Universitat Politècnica de València

Master Thesis

Efficient Built-In NoC Support for Gather Operations in Invalidation-Based Coherence Protocols

Author:
Mario Lodde

Advisor:
Prof. José Flich Cardo

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

February 2013

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Tècnica Superior de Ingeniería Informática
Department of Computer Engineering

Master of Science

Efficient Built-In NoC Support for Gather Operations in Invalidation-Based Coherence Protocols

by Mario Lodde

Abstract

Future chip multiprocessors (CMPs) will include tens and hundreds of cores organized in a tile-based design pattern. A built-in on-chip network (NoC) will be devoted to inter-tile communication. In these systems, a shared memory programming model is appealing since it simplifies programming efforts. However, a coherence protocol is needed to keep consistency in the data stored in the different levels of the cache hierarchy. Usually an invalidation-based protocol is used, where cached copies are invalidated before a processor writes on a memory block.

In this work we propose a NoC re-organization in which a small and fast control network is dedicated only to messages related to the invalidation process. By doing this, application traffic is alleviated and traffic overhead is significantly reduced. We explore different coherency protocols to cope with an efficient mapping of network control messaging, thus optimizing the use of the control network. We address two design points in coherence protocols. The first one with a full map directory, where a precise list of the sharers of a block is recorded in the directory, and the second one without directory, where no list of sharers is maintained by the protocol, as it happens in Hammer protocol.

Experimental evaluation shows significant gains in performance when using the additional control network. With a low area overhead (less than 2.5%), the control network reduces NoC traffic and miss latency, thus reducing execution time up to 16%. Simulation results show a reduction of network traffic up to 80% and a reduction of store and load miss latency up to 70% and 40% respectively.

Contents

List of Figures

List of Tables

Acronyms

CMP	**C**hip **M**ulti**P**processor
NoC	**N**etwork-**o**n-**C**hip
MOESI	**M**odified - **O**wned - **E**xclusive - **S**hared - **I**nvalid
SAGN	**S**et-**A**side **G**ather **N**etwork

Chapter 1

Introduction

Chip multiprocessor (CMP) systems with up to ten cores are common nowadays, and the trend indicates that the number of cores will increase up to hundreds. In a tile-based CMP design, each tile includes a core with its cache hierarchy and a switch connected to the switches located at neighboring tiles, typically forming a 2D mesh on-chip network (NoC) [1]. Two different programming models can be used, depending on how cores communicate to each other. If a message passing model is used, the cores communicate by exchanging messages explicitly generated by the application, while with a shared memory model cores communicate through shared variables. The shared memory programming model is easier for programmers as communication is implicit when accessing variables. However, it requires the use of a cache coherence protocol to keep data consistency among all levels of caches and main memory, thus keeping track of which copy of data is valid. The traffic generated in the NoC comes mostly from the coherence protocol.

Invalidation-based cache coherence protocols keep consistency by invalidating all the shared copies of a block before a write operation is performed. This way, all the cores always read the value produced by the last write operation. The performance penalty of invalidation-based protocols relies in the high write miss latency, induced mostly by the indirection performed in the coherence protocol. In case of a write miss, a core has to send a write request to the L2 cache, which will send an invalidation message to each L1 holding a copy of the block. The requestor cannot write until it receives the requested data block and all the acknowledgements to the invalidation operation. Since *inv/ack*

messages are sent through the on-chip network, they have an impact both on the write miss latency and on the network traffic.

The L2 cache stores the information about the sharers of a block in a data structure, the directory, which is distributed between the L2 cache banks: each bank have a side structure including the directory information for the subset of addresses mapped on that bank. This information includes the state of the block and a bit vector pointing to the L1 caches which have a copy of the block, also called sharing code [2]. If the number of bits in this vector is equal to the number of cores, the directory can indicate exactly which cores have a copy of the block in their L1 cache. This is known as a full map directory. However, this organization adds an area and energy overhead that does not scale, since its requirements increase linearly with the number of cores [3]. To reduce the overhead introduced by the directory, the sharing code can be compressed: in this case more than one core is mapped to the same bit of the vector. This organization reduces the size of the directory but increases network traffic: invalidations will be sent both to the actual sharers and to nodes which are not sharers but are mapped to the same bits of the sharers. The more the sharing code is reduced, the more directory overhead is saved and traffic increased. On the other hand, at the extreme design point, we can eliminate completely the bit vector. In that case we have a Dir_0B protocol, where there is no information about which cache has a copy of a particular memory block. Thus, a broadcast invalidation operation to all the nodes must be triggered to invalidate the sharers before a write operation on a shared block. The Hammer protocol employed in systems built using AMD Opteron processors [4, 5] is the most representative example of the latter[1]. We will consider the two extreme design points mentioned above, referring to them as directory-based protocol if the directory has an exact representation of the sharers (full-map directory) and as broadcast-based protocol if the directory has no sharing code.

In our research, we focus on the co-design of the NoC and the coherence protocol, with the goal of optimization and thus obtaining better performance and less power consumption. In this work, we focus, as a first step, on a dedicated control network that collects part of the control messages involved in a coherence operation. In particular, *ack* messages are handled through the control network, thus reducing network traffic and

[1]We use the terms "broadcast-based directory protocol" and "Hammer protocol" interchangeably along this work.

lowering the latency of the invalidation process. The directory-based protocol is also adapted to get the maximum profit from the control network. The target is to reduce the miss latency of the overall system.

The control network is tightly coupled with the NoC in the sense that a multicast operation within the NoC configures the control network. The control network, then, is used to speedup *ack* messages sent from multiple end nodes to a single end node. This control network is totally decoupled from the normal flow control and switching mechanisms typically found in NoCs. In contrast, the control network does not forward any message as in its simpler implementation it only consists of a single control wire for each possible destination. In this work we propose two different implementations of the control network, one being fully asynchronous (but wire demanding) and one being synchronous with the NoC (and requiring less wiring resources). In addition we apply the control network to both the full map directory protocol and the Dir_0B protocol. Results demonstrate a speedup factor in performance of up to 8% with a reduction in NoC traffic of up to 80%.

The rest of this work is organized as follows. In Chapter 2 the basic invalidation-based and broadcast-based protocols are described. In Chapter 3 we detail the control network that handles part of the invalidation messaging and we adapt the coherence protocols to the control network. Then, in Chapter 4 we evaluate the impact of the control network on the performance of the coherence protocols. Finally, Chapter 5 describes the related work, and in Chapter 6 we draw the conclusions.

Chapter 2

Basic Coherence Protocols

In this section we shortly describe the basic directory-based and broadcast-based protocols. We assume as the base system a 16-tile CMP system organized in a 4×4 layout, each tile including a switch, a core, its private L1 cache and a shared L2 cache bank. All the cores share the distributed L2 cache, and each L2 bank acts as *home* for a subset of the shared memory address space, meaning that all the requests for blocks of that subset are sent to that bank. Figure 2.1 shows the organization of the CMP system we consider in this work.

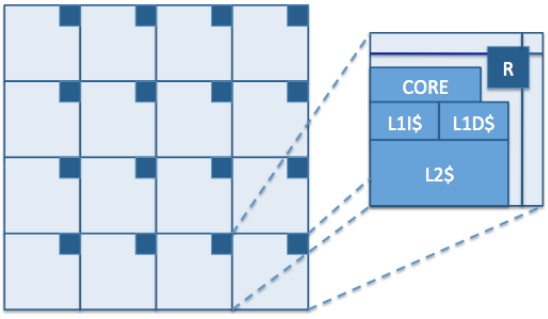


FIGURE 2.1: Base 4×4 tiled CMP system

Whenever a processor issues a read or a write operation, the private L1 cache is accessed. In case of a miss, a request is sent to the L2 bank which is the *home* of the requested block, which also stores the directory information for that block.

2.1 Directory-based Protocol

The full map directory-based protocol uses a sharing code with a number of bits equal to the number of cores of the system, thus giving an exact information about which cores have a copy of the block in their private L1 caches. Figures 2.2 and 2.3 show the four basic cases of block requests.

