configuration) are able to achieve reduced hop counts. Our goal is to achieve similar results in hop distance to the private configuration but keeping the shared configuration of L2 banks.

To achieve this, we propose Runtime Home Mapping (RHM), a new dynamic approach where the LLC home bank is determined at runtime in hardware by the memory controller (MC). While in D-NUCA the mapping is



(a) Mapping on requestor's tile (b) Mapping on application's partition (c) Avoiding faulty L2 banks (d) Migration support (e) Replication support (f) Searching for home

Figure 3: Runtime Home Mapping. Different scenarios.

partially static, in RHM a block can be mapped to any L2 cache bank, potentially reducing the distance from L1 requestors to L2 banks. Figure 3.(a) shows the basic steps of the RHM mechanism comprising the request of a block to the MC, the selection of the home L2 bank while fetching the block from memory, and the forwarding of the block to the home L2 bank and to the requestor. In this case, the selected home is the local L2 bank of the requestor's tile.

RHM can be viewed as a D-NUCA configuration where a single bank set is configured comprising all the L2 banks. However, RHM differs from D-NUCA in the sense that it enables further optimization opportunities such as partitioning/virtualization, thread migration, and fault-tolerance. As an example, in Figure 3.(b) three applications are mapped on different resources of the chip. The MC, by using RHM, is able to map memory blocks belonging to an application to the L2 banks mapped for the application, thus guaranteeing network-level and memory-level partitioning. In this case the selected L2 bank is not at the tile's requestor. Also, Figure 3.(c) shows the case where one L2 bank has been disabled possibly due to some manufacturing defects.

In that situation, the MC, by using RHM, is able to filter out the failed bank and thus map blocks to functioning L2 banks. In Figure 3.(d) we can see the case where migration is enabled in RHM. In this case, one requestor solicits a copy of a private block. The RHM method triggers in this case a migration process. To further reduce hop distance from L1 requestors to L2 home banks, we enable in RHM replication of shared blocks. Figure 3.e shows the case where a home decides to launch a replica for a block.

All the previous examples hide two potential problems that need to be solved. The first one is the consistency of the coherence protocol. Multiple race conditions can arise when several processors trigger load and store requests on the same memory block. A careful design of the coherence protocol is needed. This is more difficult to achieve when enabling migration and replication of memory blocks. We provide a description of the supporting coherence protocol for the RHM method in all these special cases. The second problem is the efficiency of the coherence protocol. Indeed, the principal source of inefficiency is that L1 caches do not know which L2 bank is the home for a particular block. Since the *home* bank is not known a priori, a search must be performed each time an L1 miss occurs. This is shown in Figure 3.(f) where a request triggers a broadcast action in search of the home for a block. This problem affects the network infrastructure providing connectivity support to RHM. Thus, it affects the underlying on-chip network.

Conversely to previous approaches to home location, based on the use of limited-size tables, and therefore prone to costly overheads [2], we combine three different NoC mechanisms to optimize the search phase:

- An efficient *home* search method where a broadcast message is triggered to query the home. Then, a lightweight and simple dedicated control network is used to collect acknowledgments generated in the discovering process. This method and its hardware support enable a fast and efficient home search procedure. Also, basic signaling between tiles is supported by the control network.

- Parallel access of L2 tags. We highly couple tag array of L2s into the current NoC router design. At the same time the broadcast message enters the router, the tag array is accessed.

- A router mechanism aimed to reduce the broadcast messaging. On an L2 tag hit, the broadcast message is cancelled, thus reducing traffic by chopping broadcast branches.

In addition to these designs at NoC level, in this paper we develop improvements at the coherence protocol level. We provide different design alternatives as support for migration and replication. Also, we eliminate completely the directory structure by relying on the fast notification network. By a close design process between the NoC and the coherence protocol, RHM is able to effectively place the blocks near the core which is using them, reducing the average number of hops per request by more than 60% on average compared to static mapping, which leads to reductions of more than 35% in terms of cache latency and 28% in execution time. NoC and LLC energy consumption is also reduced by 40% and 23% on average, respectively.

The rest of the paper is organized as follows: in Section 2 we show the basic RHM mechanism and its NoC-level support. In Section 3 we present dynamic optimization to reduce the access latency to the blocks. In Section 4 we show how RHM can be merged with a broadcast-based coherence protocol. In Section 5 we show the evaluation results. In Section 6 we describe the related work. In Section 7 we discuss future work and conclusions.

2. Runtime Home Mapping

RHM aims to map blocks to L2 banks at runtime, in order to allocate them as close as possible to the requesting cores, preferably at the L2 bank of the requestor's tile. The mapping is performed by the MC each time it receives a request.

Figure 4 shows the global overview of the RHM protocol. For every processor access, the local L1 cache is accessed. In case of an L1 miss, a request is sent to the local L2 bank in the same tile. If the block is found on the L2 bank it means the L2 bank is the *home* for that particular block. Thus, coherence actions are triggered and at the end the block is delivered to the L1 cache. Coherence actions are described below.

On a miss on the local L2 bank, a broadcast is sent to all other L2 banks. When an L2 bank receives this broadcast request, it checks its tag array. In case of a hit, it sends the data back to the L1 requestor and a *hit* signal to the L2 bank that triggered the broadcast. All the L2 banks, upon receiving the broadcast message, trigger an acknowledgement (ACK) back to the L2 bank which issued the broadcast. When the requestor L2 bank receives the ACKs it checks the *hit* signal. If the *hit* signal has not been received it means

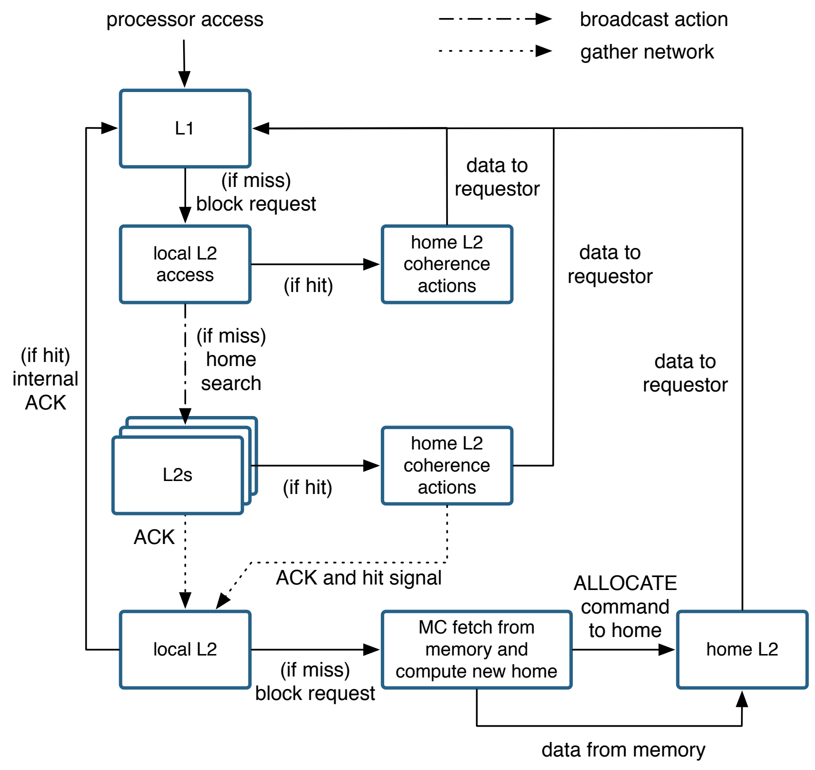


Figure 4: RHM global overview. From processor access to MC access.

the block is not cached on chip, so the bank sends a request to the memory controller (MC), which in turn fetches the block from main memory. If the *hit* signal has been received it means the block is on its way to the L1 requestor. Thus, an internal ACK signal is generated.

Upon receiving the request, the MC triggers the access to main memory to fetch the block. Meanwhile, it computes which L2 bank will act as home for the incoming block (the mapping policy and algorithm are described in Section 2.2). Once an L2 bank is chosen as the *home* for a particular block, the MC notifies the bank so it can start replacing a cache line, if needed, and allocate a new line while the MC is still waiting for the block. When the block is received at the MC, it is sent to the chosen *home* L2 bank, which in turn will send the block to the requestor L1 cache.

Upon a hit, either on the local L2 bank, or on a remote L2 bank, RHM follows the typical MESI protocol. Figures 5 and 6 shows the details of the

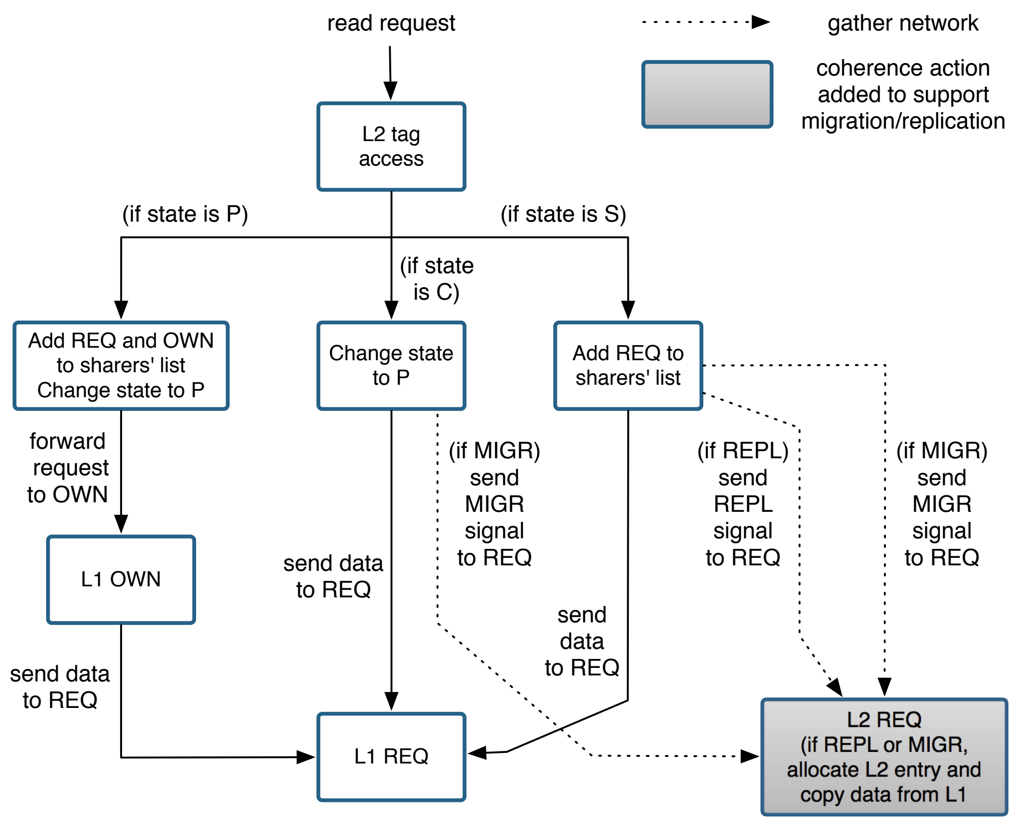

read request
gather network
coherence action added to support migration/replication
L2 tag access
(if state is P)
(if state is S)
(if state is C)
Add REQ and OWN to sharers' list Change state to P
Change state to P
Add REQ to sharers' list
forward request to OWN
L1 OWN
send data to REQ
(if MIGR) send MIGR signal to REQ
send data to REQ
(if REPL) send REPL signal to REQ
(if MIGR) send MIGR signal to REQ
send data to REQ
L1 REQ
L2 REQ (if REPL or MIGR, allocate L2 entry and copy data from L1