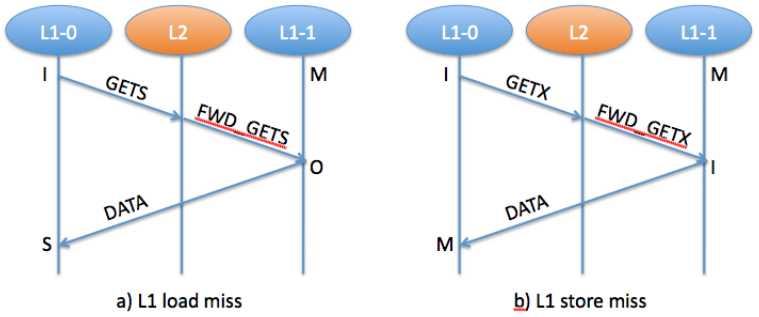


FIGURE 2.2: Requests for a private block (Directory protocol)

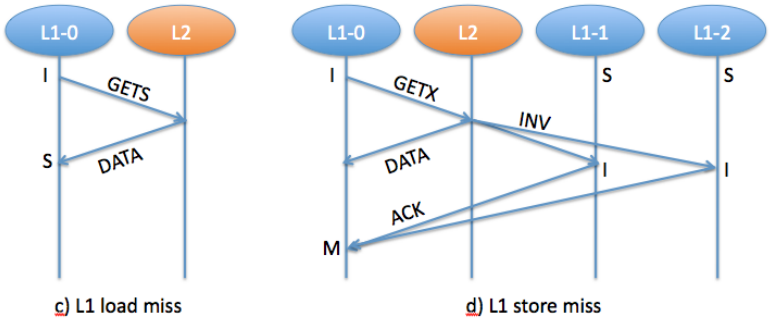


FIGURE 2.3: Requests for a shared block (Directory protocol)

Figure 2.2.a shows how a read miss is resolved if the requested block is private (only one processor has a copy of the block with permission to read and write). The involved L1 nodes are identified by the tile number (L1-0 is the L1 at tile 0) and the L2 bank is the *home* of the requested memory block. Figure 2.2.a also shows the state of each L1 entry (assuming a MOESI protocol). The requestor (L1-0) sends a GETS message to the home L2, which forwards the request to the L1 which owns the block (L1-1) and adds the requestor to the sharer list. The owner (L1-1) sends then a copy of the block back to the requestor (L1-0). The L1-0 entry is set to S (sharer) state and L1-1 entry is set to O (owner) state. In case of a write miss (Figure 2.2.b) to a private block, the requestor (L1-0) sends a GETX message to L2, which forwards the request to the owner (L1-1), which in turn invalidates its copy and sends the block back to the requestor.

Then, L1-1 entry state is set to M (modified) and L1-0 entry state is set to I (invalid). The L2 changes the Owner pointer of the directory entry from L1-1 to L1-0.

Figure 2.3.c shows how a read miss is resolved if the requested block is shared by a set of processors (each sharer has a copy of the block in its private L1 with read only permission). The request is forwarded to the L2, which adds the requestor to the sharers list and sends him a copy of the block. The entry in L1-0 is set to S (shared). In case of a write miss, Figure 2.3.d, the directory sends the data to the requestor and issues invalidation messages to the sharers. The sharers, upon the reception of the invalidation message, send an acknowledgement (*ack*) message to the requestor, which blocks its write operation until all *ack*s and the data block have been received. The message sent by the L2 bank specifies the number of *ack*s the requestor has to wait for. At the end, the L1-0 entry is set to M (modified) while the sharers set their entry to I (invalid).

Since the full-map directory indicates exactly which cores share or own a block, the minimum amount of traffic required by the coherence protocol is injected in the network. The size of each directory entry however grows linearly with the number of nodes, which makes this solution not suitable for large systems: a CMP with 256 nodes, for instance, requires a 32-byte directory entry. If we assume 64-byte cache lines, the directory would increase the overall L2 cache size by 50%.

2.2 Broadcast-based Protocol

The broadcast-based protocol does not use the sharing code, so the directory area overhead is completely removed but each time an L2 cache bank has to forward a request or send invalidations to the sharers, it issues a broadcast message which is received by all cores. Each core in turn must answer to the broadcast message, with the copy of the block if it is going to send a copy of the requested block to the requestor or with an *ack* if it has received the message and performed the operations stated by the coherence protocol but will not send a copy of the block to the requestor.

So the broadcast-based protocol behaves as the directory-based protocol in case of read request for a shared block (case c in Figure 2.3) while in all other cases a message is broadcasted to all cores and the requestor receives a response message from all other

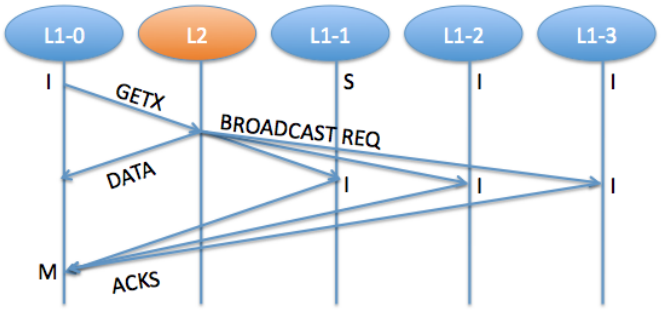


FIGURE 2.4: Write request for a shared block (Hammer protocol)

cores. The requestor is allowed to read or write the block once it receives the data and all the ACKs.

The case of a write request for a shared block in a 4-core system is shown in Figure 2.4. The data block is provided by the L2 cache and the request is broadcasted to all the L1 caches of the system. All nodes reply to the broadcast by sending an ACK and invalidating the local copy, if any. L1-0 can write once it receives the data and all the ACKs.

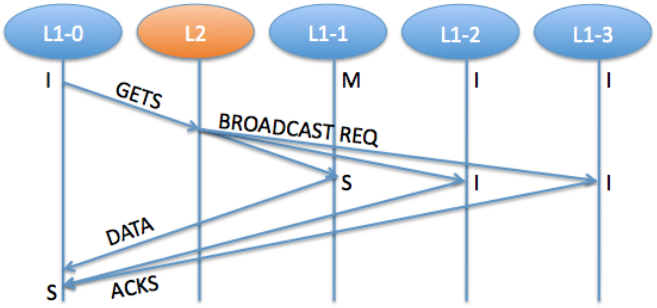


FIGURE 2.5: Read request for a private block (Hammer protocol)

Figure 2.5 shows the case of the read request for a private block. This time the L2 cache just forwards the request to all the L1s without sending the data to the requestor. When

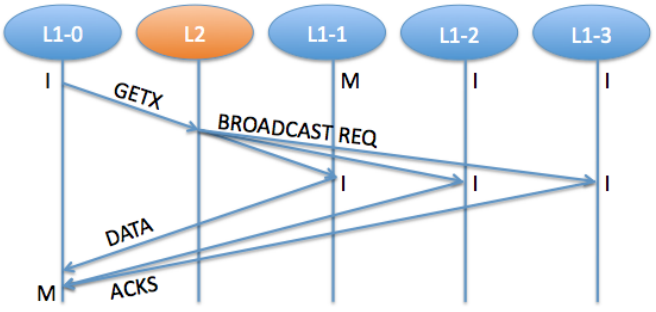


FIGURE 2.6: Write request for a private block (Hammer)

the Owner receives the broadcast request, it sends the block to the requestor[1]. All other nodes just send an *ack* to the requestor. A similar behavior is shown in Figure 2.6 for a write request on a private block.

2.3 Multiple Requests

The L2 home bank acts as synchronization point when multiple requests for the same block are issued at the same time by different tiles: the cores access the requested block following the order in which the requests are received at the home bank. Additional states and control messages must be added to the basic protocol to correctly address all the possible race conditions caused by the reception of a request while the requested block is in a transient state. Expecially the transition of a block from a shared to a private state and vice versa must be handled carefully.

With these two coherence protocols in mind, the directory-based and the broadcast-based, in the next chapter we describe the NoC support for gathering messages. Notice that we will provide two different but similar solutions, one for each protocol.

[1] Additional traffic between the Owner and the L2 to manage the block state change from private to shared is not shown for simplicity reasons.