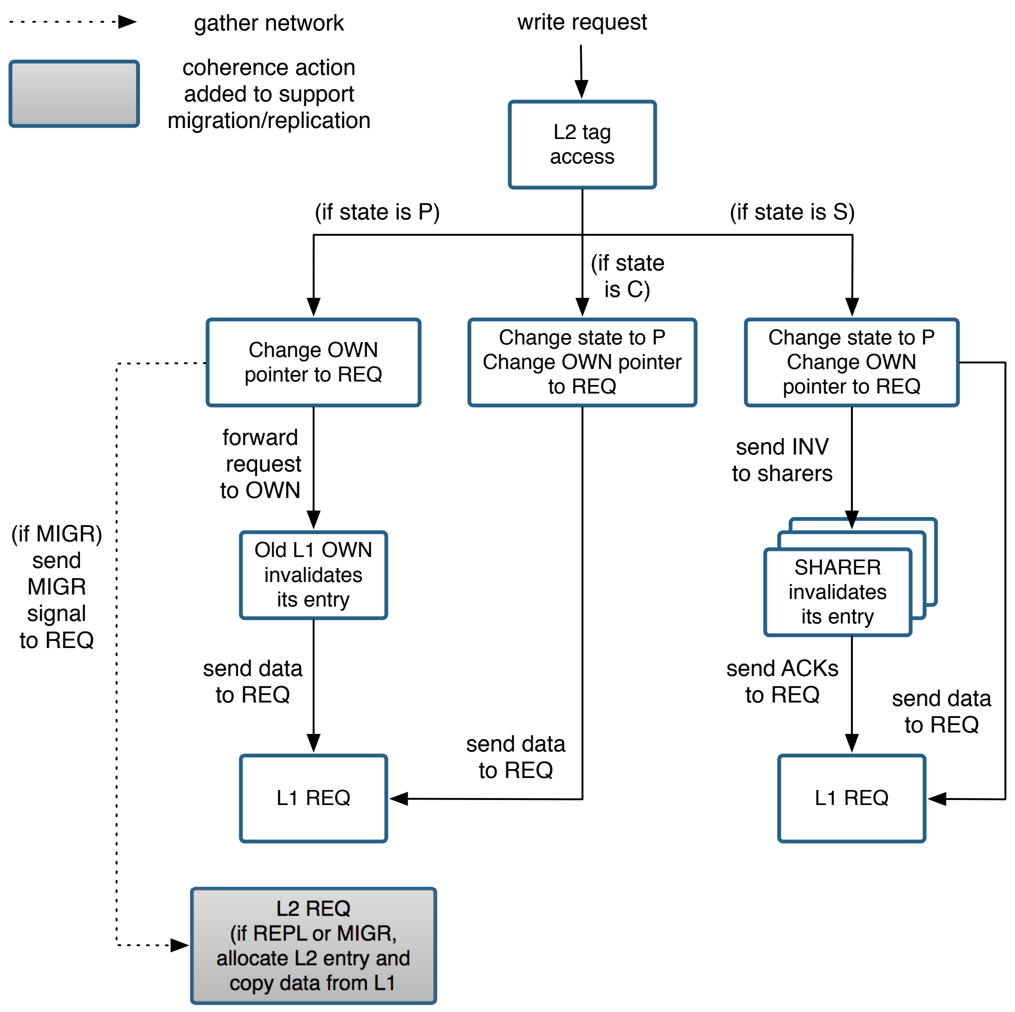
Figure 5: RHM coherence actions: load miss

protocol.[1]

For read accesses (Figure 5), depending on the current state of the block in L2 different actions are performed. If the block is in *private* state, then the request is forwarded to the *owner* L1 which in turn sends the block to the *requestor*. The L2 home updates the sharing list accordingly and the new state of the block (*shared*). If the block is in *cached* state, then it means there is only one copy of the block and lies in the home L2 bank. Thus, the block is sent to the requestor and the state is changed to *private*. If the block is in

[1]Notice that race conditions in coherence protocols need to be taken into account. The figure, however, for the sake of description, does not show the additional states and messaging needed to solve them. They have been carefully analyzed and solved.

shared state, then the requestor is added to the sharing list and the L2 bank sends the block to the requestor.



gather network
coherence action added to support migration/replication
write request
L2 tag access
(if state is P)
(if state is C)
(if state is S)
Change OWN pointer to REQ
Change state to P Change OWN pointer to REQ
Change state to P Change OWN pointer to REQ
forward request to OWN
send INV to sharers
(if MIGR) send MIGR signal to REQ
Old L1 OWN invalidates its entry
SHARER invalidates its entry
send data to REQ
send ACKs to REQ
send data to REQ
send data to REQ
L1 REQ
L1 REQ
L2 REQ (if REPL or MIGR, allocate L2 entry and copy data from L1

Figure 6: RHM coherence actions:store miss

Similar actions are taken when the access is a write operation (Figure 6). In this case, if the block is in *private* state, the request is forwarded to the *owner* which in turn sends the block to the requestor. The block is invalidated in the owner. Also, the owner pointer in the L2 bank is updated. If the block is in *cached* state (only one copy in the chip), then is simply forwarded to the requestor and the state and owner pointer are updated. If the state of the block is *shared* then a multicast message is sent to the sharers in order to invalidate their copy. Sharers in turn send an acknowledgment to the requestor. In addition, the L2 bank sends the block to the requestor and changes the state back to *private*.

In addition to the previous description, RHM supports migration and replication of blocks. In particular, blocks in *cached* or *shared* state for *read* requests and blocks in *private* state for *write* requests can migrate. On the other hand, blocks in *shared* state for *read* requests can be replicated over the chip. In those cases, and when a given threshold is reached (described in Section 3), the L2 bank triggers a signal (either MIGR or REPL) to the L2 bank that triggered the request (the one in the same tile of the L1 requestor). When the block is received by the L1 requestor, and if the signal has been received, the block is also copied on the L2 bank. Further descriptions are given in Section 3.

From Figures 4 and 5 we can deduce the network latency when accessing blocks. If the block is mapped in the L2 bank in the local tile then no access to the network is made. This will be the frequent case as the MC will try to map most of the blocks to the requestor's tile. However, when the block is mapped in an L2 bank on a different tile then different accesses to the network will be made. First, a broadcast to find the L2 bank. Then, all the L2 banks sending an ACK signal to the requestor. Also, the L2 home probably will send new messages to other nodes (from Figure 5) and finally sending a hit signal and the block to the requestor. In case the block is not mapped on cache then the MC is accessed and a new L2 home is computed. This means more messages through the NoC.

The L2 home search policy just described above has high network resources demand: every time a request misses in the local L2 bank, a broadcast is issued. Also, all other banks must answer to the broadcast with an ACK or with the data. The first action can be attenuated by implementing a tree-based broadcast mechanism within the NoC: a broadcast is sent as a single message, that replicates at switches to reach every L2 bank. This reduces NoC traffic and eliminates the serialization of multiple copies of the same request (one per destination). ACKs however still represent a problem: they are indeed sent roughly at the same time and will probably serialize in the network and, most important, all of them must reach the same L2 cache bank (i.e., the one that initiated the broadcast). In addition, different signals (*hit*, *MIGR*, and *REPL*) are sent from the L2 bank to the L2 requestor. These signals need to reach the requestor at the same time ACK signals do.

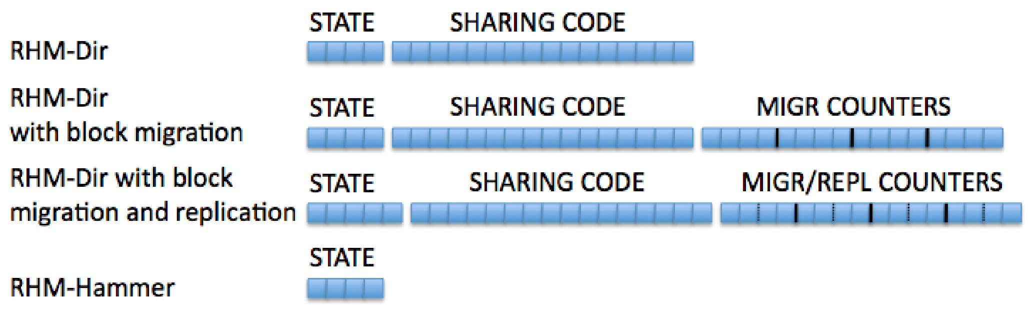

STATE
SHARING CODE
RHM-Dir
RHM-Dir with block migration
MIGR COUNTERS
RHM-Dir with block migration and replication
MIGR/REPL COUNTERS
RHM-Hammer

Figure 7: Control info required for each version of RHM.

To reduce the network impact of all those signaling commands, we develop the Gather Control Network (GCN) described in the next section. This network optimization is vital to make the RHM method become effective and efficient.

Figure 7 shows the control info required for the coherence protocol. Different versions of RHM will be described in next sections: a version which uses a directory-based coherence protocol (RHM-Dir), a version using a broadcast-based coherence protocol (RHM-Hammer) and two optimizations where blocks can migrate from a bank to another and be replicated in different banks; although we implemented these optimizations on the directory-based version, they are orthogonal to the coherence protocols. For each cache line, the L2 bank will keep the state of the block. Four bits are needed to encode all the possible states (the L2 state machine has less than 16 states) except for the version with block replication and migration support, which needs five bits (up to 27 states are used to correctly manage the possible race conditions). All the versions, except RHM-Hammer need the sharing code, to reflect the list of sharers of a block. Also, migration and replication support need counters to trigger the migration and replication processes. As can be deduced, the only field that grows with system size is the sharing code while the size of the counters only depends on the thresholds which are chosen to trigger the migration and replication of a block.