Chapter 3

Set-Aside Gather Network

3.1 Overview

To speedup the reception of *ack* messages we propose a configurable control network with dedicated logic at every router. In particular, we define a so-called Set-Aside Gather Network (SAGN) from every core to every core. Thus, in a 16-tile system with a core in each tile we define 16 SAGNs, each with 15 sources reaching a different and single destination. Each SAGN consists of a tiny one-bit network spread throughout the NoC with one AND logic gate on every switch.

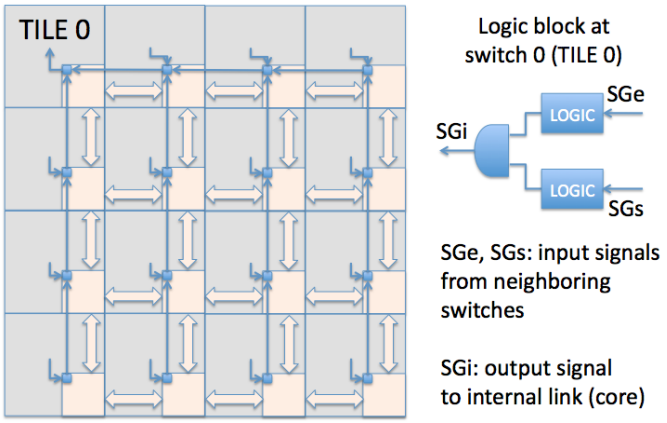


FIGURE 3.1: Set-Aside Gather Network (tile 0)

Figure 3.1 shows a schematic view of the switch logic with the support for one SAGN network. In particular, SAGN for TILE 0 is shown. The network follows the Y-X routing algorithm, which follows the X-Y routing assumed for data messages through

the regular NoC but in opposite direction. In each switch the AND gate combines the different input signals from neighboring switches and from the local L1 cache.

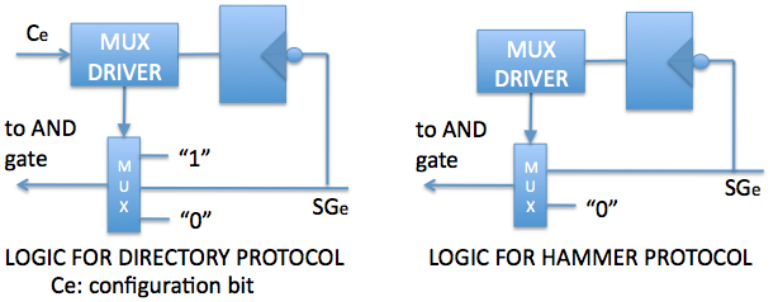


FIGURE 3.2: Logic at the inputs of the AND gate

Figure 3.2 shows the logic at the inputs of the AND gate. The input signals are configured by a C_x bit (x is the input port the signal is coming from), one per each input signal, which enables/disables that input signal. If the input signal is disabled (the associated configuration bit is set to zero), the logic sets the input to the AND gate to one. On the contrary, if the input signal is enabled, the input to the AND gate is the input signal. Notice that these bits are used to build a control tree network ready to allow all the sharers of a block to acknowledge the requestor. Notice also that these configuration bits are not needed for a Dir_0B coherence protocol since a broadcast action reaches all the nodes, thus the AND tree does not need to be configured every time. All the nodes participate in the ACK stage with the requestor. With the directory-based protocol, however, the configuration bits enable only the inputs of the nodes which will actually receive the invalidation message and have to send an *ack*. In both cases, the SAGN is not flow controlled and does not implement any switching mechanism. Indeed, is a pure combinational block with no input clocks. Later we propose a synchronous version.

An extra logic is necessary to reset the input signals each time a new invalidation message is multicasted (in case of directory-based protocol) or broadcasted (in case of hammer protocol). For the directory-based protocol, the multiplexer allows up to three entries, "0", "1", and the input signal. When the multicast message arrives to the switch, if the input port belongs to the AND tree, then the multiplexer is configured to the "0" signal, to prevent a too-early notification due to acknowledgements for previous requests. As the multicast message travels down to the next switch, and configures the AND gate at that switch, the input signal will be set to zero and will reach the current switch. In that case, the logic detects the transition of the input signal (through the flip-flop)

and restores the multiplexer entry to the input signal, thus preventing the output of the AND gate being set to one too early. Notice that if a multicast message arrives and an input port is not part of the AND tree, then the multiplexer entry is set to the "1" signal. For the broadcast solution, the "1" input to the multiplexer is not required as all possible inputs always belong to the AND tree.

Multicast invalidation messages are used to properly set the configuration bits for the directory-based protocol: when *inv* messages are sent through the NoC, every switch configures the SAGN network for the corresponding sender. If a multicast *inv* message is forwarded through several output ports, the corresponding SAGN configuration bits (C_x) for the input signal of the output ports are set. Thus, in a tree branch of the multicast operation of the *inv* message, several input signals will be enabled.

The SAGN is a fast built-in network that enables a fast notification to the sender (TILE 0 in the figure) due to its simplicity. Indeed, when an *inv* message is received at a node, it simply needs to trigger the signal for the SAGN of the sender. Once all the sharer nodes receive the multicast message and trigger the SAGN signal, the sender will be notified through the SAGN within the delay of the AND tree network.

3.2 Detailed Description of a Logic Block

The SAGN logic at each switch is connected to its neighbors' control logic blocks with dedicated wires. Figure 3.3 shows the gather network logic at switch 5. Notice that each subnetwork is made of a single wire, coming from different input ports. The task of the logic block is simply to AND the corresponding input signals and to distribute the results through the proper output port, depending on the location of the switch in the mesh topology and the selected layout.

The logic receives as input 15 control signals from the local core, each of which is addressed to a different destination node. X_L indicates a control signal coming from the local port and addressed to destination X. Thus, we have from 0_L up to 15_L signals (excluding the one with the ID of the local core). In addition, we have up to 20 control signals coming from either north (N) or south (S) input ports and up to 5 control signals from either east (E) or west (W) input ports. Switches at the boundaries of the mesh have a lower number of input control signals. These signals are then grouped depending

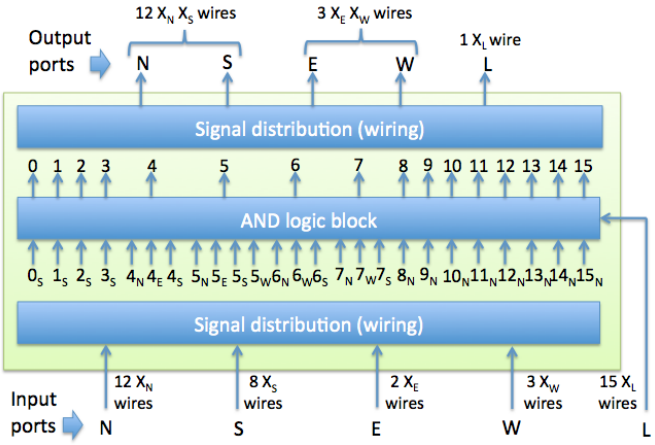


FIGURE 3.3: SAGN block at switch 5

on the destination tile and assigned to the corresponding inputs of the AND gate array. Notice that most input signals will not be required, thus simplifying the array. The outputs of the AND gates are then distributed over the output ports, depending on the location of the switch and the layout. Notice that 16 output control signals are generated, one per destination in the system, 15 of which are sent to neighboring switches and one to the local node.

The required logic at each switch is small, consisting of 16 AND gates, a set of multiplexers, configuration bits and associated logic. The signal distribution blocks are simply a rearrangement of the input and output control signals to the appropriate inputs and outputs of the AND gates.

3.3 Control Signals Distribution

One important aspect of the control network is the floorplan of the wires over the NoC area. Figure 3.4 shows the number of wires of the SAGN control signals between the switches. Each switch handles both input and output control signals through all its ports. For a $N \times N$ mesh NoC, the number of outgoing control signals through all the output ports of a switch is $N^2 - 1$, in our case 15. Each control signal is a different one-bit subnetwork addressed to a different destination ($N^2 - 1$ destinations). Notice that some output ports handle more control signals than others. This is due to the mapping we have performed for the control signals, following the Y-X layout.