2.1. Gather Control Network

The Gather Control Network (GCN), can be logically seen as 16 one-bit wide subnetworks, one per tile. Each subnetwork is a tree of AND gates, connecting the destination tile (the root) with all other tiles (located at the

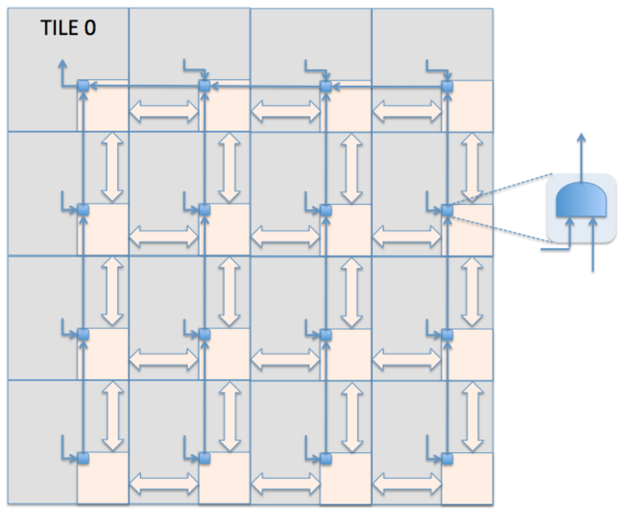
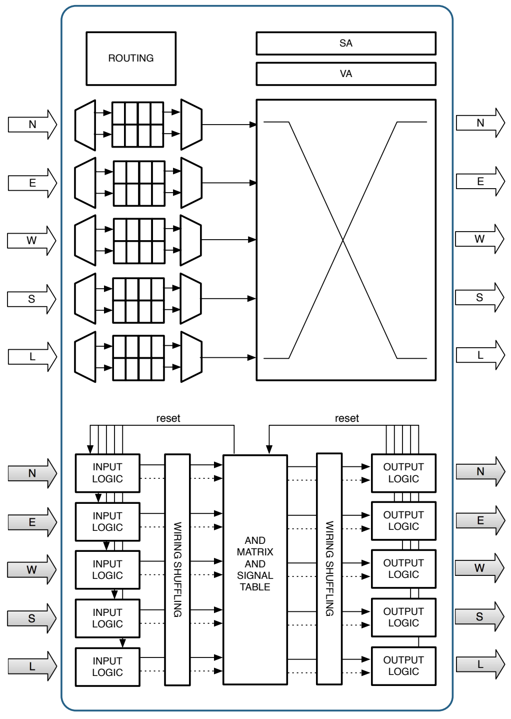


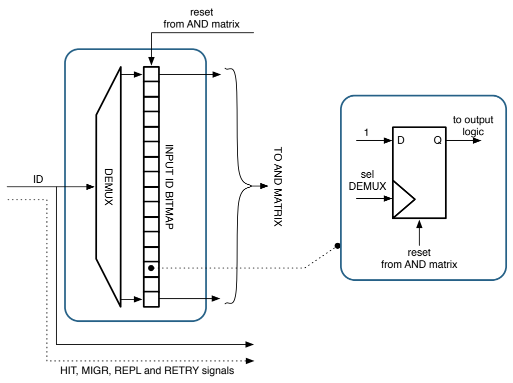
Figure 8: Gather control network for tile 0.

leaves of the AND tree). Figure 8 shows the logical view of one subnetwork with root in tile 0. A one-bit subnetwork (darker arrows) is added to the regular NoC (bidirectional arrows). If a request misses in the L1 and L2 caches of tile 0 (L1-0 and L2-0 from now on), L2-0 broadcasts the request to all other L2 banks through the regular NoC. When an L2 bank receives this request, it triggers the output signal of the GCN for tile 0. Once all L2 banks trigger their output signals, the output of the AND tree will notify the L2-0 and thus will act as a global ACK message. Two different implementations of the GCN were used in our previous works to speedup the ACKs sent by L1 caches to the L1 requestor in directory-based and broadcast-based coherence protocols [3, 5, 4]. In this work, we provide a detailed description of the GCN logic and extend its use to collect acknowledgements generated by and destined to different levels of the cache hierarchy and to deliver simple control information together with the global ACK.

By sending ACKs through the GCN we achieve two major benefits. The first one is the reduction in network traffic and the associated power consumption saved. The second one is the reduction in message latency, as ACKs now are sent as signals rather than messages, thus avoiding routing, flow control, and arbitration in the NoC at each hop and message serialization at the destination node. Indeed, the GCN takes profit from the fact acknowledgments can be reduced to only one notification event with a single



(a) Router with RHM



(b) RHM input logic

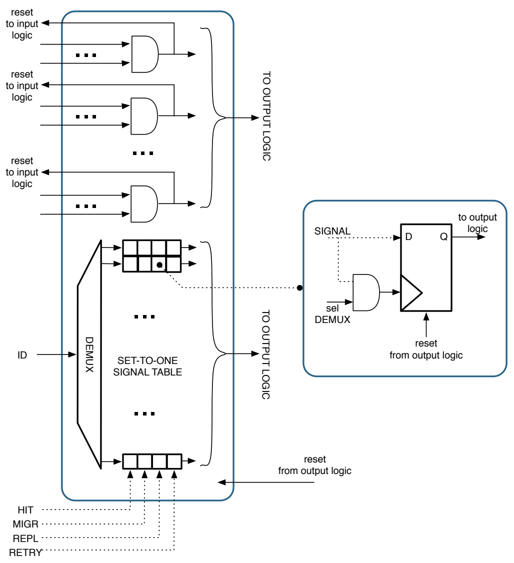
Figure 9: Router design with GCN network and RHM input logic.

signal as the receiver is waiting for a known event for a known block.

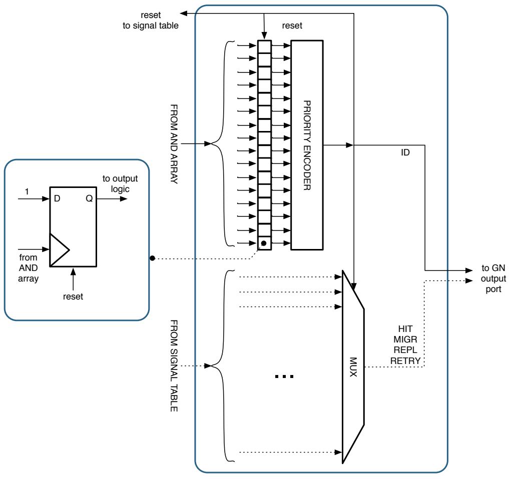
Although the GCN can be implemented as a very specific combinational block, we choose a sequential implementation with a more general behavior. Instead of wiring different AND trees we develop a network capable of collecting identifiers with associated control signals. This makes the GCN scalable with system size and adds more functionality. We will refer to the number of supported identifiers by n.

The GCN is distributed over the routers with the same logic blocks but instantiated differently depending on the position of the router on the 2D mesh. Figure 9.(a) shows a typical NoC 5-port switch with the added logic for the GCN. The bottom part is the new logic. In detail, the GCN consists of three main blocks: the input logic, the central logic, and the output logic.

The input logic (Figure 9.(b)) is located on every input port and has an associated input channel. The input logic receives IDs and three control signals: *hit*, *MIGR*, and *REPL*. The input logic has one bit (implemented



(a) RHM central logic.



(b) RHM output logic

Figure 10: RHM central and output logic.

as a flip-flop) per identifier, building the *input ID bitmap*. Thus, the bitmap register is of size n. Whenever an ID is received the associated bit is set. Also, the incoming ID and the control signals are forwarded to the central logic.

The central logic (Figure 10.(a)) is in charge of two actions. First, it has to AND the input identifiers with the same value, coming from different input ports. To achieve this, the central logic is made of n AND gates. The signals coming from the input ports are reorganized appropriately at the previous *wiring shuffling* stage. Wires from IDs with the same values are put close each other. Second, the central logic has a signal table ($n \times 3$ matrix) combining all the control signals coming from the input ports. Whenever an input signal comes with value set to one, it is registered. Notice that input signals coming with a value set to zero do not reset the value in the signal table. We can view this table as an OR operation of input signals with the same IDs.

Finally, each output port has a output logic (Figure 10.(b)). The main function of this logic is to forward IDs that have been combined by the central logic. To do this, IDs are stored in a bit vector (*output ID bitmap*) and encoded when forwarded. A priority encoder is used. The output of the

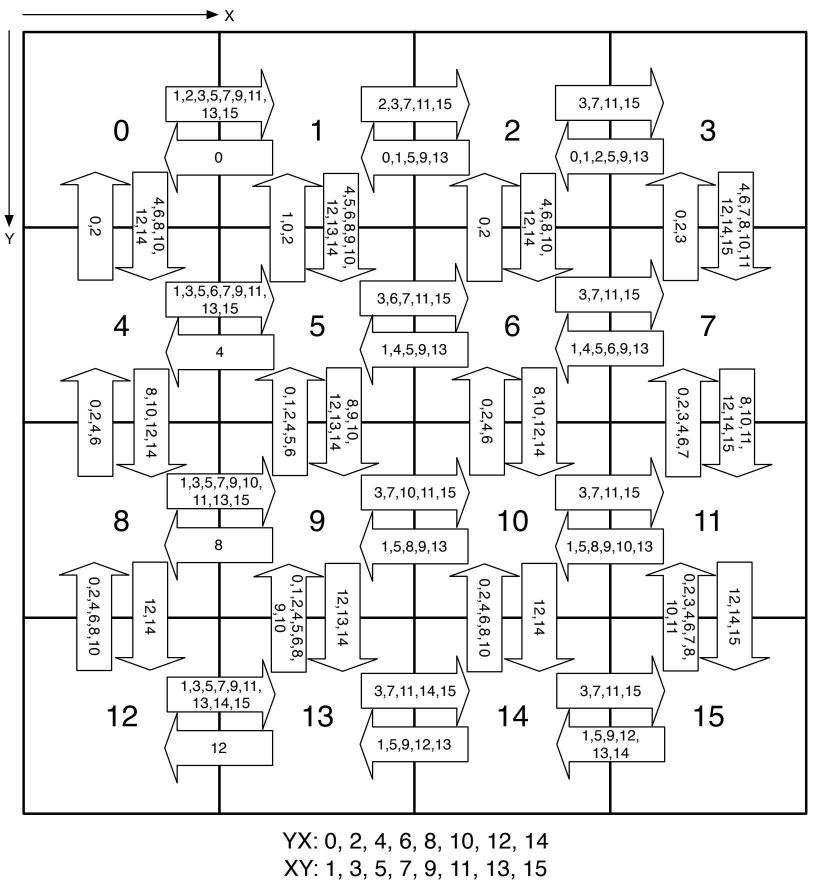


Figure 11: GCN Mapping of IDs.

encoder also selects the control signals stored in the signal table, thus the output port forwards the ID with its combined control signals.

As we can see, IDs are stored both at the input and at the output ports, while signals are centralized in the central logic. Those IDs and signals need to be reset every time the ID is collected and forwarded through the output port. To do this, reset signals are triggered from the central logic to the input logic (to reset IDs) and from the output logic to the central logic (to reset the signals). The IDs at the output logic are reset whenever the ID is forwarded through the port.

Notice that the size of registers at each input port will vary from switch to switch in a 2D mesh. This depends on the mapping strategy used to

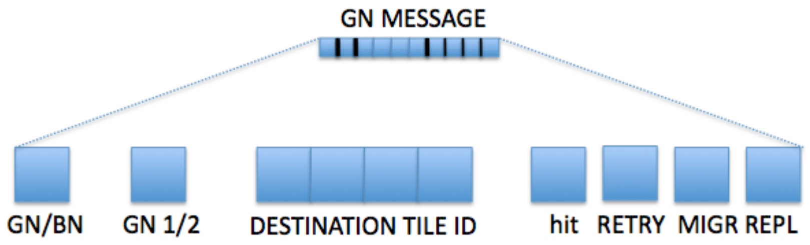


Figure 12: GCN message format.

collect IDs (e.g. they can be collected following X-Y patterns or Y-X patterns). In particular, we follow the mapping strategy shown in Figure 11. IDs associated with tiles with odd identifiers follow XY routing whereas IDs associated with tiles with even identifiers follow YX routing. This creates a balanced distribution and reduces the number of IDs per output port. In particular, the maximum number of IDs at an output port (and input port) is 9 (e.g. north output port at tile 13). In contrast, there are output ports forwarding only one ID (e.g. west output port at tile 1). Therefore, the size of the registers at the input and output logic will vary between 0 and 9. By collecting all the output IDs in a tile through all the output ports, we can see that the sum is always 15. Indeed, each tile will send one single copy of each ID except for the local one. The figure does not show the local ports. In this case, the injection port will have as many IDs as possible destinations (N-1) and one single ID at the ejection port (the one associated with the local node). Also, as the central logic will be shared by all the input ports, the number of AND gates and entries in the signal table will not vary and will be set to n. In the evaluation section we will show overhead costs of the GCN implementation.

In principle, IDs through the GCN will be used to communicate between L2 banks. However, this can be extended to get more functionality. As an example, with one extra bit for IDs (duplicating the number of IDs) we can build two GCNs (using the same logic) but having two logical GCNs, one for gathering ACKs for (notifying) L2 banks and one for gathering ACKs for (notifying) L1 caches. Also, out-of-order processors may trigger different request to the network, thus requiring different collecting processes through the GCN. This can be supported by extending the number of IDs. Figure 12 shows the GCN message fields we assume in this paper. The target system is a CMP system with 16 tiles. Thus, four bits are used to address the target tile. With one extra bit we indicate whether the target component is the L2 bank or the L1 cache. In-order processors are assumed. With the ID we also send the three control signals: *hit*, *REPL*, and *MIGR*. An additional bit, *RETRY*, is used to manage concurrent request through a fixed priority policy.

2.2. Mapping Algorithm

Each time the MC receives a request, a mapping algorithm chooses the *home* bank for the requested block. The *home* is chosen depending on the requestor's tile and current L2 banks utilization. The MC takes statistics about cache utilization, which are stored in a table (*alloc* table) with $N \times M$ entries, where N is the number of L2 cache banks and M the number of L2 sets. Each entry contains the number of allocations performed in set m of L2 bank n. If the associativity of L2 sets is Z, the table has to store at minimum $N \times M \times \log_2 Z$ bits. However, the table will double the bits in each entry. For a 16×16 tile system with 16-way 256KB bank sets, the minimum memory requirements for this single table is 2KB ($m = 16$, $M = 256$, $Z = 16$). With the increased size, the table will grow to 4KB (to allow 256 allocations per set). The pseudocode shown in Figure 13 describes the simple algorithm we implemented.

If there is room in the set in the local L2 bank (r tile), then the *home* is the local tile of the requestor. Otherwise, the algorithm scans the neighbor banks in distance order (first *for* loop). This search is performed until the threshold $MaxHops$ is reached, which can be equal to the physical threshold forced by the system size (number of hops from the requestor to the furthest tile) or lower.

If all the L2 banks are full (*alloc* higher than *num_ways*), the algorithm tries to balance the number of allocations (thus, replacements) in all banks (second *for* loop). A threshold (*UtilThr*) is used. If the difference between the number of allocations in the local tile's bank and a neighbor bank is higher than the threshold, then the neighbor bank is selected as the *home* bank.

If all the banks are balanced, then the block is mapped to the requestor's tile. Notice that this does not imply that RHM defaults to private L2 caches. With private caches all the data accessed by a core must be present in the L2 bank of the same tile, while in RHM this does not apply. For instance, a shared block will be replicated in all L2 caches if they are private, while in RHM it will be present only in the *home* tile. The proposed policy defaults

```
int function allocate(int r, address a) {
 banklist n; bank b; set s; bank h;

 s = get_set(a);
 if(alloc[r,s]<num_ways) {alloc[r, s]++; return r;}

 for(int h = 1; h <= MaxHops; h++){
   n = BanksReachable(r, h);
   for (int i = 0; i < size(n); i++){
     b = SelectBankClockWise(n, i);
     if (alloc[b,s]<num_ways) {alloc[b,s]++; return b;}
   }
 }

 for(int h = 1; h <= MaxHops; h++){
   n = BanksReachable(r, h);
   for (int i = 0; i < size(n); i++){
     b = SelectBankClockWise(n, i);
     if (alloc[r,s] - alloc[b,s] > UtilThr) {alloc[b,s]++; return b;}
   }
 }

 alloc[r, s]++; return r;
}
```

Figure 13: Mapping algorithm performed by the MC.

to private caches only if all L2 banks are full, each core is requesting private blocks and all banks are uniformly used. In this case, very unlikely in a parallel application, all blocks are allocated in the requestor tile, which indeed is the best choice since it minimizes the data access latency.

3. Optimizations to RHM

In the previous sections we have detailed the RHM method and the GCN network. Now, we focus our attention on additional optimizations to RHM in order to make the final solution more efficient. On the one hand, we provide migration and replication of blocks (to reduce the access latency to L2 home banks). On the other hand, we provide faster L2 bank access (through a parallel tag access approach) and remove the directory structure.

3.1. Block Migration

RHM reduces the access latency by mapping blocks closer to requestors. Once the block is on-chip, however, the core which actually use it may change at runtime. To reduce the access latency in these cases, the initial placement of the block can be adjusted at runtime allowing blocks to migrate to a new L2 bank. Notice that a sort of migration mechanism is implicit in RHM, since each time a block is replaced from an L2 bank and then requested again it may be mapped to another L2 bank by the MC. However, this may not always be effective. If the block is never replaced by the L2, it stays in the bank.

If the initial *home* allocation performed by the MC results sub-optimal, block migration can be enabled to further reduce the number of hops between an L1 cache and the L2 bank where the block is mapped. We propose a migration scheme similar to the one used in D-NUCA but without the constraint of being limited within a bank set: in RHM a block is allowed to migrate to any L2 bank. However, since the migration process introduces an overhead in terms of traffic and energy, it should be performed only if it actually leads to a benefit in terms of miss latency reduction.

Solutions in D-NUCA reduce unnecessary migrations and avoid the ping-pong effect by using a saturating counter for each direction to which a block can move. A counter is updated each time a request comes from a node located in the counter's direction; when the counter saturates, the migration process towards that direction is triggered. In our case a block may migrate in any direction, so four counters are needed, one per direction. Each time a request is received from a tile, the counters are updated adding the distance in hops from the requestor. When a counter is incremented, the one in the opposite direction is decremented. When a counter saturates, it starts the migration process: the block migrates to the L2 bank located in the same tile of the L1 which sent the request that triggered the migration.

Notice that migration occurs between two L2 banks, one requesting the block and the other being the L2 home bank. The L2 home bank notifies the L2 requestor through the GCN by activating the *MIGR* signal (it also sets the *hit* signal). This signal will arrive to the L2 requestor at the same time all ACKs are collected. Upon receiving this signal, the requestor L2 performs an internal copy of the block from the requestor L1 cache (in its tile) once the block is received. The former L2 home bank deallocates the block. See Figure 5.

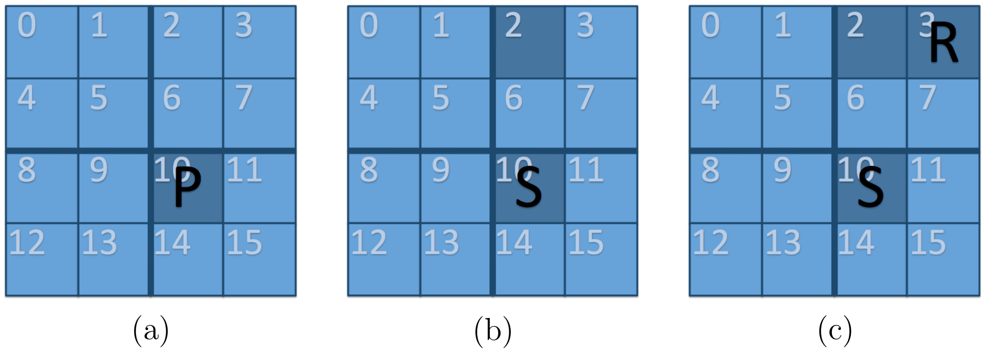


Figure 14: Block replication.

3.2. Block Replication

Block migration effectively reduces the LLC access latency of private blocks. Each time the accessing core changes, the block potentially migrates to the L2 bank in its tile. For shared blocks, however, this may not be the optimal solution. First of all, the block may be mapped close to one of the sharers but away from the others. Second, the block may ping-pong between two L2 banks when different sharers continuously request it. Since a block cannot be modified while it is shared, it is safe to replicate it in more L2 banks to reduce the access latency of the sharers.