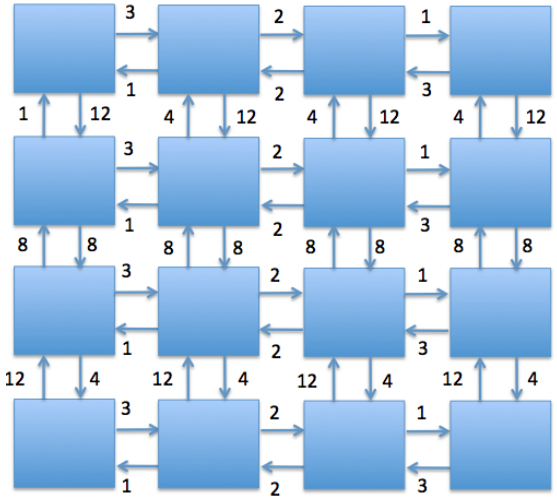


FIGURE 3.4: Control signals distribution for the gather network. YX layout

Alternatively, we can design a different mapping strategy, where XY is used instead. In this case we have a mirror effect on the distribution of wires between horizontal ports and vertical ports. To better balance the wires, we can use a mixed approach where wires for half the nodes are mapped YX and wires for the other half of nodes are mapped XY. The latency through the SAGN does not change as the path follows the same manhattan distance. Figure 3.5 shows the case where nodes with the underlined ID number follow the YX mapping and the rest follow XY mapping. In this case we achieve a perfect distribution of wires, where each bidirectional port handles 10 wires for a 4×4 mesh network. Notice that this mapping cannot be used with directory-based protocol, since the SAGN must always follow the X-Y routing followed by broadcast messages (but in opposite direction) for a proper configuration of the (C_x) bits. However, it can be used for the broadcast-based protocol.

As the system size increases, the number of wires increases, but not the logic complexity at switches. In practice, for a $N \times N$ network, this mapping strategy requires $(N^2+N)/2$ wires per direction and dimension. For a 8×8 system, the number of wires per port is 36, well below typical CMP port widths of 256 bits. For a 16×16 system, the number increases up to 136 wires.

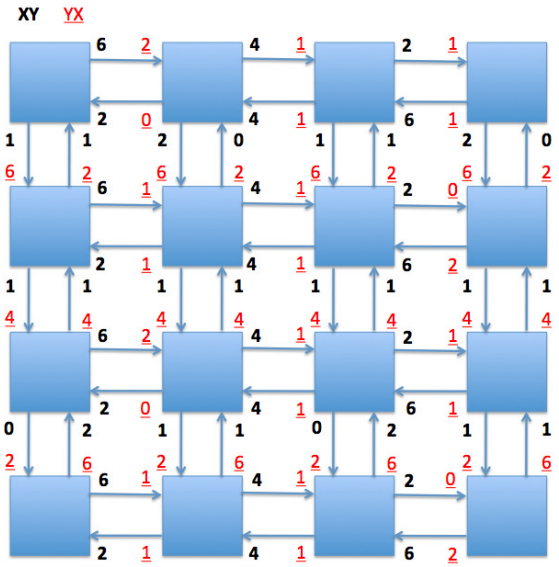


FIGURE 3.5: Control wire distribution for mixed XY/YX mapping. Gather networks for odd destinations follow XX routing and gather networks for even destinations follow XX routing.

3.4 Sequential Implementation of the SAGN

The previous design provides a combinational logic, which is simple and fast. However, the number of wires dedicated to the SAGN increases with the number of cores, potentially leading to an unacceptable number of wires between switches. Also, the design can handle only one request per core; to handle multiple requests at the same time it would be necessary to have more than one dedicated network per core, thus increasing the number of wires.

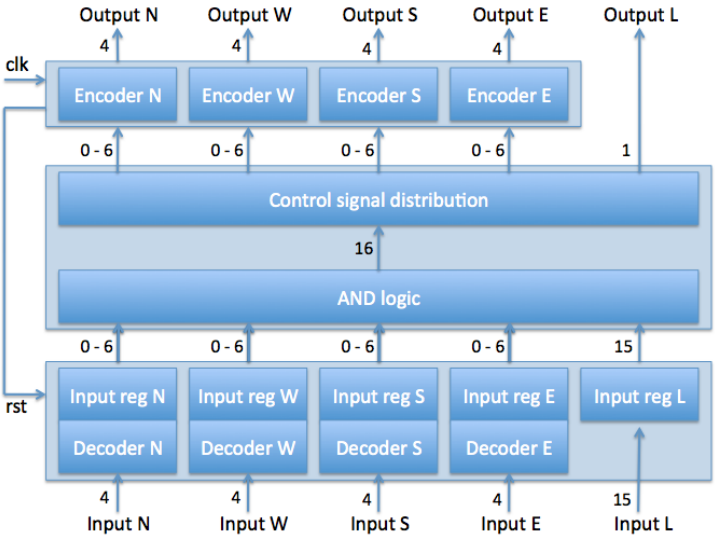


FIGURE 3.6: Sequential SAGN block at switches for a 4×4 mesh NoC

The implementation we describe in this subsection reduces the number of wires to a logarithmic scale and allows to handle multiple requests to the same tile at the cost of increased latency: while the combinational SAGN is not bound to the clock frequency of the switches, this implementation uses sequential logic and has a latency of 1 cycle per hop (we will discuss latency issues in detail in Chapter 6). As will be shown in the evaluation section, this is not an issue, since the increased latency of the SAGN will not affect the performance. On the other hand, the number of wires between switches will be drastically reduced, providing a scalable solution for large systems (beyond 64 tiles).

Figure 3.6 shows the sequential implementation of the SAGN at each switch. Each port has its own SAGN decoder and encoder, through which it can receive and send the IDs of the nodes to which the *ack* is sent. This reduces the number of wires, since each port will only have, in the system we considered, 4 input wires and 4 output wires (a reduction from $(N^2+N)/2$ to $\log_2(N \times N)$). The received IDs are decoded and saved in the input registers. When all expected *ack*s are received for a node, the node ID is encoded and transmitted through the proper port. Notice that the AND logic block at each switch is the same for both implementations. The number of wires in each connection shown in Figure 3.6 refers to a 4×4 system with a mixed XY-YX control signal distribution. The number of wires connecting the AND logic block with the input and the output blocks varies from 0 (in case the tile does not have any connection through that specific direction) to 6 depending on the tile position in the 2D mesh.

To allow multiple requests, a request ID can be transmitted together with the core ID: this way with $\log_2(N \times N)$+R bits per port per direction it is possible to handle 2^R requests per core in an $N \times N$ system. This means, for instance, that a sequential SAGN can handle 4 requests per core in a 16×16 system using 6 bits per port per direction, which adds a low overhead to the typical NoC ports of 128 or 256 bits.

The input register has a bit for every possible input control signal for every SAGN. This means, theoretically needing a register with 5N bits, five input ports and N SAGNs. However, this is not the case for the XY-YX mapping.

Once all the input signals for a SAGN are received (their corresponding bits at the input registers are set), the AND gate output signal is set and forwarded to the corresponding output port. There, an arbiter selects one output signal (notice at the same cycle two AND gates can be activated for two different SAGNs outputs through the same output

port). The selected output signal (belonging to a SAGN) is coded and stored in the output register (behaving as a latch). Notice that once the output signal is forwarded to the next switch, the reset logic sets to zero all the bits in the input registers belonging to that SAGN.

3.5 Modified Protocols

The directory-based protocol described in Chapter 3 must be modified to work properly with the SAGN. From the SAGN description, it can be noted that a tree structure must be configured between the node that triggers *inv* messages and the nodes that receive the *inv* message. Once the tree is configured, the SAGN collects all the *ack* messages in a fast manner. To take benefit of the control network we implement a switch with multicast support. The node triggering *inv* messages will send a single multicast message to all the sharers, following the XY routing. The message is used at every router to configure the control network bits. Notice from Figure 2.3.d the sender of *inv* messages differs from the receiver of *ack* messages in an invalidation operation. This prevents configuring the SAGN network in a proper way when the directory-based protocol is used. We need to adapt the coherence protocol in a manner that the sender of *inv* messages also receives all the *ack* messages. Thus, both stages will follow the same paths through the network (but in opposite direction[1]).

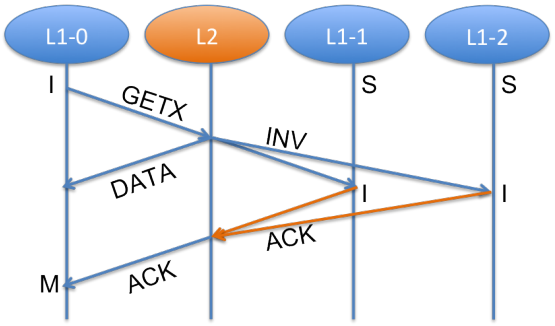


FIGURE 3.7: First alternative: the L2 invalidates the sharers

We propose two alternative designs of the basic directory-based protocol described in Chapter 2. In the first modified protocol, shown in Figure 3.7, the home L2 node sends a multicast invalidation message to all the sharers, which send the *ack* back to the L2

[1]YX mapping is needed for this case.

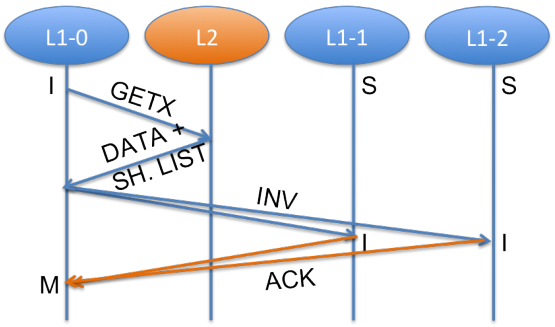


FIGURE 3.8: Second alternative: the L1 invalidates the sharers

node through the SAGN network. Once the home receives all the *acks* (the SAGN output is set), it sends an *ack* message to the requestor. In the second modified protocol, shown in Figure 3.8, the home L2 does not invalidate the sharers. In contrast, it piggybacks the sharers list to the data message sent to the requestor. The requestor, then, handles the invalidations by sending a multicast *inv* message. The sharers just notify the requestor through the SAGN network. Once all ACKs are received, the requestor unblocks and can issue a new request. In both cases, invalidating memory copies will take four steps, one more than the basic protocol. However, due to the fast reception of acknowledgements, both alternative protocols will achieve better performance than the basic protocol. Notice that the serialization effect of *ack* messages at destination is removed when using the control network. For the Dir_0B protocol, these changes are not needed. Indeed, when the L2 home node sends a broadcast request, it includes in the packet the ID of the node that has to be acknowledged. Thus, the nodes select the proper SAGN wire.

Chapter 4

Evaluation

In this chapter we provide evaluation results of the gather network. First we provide results for the implementation of the network. Then, we focus on cycle-accurate performance estimations using synthetic memory access traces and real applications.

4.1 Implementation Analysis

In this section we provide an analysis of the overhead of the control network. We have designed a basic 4-stage pipelined 5-port switch with buffers at the input side and with wormhole switching. The output of the switch is registered. Each input port has one 4-flit-wide buffer. Later, we will use virtual channels to obtain performance numbers with applications. Virtual channels are used to separate different traffic classes (a message will be routed always through the same VC), thus avoiding the protocol deadlock problem induced by coherence protocols. Link width and flit size are set to 8 bytes. Stop&Go flow control protocol and XY routing have been implemented. A round-robin arbiter according to [6] is used.

The switch has been implemented using the 45nm technology open source Nangate [7] library with Synopsys DC. Cadence Encounter has been used to perform the Place&Route. Table 4.1 summarizes the delay and area for each of the modules of the switch[1].

[1] Area numbers in the table are for a single instance of each module, thus some of them are replicated in the complete switch.

Notice that these values do not take into account the link delay neither the control logic needed to implement the communication between switches. Table 4.2 shows the whole network critical path considering the switch delay plus the link delay.

module	area (mm^2)	critical path (ns)
Input port	3.08×10^{-3}	0.58
Routing	8.91×10^{-5}	0.30
Arbiter	1.09×10^{-3}	0.74
Crossbar	4.47×10^{-3}	0.43

TABLE 4.1: Area and delay for the switch modules

4.1.1 Combinational SAGN

An extra module with the combinational SAGN circuit has been added to the switch we just described. The configuration bits are computed in the routing module. By using the switch, a 4×4 and 8×8 2D mesh networks have been implemented. Two scenarios are analyzed, a conventional 2D mesh without the control network, and the 2D mesh with the control network.

Table 4.2 shows the critical path (end to end delay) of the control network analyzed independently of the rest of the network. To compute the combinational SAGN critical path, each block must be properly placed next to the switch it is connected to. Then, on the implementation process some constraints must be forced to the placement&route tools. First, the highest metallization layers must be used. By doing this, lower metallization layers get free, and hence, other logic as SRAMs could be placed under SAGN wires. Repeaters are inserted by the own tool in order to fulfil delay constraints imposed by the designer. The critical path of the control network is fixed by the SAGN logic that connects the two most physically separate nodes in a chip. Notice also that the latency of the control network depends on the mesh radix. The table also shows the delay of a single switch. Two link lengths are analyzed: 1.2 mm and 2.4 mm.

Critical path (ns)	4x4 Network		8x8 Network	
link length (mm)	1.2	2.4	1.2	2.4
Control network	1.23	2.20	2.65	4.32
switch delay	1.35	1.75	1.35	1.75

TABLE 4.2: Control network critical path

For a 4×4 network with a link length of 1.2 mm, the control network critical path is smaller than the delay of a single switch, and hence, the control network is able to

work at the same operating frequency than the switch (in a switch cycle the control network is able to notify all the nodes about possible *ack* messages). In contrast, if the link length is increased, the control network has a higher critical path. However, only two cycles are needed for the control network. For the 8×8 network, it can be seen that the control circuit does not scale as well as the point-to-point communication protocol of the NoC. However, it can be noticed that the worst case is for a control network with a delay of 4.32 ns (3 clock cycles when compared to the switch). The area of the switch is 20.418×10^{-3} mm^2, while the area of the control logic in each switch is 0.28 $\times 10^{-3}$ mm^2, being a 1.3% overhead. Notice that when virtual channels will be added the area overhead will be much lower.

4.1.2 Sequential SAGN

The sequential control logic described in Section 3.4 has also been implemented. This implementation has a higher area overhead than the combinational implementation, being basically the same circuit with decoders at the input ports and encoders at the output ports. The encoders at each output port also include an arbiter in case more than one ACK signal is generated at the AND logic block in the same cycle. Since the control logic area varies depending on the tile position, we considered the worst case, which in a 4×4 system with mixed XY/YX mapping is one of the switches located in the central tiles. These switches indeed are connected with another switch in each direction and are crossed by more signals than the switches located on the border. The area of the control logic in these tiles is $0.472 \times 10^{-3} mm^2$, which is 2.3 % the area of the switch. The critical path of the control logic in the worst case is 0.52 ns, which is lower than the critical path of the slower module of the router, which is 0.74 ns for the arbiter, as shown in Table 4.1. This means at each cycle the control logic can propagate up to 5 ACK signals (one per port). The control logic area is reduced to $0.153 \times 10^{-3} mm^2$ for the switches located at the corners of the 2D mesh, which are connected only in two directions.

4.2 Performance Evaluation

4.2.1 Combinational SAGN

We have implemented and evaluated four versions of the directory-based protocol using the gNoCsim simulation platform. gNoCsim is composed by two main modules: the memory module which simulates the memory hierarchy and the coherence protocol, and the network module which performs a cycle-accurate simulation of the NoC. The simulator is fed by a set of memory access traces, that can be generated by a functional simulator connected to gNoCsim [2].