The minimal access latency is achieved when each sharer has a replica of the block in its local L2 bank. This case, however, may end up suboptimal as the on-chip cache capacity will be highly reduced due to the high number of replicated blocks. To reduce the number of replicas, we partition the chip in a reduced number of replication regions. At most one copy of the block can be present in the L2 banks of each region. To avoid unnecessary replicas, we use a saturating counter for each region.

Figure 14 shows how the replication process works. The CMP is divided in four replication regions, each one including four tiles, marked with a thicker line. Initially, there is only a private copy of the block in the L2 of tile 10 (L2-10), which is the tile the MC mapped that block to (or the destination tile of a previous migration process). In Figure 14.(b) L1-2 requests the block with read permission, so the block is shared by L1-10 and L1-2. If L1-2 or any other L1 of the same replication region requests the block several times until the counter for its region saturates, the replication process begins: L2-10 sends the block both to L1-3, which requested it, and to L2-3, where it is saved as a replica (Figure 14.(c)). Notice that L2-10 remains the home bank for the block: it keeps updating the directory and managing all requests, except for the read requests issued by L1s located in the same region of L2-3. For those requests, L2-3 is in charge of providing the data to the requestors.

When any node issues a write request for a replicated block, the home node sends an invalidation message to all replicas and to the L1 sharers which are in regions where the block is not replicated. Each replica, at the reception of the invalidation message, invalidates the sharers in its region. The directory is extended with extra bits to track the replicas at the home node. As shown in Figure 7, the migration counters are used to control the replication process also, since a block migrates or replicates depending on its state but can't do both things being in the same state. We assumed to divide the chip in four replication regions; for each region, two bits are used for the counter and the other two are used to codify the ID of the bank holding a replica within each region (banks are numbered 0 to 3).

When the home bank has to replace a replicated block, it is not necessary to invalidate the sharers. One of the L2 banks with a replica of the block is chosen to become the new home for the block, so is notified with a message which includes the directory information for that block. On the other hand, if an L2 bank has to replace a replica, it only has to notify the home bank. The home bank will remove its ID from the list of replicas and manage the requests originated in that region.

4. RHM and Broadcast-based Coherence Protocols

In previous sections we assume a directory-based coherence protocol, with a data structure (the directory) associated to each L2 cache line to store the list of sharers. The L2 cache uses this information when it has to communicate with L1 caches to manage a request; two typical cases are the invalidation of the sharers of a shared block upon a write request and the forwarding of read and write requests to the L1 which owns the modified copy of a private block. The directory introduces an area overhead, since part of the on chip memory has to be used to store the directory entries. The size of the directory grows linearly with system size, making full-map directories unfit for systems with more than a hundred of cores (the size of the directory would be comparable to the line size).

Broadcast-based protocols completely eliminate the list of sharers from the directory information, thus eliminating its extra area and power requirements. One example is the Hammer protocol [27, 28], used by AMD in its systems based on the Opteron processor family. The drawback of broadcast-based protocols is the amount of traffic they generate. While in directory-based protocols the L2 cache always knows exactly to which node it has to communicate, thus injecting in the NoC the minimum amount of traffic, in broadcast-based protocols a broadcast must be sent to all L1 caches each time the L2 home bank has to invalidate the sharers of a block or forward a request to the L1 which has a private copy of the block. In addition, each node must answer the broadcast message by sending an acknowledgement message (ACK) to the requestor, and, if the node is also the owner of a private block, the requested data block. Due to these additional communication, broadcast-based protocols have usually worse performance than directory-based.

However, now that we have the GCN we can conceive an RHM implementation with sharing-less information. By using the GCN we can save the area and power overhead of the directory. Also, we can reuse the broadcast phase of RHM when searching the home L2 bank. Coherence actions of the hammer protocol will be embedded in the broadcast search method of RHM. To achieve this goal, we extend the Gather Control Network and implement a Broadcast Control Network to send a fast notification from any node to all other nodes.

4.1. Broadcast Control Network

As described in Section 8, the Gather Control Network can be logically described as a set of AND trees, one per tile. Each tree has its root in the destination tile and all other tiles are at the leaves. The implementation we actually use combines IDs of the destination node, following the structure of the AND tree. By crossing the tree in the opposite direction, it is possible to broadcast the ID of a node to all other nodes.

Figure 15 shows the implementation details of the BCN. It reuses the links of the GCN. To differentiate IDs from the GCN an additional bit is used. Whenever the BCN is used the bit is set. This bit drives a demultiplexer thus the incoming bit at the input port is forwarded either to the GCN input logic or to the BCN logic. The BCN simply disseminates the ID through some output ports, following the XY pattern. At each output port, a register with as many bits as IDs is implemented. When an ID is received at an output port, the associated bit is set. The register is then inputed to a priority encoder and the output link is arbitrated between the GCN output logic and the BCN output logic.

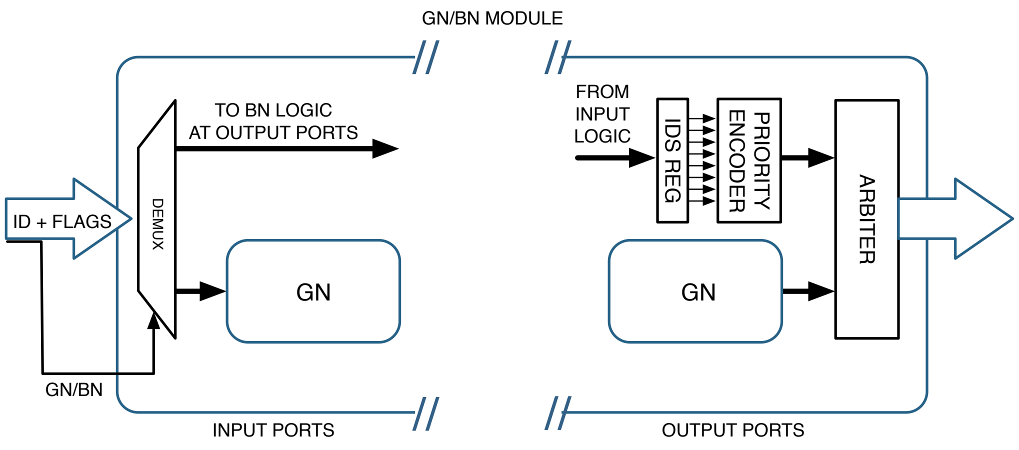


Figure 15: BCN implementation.

4.2. Merging Hammer protocol and RHM

The basic coherence actions when Hammer protocol is combined with RHM (RHM-Hammer) are shown in Figures 16, 17 and 18. Each time the L2 home bank needs to communicate with an L1 cache to satisfy a request, it broadcasts the request to all L1 caches, which answer by sending an ACK back to the L1 requestor. The GCN we considered so far must then be expanded with an additional GCN: one GCN (GCN-1) will be used to collect the ACKs sent by the L2 banks during the home bank search phase, and a second GCN (GCN-2) will be used to collect the ACKs sent by the L1 caches back to the L1 requestor once the home bank has been found and it is managing the request. By adding one bit to the ID identifier we easily provide support to the GCN-2. Notice that GCN-1, GCN-2 and BCN logic blocks at each router share the same physical links. In Section 5 we will provide evaluation results of the number of conflicts inside a GCN module.

In case the requested block is not cached on chip, RHM-Hammer behaves like the Directory case (RHM-Directory): the home search phase will miss in all L2 banks, and the L2 of the requestor's tile will send a request to the MC. It fetches the block from main memory and executes the home mapping algorithm. Once the block is received from main memory, a data message is sent to the chosen L2 home, which will provide the block to the requestor.

If the requested block is cached on chip, RHM-Hammer exploits the BCN and the GCN to speedup the coherence actions. In the case of a read request

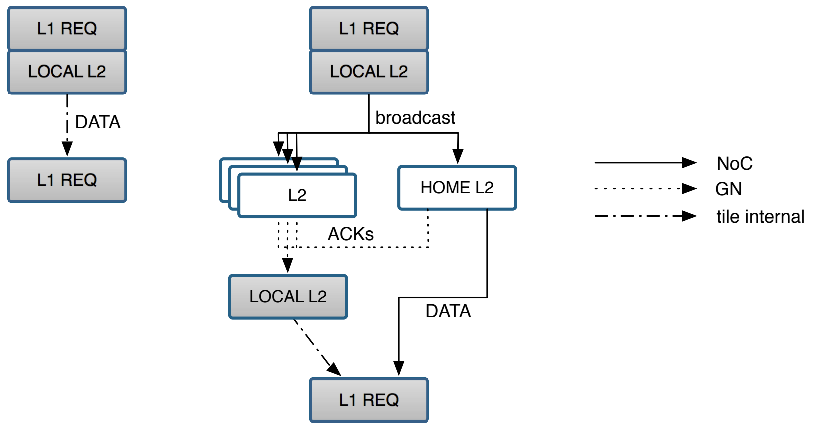


Figure 16: Read request for a shared block in case of hit (left) or miss (right) in the local L2 bank.

for a shared block (Figure 16) RHM-Hammer behaves like RHM-Directory: if the request hits in the local tile, the local L2 directly sends the data to the requestor. If, however, the request misses in the local tile, a broadcast is sent to all L2 banks and the block will be provided by the home bank.

In all other cases RHM-Hammer uses the GCN and the BCN to manage the request. Figure 17 shows the case when a request hits in the local L2 bank. In case of read or write request for a private block (Figure 17.a), the local L2 bank sends a broadcast through the NoC to all other L1 caches, which answer acknowledging the broadcast through the GCN. The L1 which owns the private copy is in charge of sending the block to the requestor. In case of a write request on a shared block (Figure 17.b), the broadcast is used to invalidate the sharers and the data is provided by the local L2 bank. Notice that RHM-Directory can not use the GCN to collect the acknowledgements since the invalidation message is only sent to the sharers, so some nodes at leaves of the requestor's AND tree would not send an ACK. The acknowledgement phase is thus faster in RHM-Hammer.

Figure 18 shows how the previous cases are managed when the request misses in the local bank. In case of read or write request for a private block (Figure 18.a), the local bank starts the home search phase; during this phase, all tiles, upon receiving the request, save in a private table, at the entry of the requestor, the block address and the request type. This table will have as many entries as IDs are supported by the BCN.