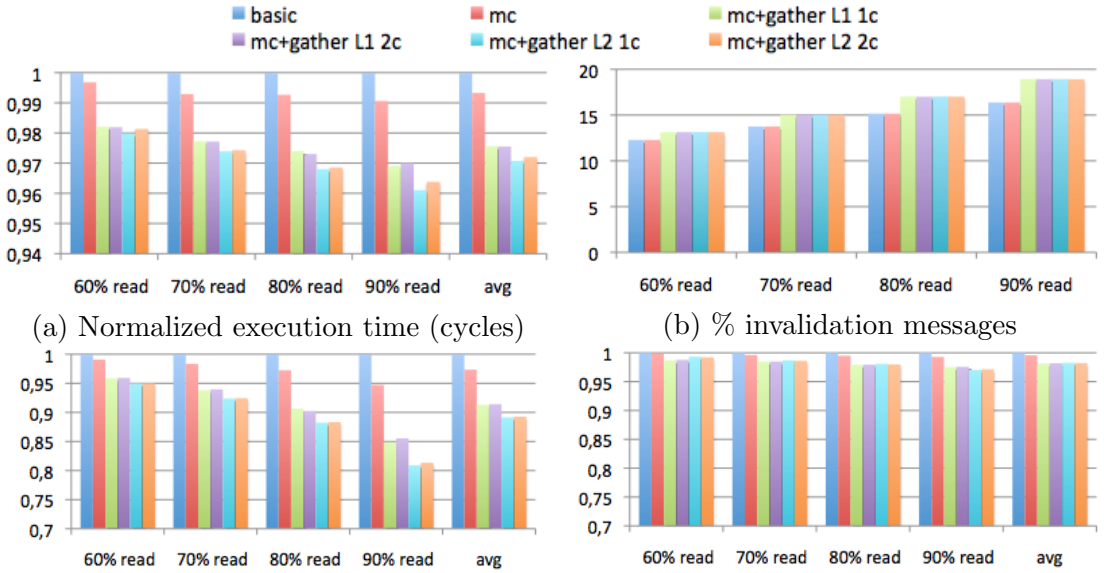
The first two versions of the protocol use the basic protocol described in Chapter 2, but they differ at NoC level. The first version does not use the multicast support (labelled as *basic*), while the second version (labelled as *mc*) uses the multicast support when sending invalidation messages [8]. The third and fourth versions use the multicast support to configure the SAGN network for the acknowledgements, and at protocol level they behave as explained in Chapter 4: the first one invalidates memory copies from the L2 node (labeled as *mc+g L2*) and the second one invalidates memory copies from the L1 node (labeled as *mc+g L1*). We evaluated the last two versions considering a gather network capable of delivering the signal in 1 and 2 cycles.

Each tile has two 64KB L1 banks (instruction and data) and a 512KB L2 bank. Tag access latency is set to 1 and 2 cycles respectively for L1 and L2 cache, while cache access latency is set to 2 and 4 cycles respectively for L1 and L2 cache.

Four sets of synthetic memory access traces have been generated and fed into the simulator. Each set is made of 200,000 random accesses to 500 different addresses. The sets differ in the percentage of read and write operations (from 60% read operations to 90% read operations).

Figure 4.1.a shows the execution time for each set, normalized to the case of the basic protocol. The multicast support alone (*mc*) slightly reduces the execution time as *inv* messages are sent with a single message and less contention is incurred in the network.

[2] In this work we have provided full support to the gNoCsim simulator for coherence protocol support and simple definition of the different protocol variants. This has been a large effort as the coherence protocols typically exhibit many race conditions that need to be solved with further refinements of the protocols.



(a) Normalized execution time (cycles)

(b) % invalidation messages

(c) Normalized store miss latency

(d) Normalized load miss latency

FIGURE 4.1: Different performance numbers. % of invalidation messages, average store and load miss latencies

With the alternative protocols and the control network, execution time is further reduced up to 4%, depending on the set of traces.

Figure 4.1.b shows the percentage of *inv* messages sent for each set (a multicast message sends multiple *inv* messages). As can be seen, the improvement in execution time is tightly coupled with the percentage of invalidations. The more invalidation messages are sent, the higher the benefits obtained by the control network. The percentage of invalidations grows with the percentage of read operations in the traces, since each write operation has to invalidate more sharers, so the performance improvement due to the gather network becomes more evident with traces with a high percentage of reads.

Figure 4.1.c shows the average store miss latency normalized to the base case (*basic* protocol). Again, the multicast support combined with the control network helps in lowering the store miss latency up to 20%. In particular, when the L1 nodes send the invalidation messages (*mc+g L1*), up to 15% reduction in write miss latency is achieved. When the L2 nodes take care of invalidations (*mc+g L2*) an extra reduction is achieved obtaining up to 20%.

Since the directory-based protocols only use the gather network to collect the acks in store misses, its effect on the load miss latency is negligible. This effect can be seen in Figure 4.1.d. When using the alternative protocols, the load miss latency is slightly

reduced (2%) with respect to the basic protocol. It should be noted also that the latency of the control network does not affect the results. Execution time, miss load latency, and miss store latency, are practically the same when the control network has a delay of one or two cycles.

Optimizing only one case out of the four exposed in Figures 2.2 and 2.3, the impact of the SAGN in systems which support a directory-based protocol is quite application-dependent: if the application generates a high percentage of write accesses on widely shared variables, the SAGN will be effective. To simulate actual applications on our system, we embedded gMemNoCsim in Graphite simulator [9] and launched various applications of the SPLASH-2 benchmark suite. Since all applications generated a very low percentage of write accesses on shared variables (0.4% of total L2 accesses on average), the effects of the SAGN were quite limited.

The SAGN is more effective if the system implements a broadcast-based protocol, which generates a higher amount of acknowledgment messages. As explained in Section 2.2, a broadcast message is issued in all cases except for the case of a read miss on a shared data, and each node except for the owner of the block must send an acknowledgement to the requestor, so *ack* messages are much more common in broadcast-based protocols than in directory-based. Indeed, the results we obtained running the SPLASH-2 applications on Graphite with the broadcast-based protocol show that the percentage of *acks* (indicated as Coherence Res in Figure 4.2) over the total number of messages is 30% on average, reaching 43% on Barnes, FFT and Water-nsquared.

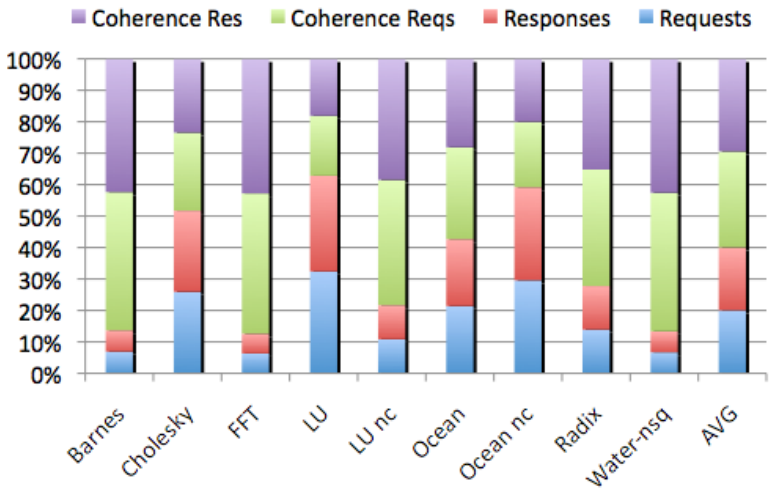


FIGURE 4.2: Normalized injected messages (Hammer protocol)

We evaluated the broadcast-based protocol with three different configurations: a basic configuration with no broadcast support and no SAGN (labelled as *Hammer*), a configuration with NoC-level broadcast support (labelled as *Hammer BC*) and a configuration with broadcast support and the SAGN (labelled as *Hammer BC GN*) and compared the performance of these three configurations with the basic directory protocol.

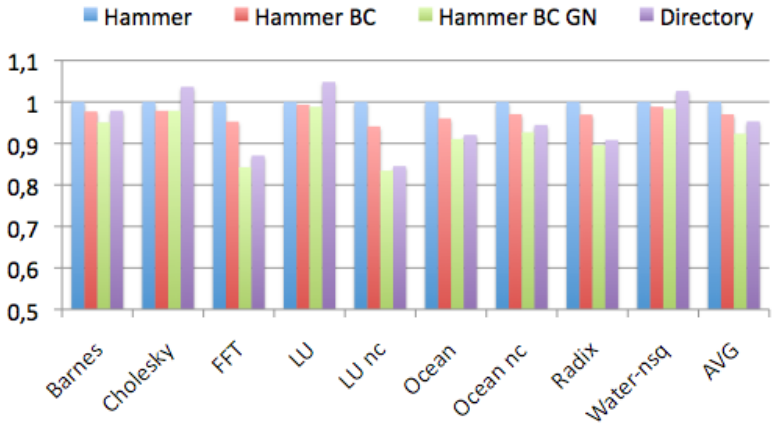


FIGURE 4.3: Normalized execution time (cycles)

Figure 4.3 shows the normalized execution time of the SPLASH-2 applications with the three configurations of Hammer protocol and the basic directory protocol. Without further support at network level, *Hammer* performs worse than *Directory* due to the amount of traffic generated at each L2 cache access and the serialization of *acks* at the requestor node's input ports. Adding broadcast support helps, but still the performance of *Directory* is better than those of *Hammer BC*. The SAGN further reduces the execution time, reaching an average execution time for *Hammer BC GN* which is 8% lower than *Hammer* and 3% lower than *Directory*.

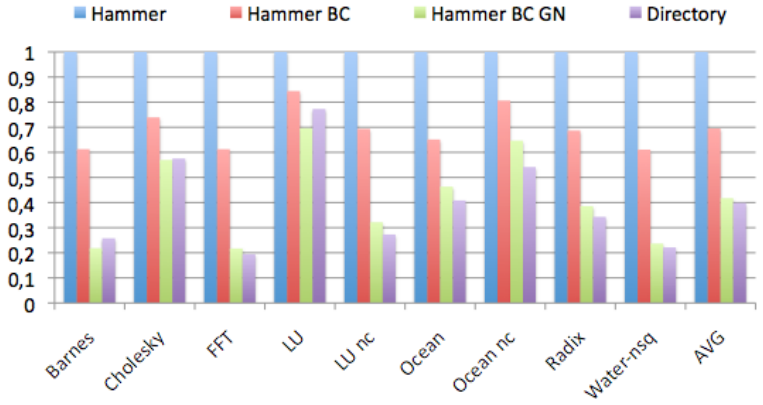


FIGURE 4.4: Normalized number of injected messages (SAGN signals are not included)

Network traffic is also drastically reduced. As shown in Figure 4.4, combining NoC broadcast support and the SAGN the number of injected messages in *Hammer* is reduced up to 60% on average and 80% for some applications. This means that *Hammer BC GN* reaches better performance than *directory*, clearing the area/traffic tradeoff: typically, directory-based protocols have a high area overhead (due to the sharing code) but generate low traffic, while broadcast-based protocols have a very low area overhead and generate much more traffic. The SAGN allows *Hammer BC GN* to overcome the performance of *Directory* with a lower chip area overhead an generating the same amount of traffic.

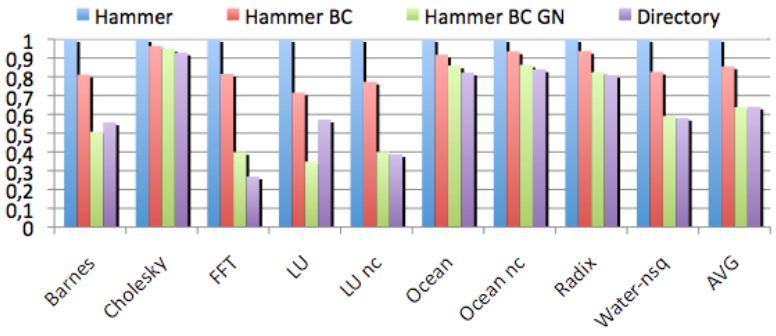


FIGURE 4.5: Normalized store miss latency (Hammer)

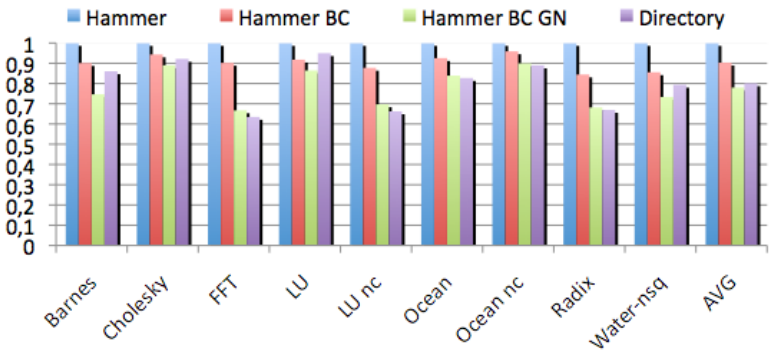


FIGURE 4.6: Normalized load miss latency (Hammer)

The impact of the SAGN on store and load miss latency when using Hammer protocol is higher than what we saw in Figures 4.1 b) and c) for Directory protocol. As shown in Figure 4.5, the combined effect of broadcast support and SAGN reduces the store miss latency of Hammer protocol by 40% on average (and up to more than 60% for some applications). The impact on load miss latency is lower, although still noticeable: as shown in Figure 4.6, the load miss latency is reduced by 20% on average, due to the fact that some read requests are managed by the L2 cache without broadcasting any message to L1 caches, as explained in Chapter 2.

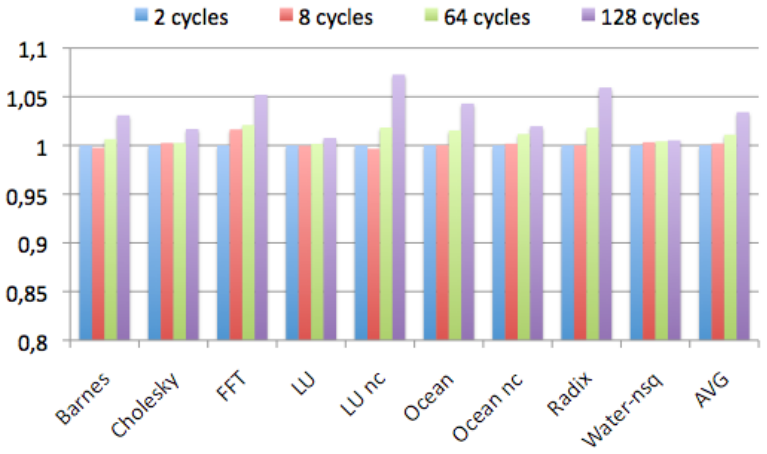


FIGURE 4.7: Normalized execution time with different SAGN delays

Figure 4.7 shows the impact of the SAGN delay on the system performance. We ran the SPLASH-2 applications on a system with *Hammer BC GN* and SAGNs with a delay ranging from 2 to 128 cycles. As shown, the performance is not significantly affected with delays up to 64 cycles: the average execution time increases by 1% on average.

So far we considered a SAGN with a delay of 2 cycles; as exposed in Chapter 3, this delay can be achieved with a combinational implementation, while the latency of a sequential SAGN would be higher (1 cycle per hop). As shown in Figure 4.7, this latency increment would not affect the performance, thus allowing to use a sequential SAGN. In Subsection 4.2.2 we provide a more detailed evaluation of ACK latencies when a sequential SAGN is used.

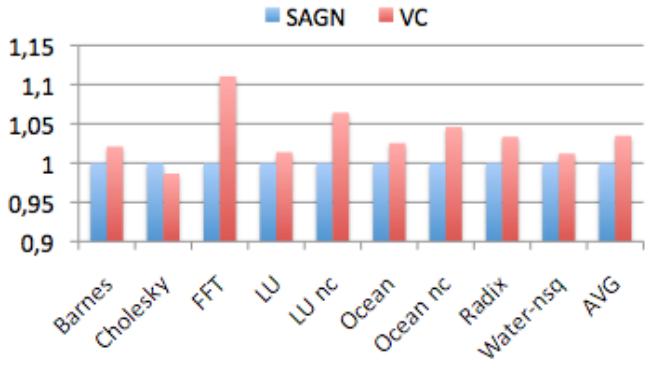


FIGURE 4.8: Normalized execution time compared to a NoC with an high priority VC for the ACKs

We conclude the evaluation of combinational SAGN comparing the performance of the SAGN with those of a system where a dedicated high-priority VC is used to transmit the ACKs. Figure 4.8 shows the normalized execution time when the SAGN and the

dedicated VC are used; SAGN has better performance than VC since it completely relieves the NoC from the large amount of traffic due to the ACKs. Notice that the VC configuration needs more switch resources contrary to the gather network solution as implementing buffering for the extra VC is costlier than implementing the logic gates of the gather network.