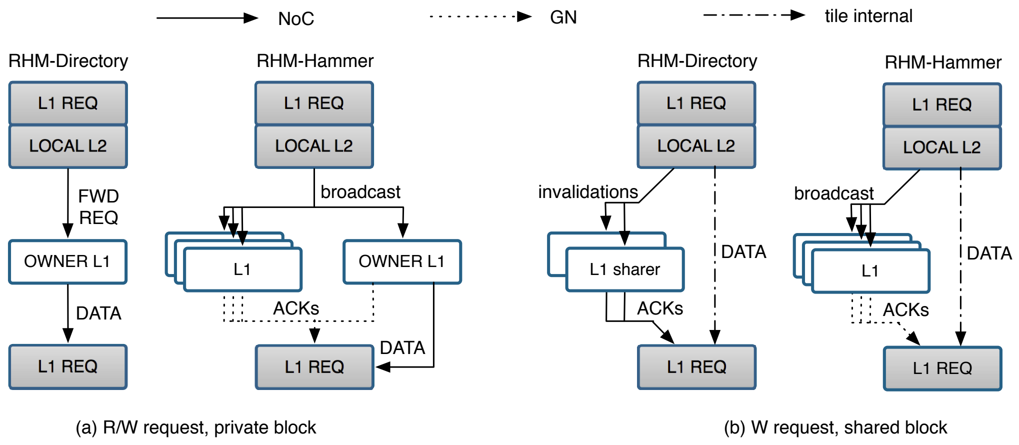


Figure 17: Request hit in the local L2 bank.

When the request is received by the home bank, a broadcast is sent through the BCN to all the L1 caches: the ID of the requestor is broadcasted through the BCN. Once an L1 cache receives the requestor ID, it checks the table to know which address and access type are associated to that ID and performs the actions established by the coherence protocol. If the ID is received at the owner L1, it sends the block to the requestor and invalidates its cache line (in case of write request) or changes the line state from private to shared (in case of read request). In case of a write request on a shared block (Figure18.b), the home bank must invalidate the sharers. Again, this is done by broadcasting the request through the BCN and all ACKs are collected through the GCN. Notice that in RHM-Directory both the invalidation messages and the acknowledgements are sent through the regular NoC.

Although RHM-Hammer seems to generate much more traffic than RHM-Directory, all the additional traffic is actually sent through the GCN and the BCN, which is faster and has much lower energy requirements than the regular NoC. Furthermore, since in case of miss in the local L2 bank the communication between the home bank and the L1s is done through the BCN, it is faster than in the case of RHM-Directory, where the regular NoC is used to forward the request to the owner or to invalidate the sharers of a block.

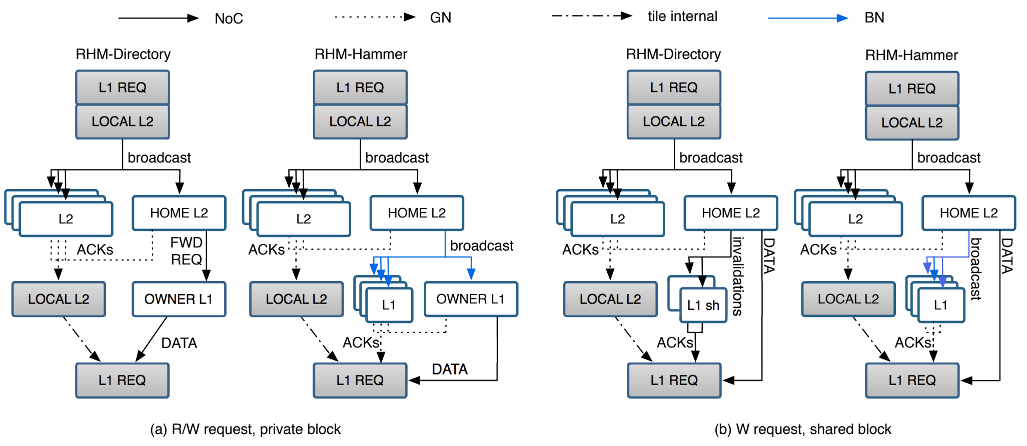


Figure 18: Request miss in the local L2 bank.

5. Performance Evaluation

In this section we evaluate RHM and compare it with other proposed NUCA configurations. In the baseline (S-NUCA) blocks are statically mapped to L2 banks using the less significant bits of the block address. In D-NUCA, blocks are statically mapped to a bank-set depending on their addresses. The matrix of L2 banks is divided in bank-sets, one per column of tiles. Blocks are inserted in the L2 bank located in the same row of the requestor and then can migrate within the bank-set, one hop each time a migration is triggered. A third configuration uses private LLCs. Finally, we consider an S-NUCA configuration in which the blocks are mapped to the L2 banks using a first touch policy [29]. The first time a block is requested, the memory page containing that block is mapped to the L2 bank in the requestor's tile. We assume 4KB as the page size.

These configurations assume a directory-based coherence protocol, and are compared against four configurations of RHM. The first one (RHM) uses a directory-based coherence protocol. In RHM M we enable private and shared block migration. In RHM M+R we enable both migration and shared block replication. In RHM HAMMER we use the Hammer coherence protocol.

The cache coherence protocol for each configuration, the NoC with broadcast support and the GCN/BCN networks have been implemented and simulated using our flit-level cycle-accurate network and cache hierarchy simulator. Each protocol has been tested for deadlocks and race conditions with all

Routing	XY	Coherence protocol	Directory / Hammer
Flow control	credits	L1 cache size	16 + 16 kB (I + D)
Flit size	8 byte	L1 tag latency	1 cycle
Switch model	4-stage pipelined	L1 data latency	2 cycles
Switching	virtual cut-through	L2 bank size	256 kB
Buffer size:	9 flit deep	L2 tag latency	1 cycle
Virtual channels:	4	L2 data latency	4 cycles
GCN/BCN delay	1 cycle/hop	Cache block size	64 B

Table 1: Network and cache parameters.

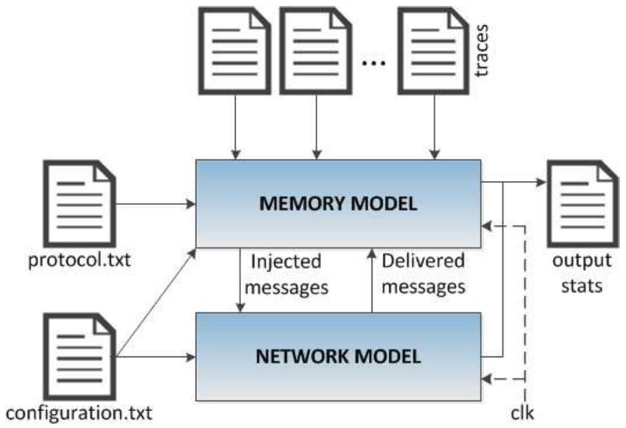


Figure 19: Organization of our cache hierarchy and NoC simulator.

the applications used in the simulation phase. Figure 19 shows the structure of our simulator: L1 cache accesses, which may be read from a trace file or generated by an external simulator into which our tool is embedded, are sent to the memory model. This module performs a cycle-by-cycle simulation of the cache hierarchy and the coherence protocol; if caches located at different levels of the cache hierarchy or located at different tiles have to communicate, a message is injected into the network model, which simulates cycle-by-cycle the advance of the flits through the NoC. When a message reaches the switch connected to the destination node (an L1 cache, an L2 cache or the memory controller) it is delivered to the memory model, and the destination node will evolve as established by the coherence protocol.

To evaluate our proposal with real applications, we captured the memory accesses of Graphite [7] and Sniper's [8] simulated cores and used our tool for cache hierarchy and NoC timing. Different applications of the SPLASH-2 and

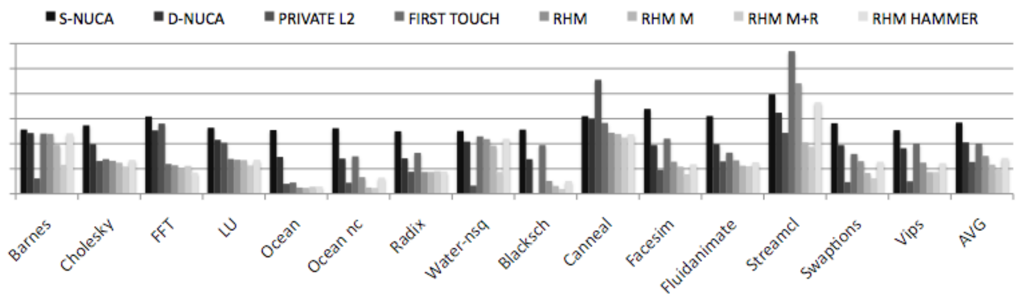


Figure 20: Avg hop distance between L1 requestors and the tile where the data is found.

PARSEC benchmark suites have been run on the 16-core system considered throughout this paper. Network and cache parameters are shown in Table 1. Cache latencies have been obtained using Cacti [9]. One memory controller is placed at the top left corner of the chip. For the sake of fairness, ACKs in D-NUCA are modeled with 2-cycle latencies (as when using the GCN). Migration and replication thresholds have been chosen running different sets of simulations with different threshold values and picking the value with the best average performance.

Figure 20 shows the average hop distance from the requestor to the *home* tile. For S-NUCA, the block is found on average at a distance of 2.85 hops. This distance is roughly the same for most applications as blocks are uniformly distributed among the L2 banks. With other configurations, however, since blocks are dynamically mapped and/or moved from a bank to another, the distance is quite variable depending on the application. For Barnes, dynamic techniques are not so effective, and the average value is always higher than 2 hops. The exception is for RHM M+R. This is due to the high sharing of blocks between cores. Thus, RHM M+R adapts to this type of sharing. For other applications, e.g. Ocean, those techniques achieve a large reduction in the average number of hops.

On average, RHM locates the data closer to the requestor than the other configurations, and this distance is further reduced if block migration is enabled. Indeed RHM M and RHM M+R achieve a locality close to that of PRIVATE L2.

Figure 21 shows the percentage of requests which hit in the L2 bank located in the same tile of the requestor. Again, results when using S-NUCA do not depend on the application due to the uniform mapping of the blocks, and this percentage is quite low (6% on average). This percentage increased

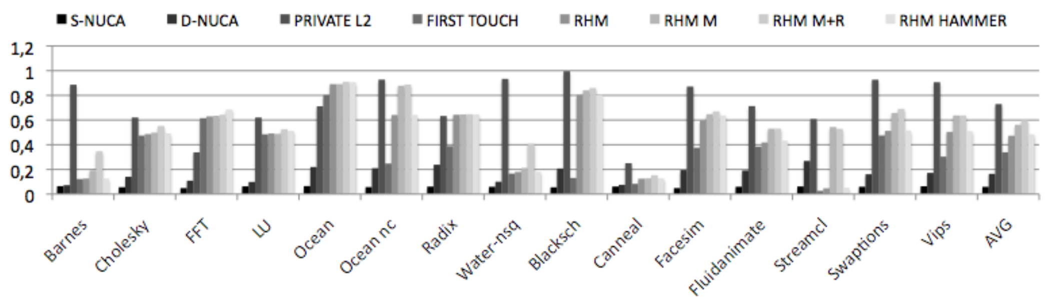


Figure 21: Percentage of hits in the L2 bank located in the tile's requestor.

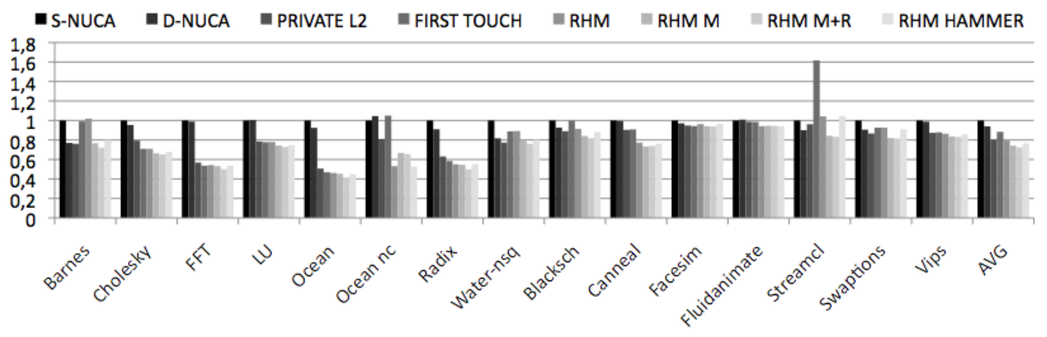


Figure 22: Execution time normalized to the S-NUCA case.

to 16% for D-NUCA, but is still much lower when compared to First Touch (33%), RHM (49%), RHM HAMMER (49%), RHM M (56%), RHM M+R (60%) and Private L2 (72%). Thus, RHM is the dynamic method which achieves the highest percentage of hits in the local bank.

However, a high locality alone is not enough to improve the performance of a system with a banked, distributed LLC: on-chip cache capacity is also a crucial factor, since the cost of an off-chip access to main memory is much higher than the cost of accessing a cached block located in a distant zone of the chip. This motivates the common design choice of shared LLC banks, since this configuration provides higher cache capacity than private LLCs, reducing LLC misses in case the application's working set does not fit in a single LLC bank.

Figures 22 shows the normalized execution time with all the configurations. We can observe how execution time is largely reduced with an average factor of 12% when using FIRST TOUCH and ranging between 20% and 28% when using the various configurations of RHM. RHM achieves lower execu-

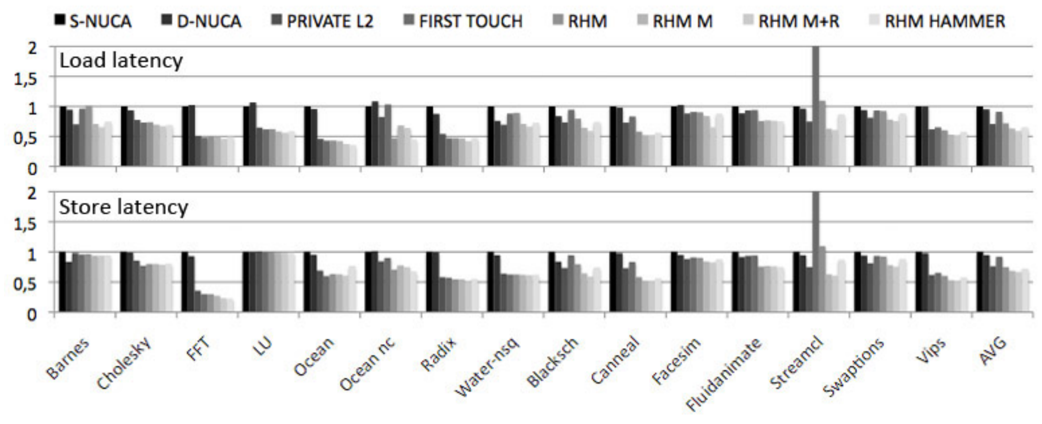


Figure 23: Average load and store latency, normalized to the S-NUCA case.

tion time due to its achieved higher locality in L2. Also, the migration and replication policy helps in further reducing execution time. Contrary to this, D-NUCA is not able to achieve large reductions when compared to S-NUCA. The use of private caches achieves large execution time reductions but its effectiveness depends on the size of the working set of every application. We can also see the on-par execution time benefits of the RHM-HAMMER protocol.

Figure 23 shows the average load and store latency, respectively, for the evaluated configurations, normalized to the S-NUCA approach. RHM configurations reduce these latencies by more than 25% on average and up to 75% (FFT store latency). Again, the effectiveness of RHM in reducing the miss latency depends on the memory access pattern of each application. Streamcluster shows a high percentage of blocks which are first accessed by a tile and then by different tiles during different phases of the application. In this case, a first touch policy has the negative effect of overloading the tile where blocks are mapped, and the migration/replication mechanism can effectively move the blocks to the correct tiles. RHM-HAMMER protocol also exhibits low load and store latencies, as it benefits from the fast GCN and BCN networks.

5.1. Performance Conclusions

When comparing results of different methods we can deduce some interesting observations. First, The S-NUCA approach has the severe limitation of its static mapping of L2 banks. This leads to the largest distances between

L1 requestors and L2 home banks (near 3 hops on average) and the lowest rate in hits in local L2 banks (6% of hits). Execution time of applications is the worst when compared to the other policies. The same occurs for the average load latency and the average store latency.

The D-NUCA approach is a first step towards providing dynamism to the placement policy of L2 homes. However, its static partitioning in bank sets still forces poor results in terms of hop distance (2 hops on average) and hit rate in local L2 bank (less than 20%). Execution time is improved, when compared to S-NUCA because of the lower load and store latencies.

Private caches (PRIVATE L2) obviously achieve low hop distances and the largest hit rate in local L2 banks. However, RHM M+R is able to reduce further the hop distance but not the hit rate. Because of its cache privacy policy, shared blocks impose an overhead which translates to larger execution time and latencies when compared to RHM. Although PRIVATE L2 reduces execution time of S-NUCA by 20%, an extra of 15% is obtained with RHM with migration and replication support.

For FIRST TOUCH, locality is not correctly promoted (average of 2 hop distance and 35% hit rate in local L2 banks). Execution time and average latencies are similar to the ones achieved by D-NUCA. Although FirstTouch is a simple mechanism not requiring any hardware assistance, it should be noted RHM allows finer-grained assignments (blocks vs pages) and also more effective thread migration as blocks can be effectively migrated along with threads.

When analyzing RHM and RHM HAMMER we can see that they achieve very close results. None of them use migration or partitioning support, thus they only differ on the way of locating sharers and owners of blocks. This impacts execution time. RHM HAMMER is able to run faster than RHM (3% faster). This is due to the use of the BCN network. Average miss latencies do not get significantly affected. Notice also that the main benefit of RHM HAMMER is its reduced overhead in control structures.

Finally we can see how RHM M+R is the best RHM option, getting close of the PRIVATE L2 results for hop distances and local hit rate. However, the extra flexibility of moving blocks between L2s makes the solution the most performant. Execution time of S-NUCA is reduced by 35% (execution time of PRIVATE L2 is reduced by 15%).

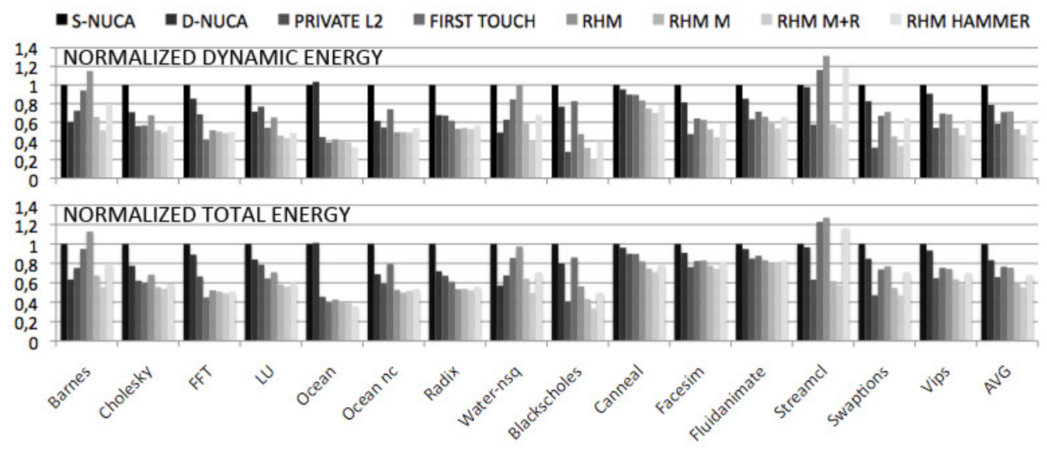


Figure 24: NoC's energy consumption

5.2. Energy

Figure 24 shows the normalized dynamic and total energy consumed by the NoC with the six configurations. Resource access (input buffer read/write, routing, switch allocation, crossbar traversal and link traversal) have been accounted and fed into Orion 2.0 [10]. If the request misses in the local L2 bank, RHM consumes more energy than the other schemes, due to the broadcasts. However, the high percentage of hits in the local L2 leads to less network activity compared to an S-NUCA. This, combined with the reduced execution time, leads to average energy reductions of 32%. Energy consumption is further reduced by 55% on average when migration is enabled (RHM MIGR).

Figure 25 shows the normalized energy consumed by the L2 cache. We used Cacti [9] to obtain the dynamic energy and the leakage per bank. Due to the broadcast access, RHM consumes more dynamic energy than other proposals (50% more energy on average), but the leakage component, reduced by the lower execution time, dominates over the dynamic energy for the configuration we choose. On average, energy consumption with RHM is reduced by 29% without block migration and 31% when block migration is enabled.

The area overhead and the power consumption of the LLC utilization table at the memory controller are a minimal fraction of the overall chip area and power requirement, due to its very small size compared to the on-chip cache and to the limited number of accesses compared to L1 and L2 accesses

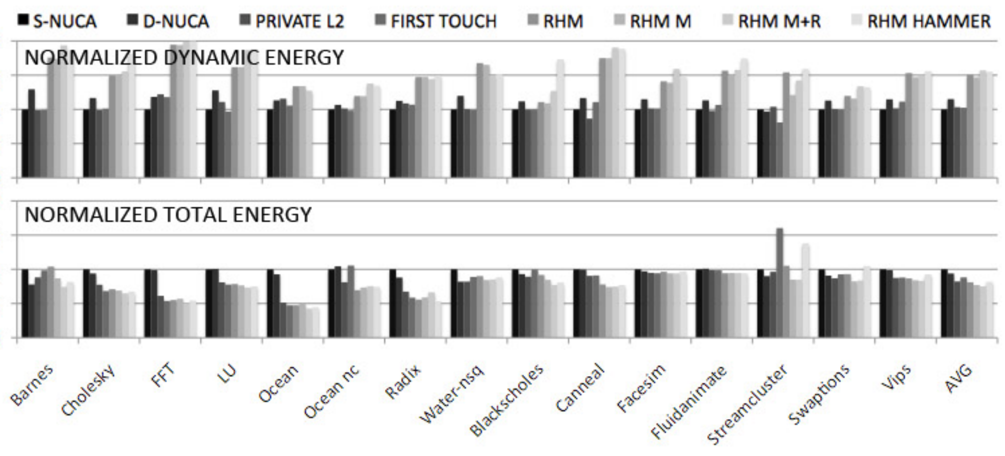


Figure 25: LLC's energy consumption

(the table is only accessed in case of L2 miss).

6. Related Work

To overcome the wire-delay problem [11], the LLC in CMP systems is usually banked, thus offering a wide spectrum of design choices: each bank indeed can expand the private cache of a core or be part of a globally shared LLC. The latter configuration is commonly called a Non-Uniform Cache Access architecture (NUCA), initially proposed by Kim *et al.* for a single core system [1] and then extended to many cores and CMPs [12], [13], and in turn offers many options when implementing the mapping of the blocks on each bank, the home bank search policy [14] [15] and the potential migration [1] or replication [21, 16] of blocks.

Both private and shared LLCs have their advantages and drawbacks, so hybrid configurations have been proposed to exploit the benefits of both design choices, such as ESP-NUCA [17] and CloudCache [18]. CMP-NuRAPID [19] decouples tags and data to allow data placement and replication in any LLC bank. Reactive-NUCA [16] also allows block replication. CMP-NuRAPID however requires an additional bus, while Reactive-NUCA is based on a 2D torus, so they can't be implemented in a 2D mesh-based system. With RHM the implementation of migration and replication mechanisms is straightforward thanks to the totally dynamic mapping policy.

OS-based techniques to achieve a better mapping of the cache blocks to the LLC banks have been proposed by Cho et al.[29], Ros et al. [20], Das *et al.*

[22] to achieve dynamic mapping through OS-level page allocation. Cuesta et al. [23] deactivate the coherence protocol for blocks which are detected as private by the OS. Compile-time and data-based techniques have also been proposed in [24] and [25]. OS- and compiler-based techniques however rely on static mapping at hardware-level and can't support block migration or replication.

Finally, Hammoud *et at.* [2] propose to implement blocks placement strategies at the memory controller(s) to prevent placing a block at an exceedingly pressured local set. To locate cache blocks at the LLC a CTCT [26] policy is assumed, which introduces 3-way communications in some cases, thus increasing the latency of L1 misses.

RHM, differently from previous proposals, allows efficient block search between L2 banks in the whole chip. The optimizations/support at the NoC level allow for an aggressive data placement policy requiring only a small table at the memory controller, and avoiding the 3-way communication of some of the previous solutions, or the static assumption of private caches or OS-level solutions.

7. Conclusions and Future Work

In this work we have proposed Runtime Home Mapping (RHM), a method to perform bank allocation to blocks at runtime. The MC is responsible for the allocation policy and the NoC is co-designed for the efficient support of the coherence protocol.

Different improvements and designs at the NoC level enable fast and efficient location of data. The aim is to allocate L2 home blocks as close as possible to requestors. To find the home of a block, a dedicated control network has been proposed. Also, to speed up broadcast-based protocols, a broadcast network has been designed. Migration and replication of blocks have been also provided.

Results indicate a large span of improvement both in execution time and in reduced miss latencies. In general, RHM is able to achieve 35% lower execution time when compared to S-NUCA approaches and 15% benefit when compared to a private L2 cache organization.

The current work can be extended in many directions, potentially leading to further improvements. Indeed, in this paper we applied baseline methods for different critical design choices of the method. As a first thing, we have plans to evaluate how the search phase behaves with different broadcast implementations on the search method of home nodes and different network topologies. A second direction, linked to the broadcast operation, is the definition of smart mapping strategies located in the memory controller. Finally, we will evaluate the performance of dynamic home mapping combined with virtualization where the memory controller is aware of the partition of chip resources to applications. Also, we plan to improve the home search phase and the migration/replication mechanism to reduce the dynamic energy.

References

[1] C. Kim, D. Burger, S. W. Keckler, An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches, in: Proc. of the 10th Intl Conference on Architectural Support for Programming Languages and Operating Systems, 2002.

[2] M. Hammoud, S. Cho, R. Melhem, A dynamic pressure-aware associative placement strategy for large scale chip multiprocessors, IEEE Computer Architecture Letters 9 (1) (2010) 29–32.

[3] M. Lodde, J. Flich, M. Acacio, Heterogeneous noc design for efficient broadcast-based coherence protocol support, in: Proc. of the 6th Intl, Symposium on Networks on Chip 2012, 2012.

[4] M. Lodde, T. Roca, J. Flich, Heterogeneous network design for effective support of invalidation-based coherency protocols, in: Proc. of the 2012 Interconnection Network Architecture: On-Chip, Multi-Chip Workshop, 2012.

[5] M. Lodde, T. Roca, J. Flich, Built-in fast gather control network for efficient support of coherence protocols, in: to appear in IET Computers and Digital Techniques INA-OCMC 2012 Special Issue.

[6] The nangate open cell library,45nm freepdk,available at https://www.si2.org/openeda.si2.org/projects/nangatelib/.

[7] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, A. Agarwal, Graphite: A distributed parallel simulator for multicores, in: The 16th IEEE Intl. Symp. on High-Performance Computer Architecture, 2010.

[8] T. E. Carlson, W. Heirman, L. Eeckhout, Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations, in: Intl. Conference for High Performance Computing, Networking, Storage and Analysis, 2011.

[9] Cacti 5 technical report, available at http://www.hpl.hp.com/techreports/2008/hpl-2008-20.html.

[10] A. Kahng, B. Li, l. Peh, K. Samadi, Orion 2.0: A power-area simulator for interconnection networks, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 20 (1) (2012) 191–196.

[11] D. Matzke, Will physical scalability sabotage performance gains?, Computer 30 (9) (1997) 37–39.

[12] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Bourger, S. Keckler, A nuca substrate for flexible cmp cache sharing, in: Proc. of the 19th Intl Conference on Supercomputing, 2005.

[13] B. Beckmann, D. Wood, Managing wire-delay in large chip-multiprocessors caches, in: Proc. of the 37th Intl Symposium on Microarchitecture, 2003.

[14] J. Lira, C. Molina, A. Gonzales, Hk-nuca: Boosting data searches in dynamic non-uniform cache architectures for chip multiprocessors, in: Proc. of the 2011 IEEE Intl Parallel and Distributed Processing Symposium, 2011.

[15] R. Ricci, S. Barrus, R. Balasubramonian, Leveraging bloom filters for smart search within nuca caches, in: Proc. of the 7th Workshop on Complexity-Effective Design, 2006.

[16] N. Hardavellas, M. Ferdman, B. Falsafi, A. Ailamaki, Reactive nuca: near-optimal block placement and replication in distributed caches, in: Proc. of the 36th Intl Symposium on Computer Architecture, 2009.

[17] J. Merino, V. Puente, J. Gregorio, Esp-nuca: A low cost adaptive non-uniform cache architecture, in: Proc. of the Intl Conference on High Performance Computer Architectures, 2010.

[18] H.Lee, S. Cho, B. Childers, Cloudcache: Expanding and shrinking private caches, in: Proc. of 44th Int'l Symp on High-Performance Computer Architecture, 2011.

[19] Z. Christi, M. Powell, T. Vijaykumar, Optimizing replication, communication and capacity allocation in cmps, in: Proc. of Intl Symposium on Computer Architecture, 2005.

[20] A. Ros, M. Cintra, M. Acacio, J. Garcia, Evaluation of low-overhead organizations for the directory in future many-core cmps, in: Proc. of the Intl Conference on High Performance Computing, 2009.

[21] P. Foglia, C.A. Prete, M. Solinas and G. Monni. Re-NUCA: Boosting CMP Performance Through Block Replication, in: Euromicro DSD 2010

[22] A. Das, M. Schuchhardt, N. Hardavellas, G. Memik, A. Choudhary, Dynamic directories: A mechanism for reducing on-chip interconnect power in multicores, in: Proc. of Design, Automation and Test in Europe, 2012.

[23] B. Cuesta, A. Ros, M. Gomez, A. Robles, J. Duato, Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks, in: Proc. of the 38th Intl Symposium on Computer Architecture, 2011.

[24] Y.Li, A.Abousamra, R.Melhem, A. Jones, Compiler-assisted data distribution for chip multiprocessors, in: Proc. of 19th Intl Conference on Parallel Architectures and Configuration Techniques, 2010.

[25] Y.Zhang, W.Ding, M.Kandemir, J. Liu, O. Jang, A data layout optimization framework for nuca-based multicores, in: Proc. of 44th Intl Symposium on Microarchitecture, 2011.

[26] M. Hammoud, S. Cho, R. Melhem, Acm: An efficient approach for managing shared caches in chip multiprocessors, in: Proc. of the 4th High Performance Embedded Architectures and Compilers Int'l Conference (HiPEAC-09), 2009, pp. 355–372.

[27] J.M. Owen , M.D. Hummel , D.R. Meyer and J.B. Keller. United states patent: 7069361 - system and method of maintaining coherency in a distributed communication system. June 2006.

[28] P. Conway and B. Hughes. The amd opteron northbridge architecture. *IEEE Micro*, 27(2):10–21, March 2007.

[29] Cho S. et al: Managing Distributed, Shared L2 Caches through OS-Level Page Allocation MICRO 2006