4.2.2 Sequential SAGN

In the previous section we assumed a combinational SAGN. Let's now compare its performance to that of a sequential implementation. As explained in Subsection 3.4, in the sequential implementation the ACKs are transmitted through the SAGN hop by hop and cycle by cycle, and the wiring is no more dedicated (in the combinational implementation each wire is dedicated to a subnetwork) so there could be contention if two different ACKs must be transmitted through the same output port of an SAGN module at the same cycle.

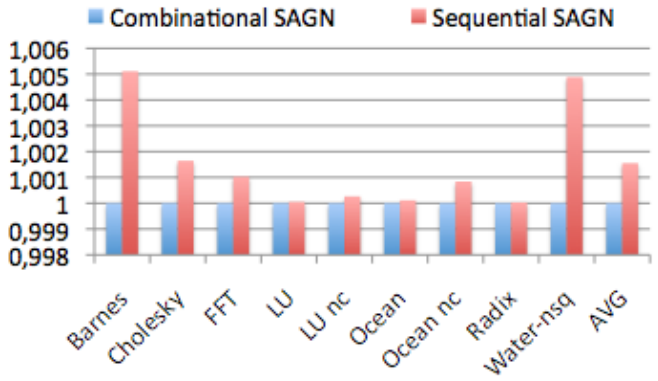


FIGURE 4.9: Normalized execution time with the two implementations of the SAGN

Figure 4.9 shows how the execution time is affected when the sequential SAGN is used. The increased latency in delivering the ACKs has a very low impact on overall performance: the execution time increases by a 0,15% on average and 0,5% in the worst case (Barnes an Water-nsq). For some applications no performance degradation can be noticed.

The increased execution time is due to conflicts in the SAGN modules and to the increased latency of ACKs. Figure 4.10 shows how many conflicts occur at the output ports of all SAGN modules for each ACK received at the destination node. The percentage of conflicts is quite low: on average, 25 conflicts occur during the transmission

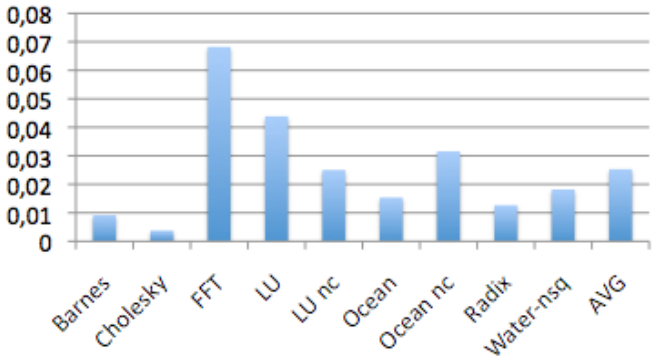


FIGURE 4.10: Number of conflicts per gather message received at destination node

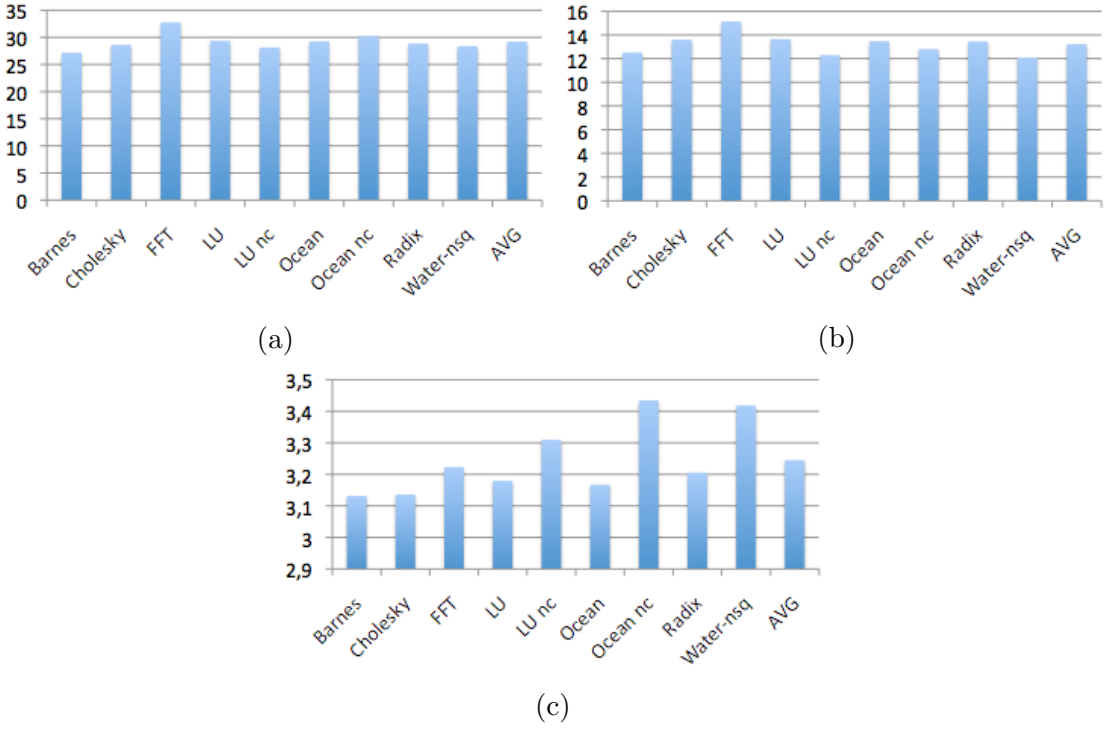


FIGURE 4.11: Average SAGN latency (sequential implementation)

of 1,000 ACKs. Notice that we are considering the mixed XY-YX mapping of Figure 3.5 to achieve a balanced distribution of ID transmissions through the different output ports of each SAGN module.

Figure 4.11 shows the average latency of ACKs with the sequential SAGN, measured in three different ways: Figure 4.11 (a) shows the elapsed time between the triggering of the first ACK and the reception of all the ACKs; Figure 4.11 (b) shows the elapsed time between the triggering of the last ACK and the reception of all the ACKs, and Figure 4.11 (c) shows the elapsed time between the triggering of the ACK by the node located farther from the requestor and the reception of all the ACKs. Notice that the

last node to trigger the ACK is not always the one that is located more distant to the requestor. The latter case therefore is the fittest to be compared to the combinational SAGN latencies shown in Subsection 4.1, since it represents the latency of the sequential SAGN in transmitting the ACK when the last signal is triggered. On average, the SAGN latency is thus increased to 3.2 clock cycles when the sequential implementation is used in a 4×4 cycles.

The moderate latency increment, which is well below the 128 cycles slack shown in Figure 4.7, and the low number of conflicts at each SAGN module lead to a negligible performance degration when using the sequential SAGN instead of the combinational one.

Chapter 5

Related Work

In an isolated design environment, where the NoC and the coherence protocol are designed separately, many optimization opportunities are lost. Cache coherence protocols have traditionally maintained a firm abstraction of the interconnection network fabric as a communication medium, thus disregarding the opportunities of optimizing the NoC at design time to efficiently deal with the coherence operations. More recently, however, some proposals exploring on-chip network optimizations for cache coherence protocols have appeared.

Cheng *et al.* [10] leveraged the heterogeneous interconnects available in the upper metal layers of a chip multiprocessor, mapping different coherence protocol messages onto wires of different widths and thicknesses, trading off their latency-bandwidth requirements. They achieved a good speedup with the two-level tree interconnect assumed in the paper but a moderate performance improvement with a 2D torus topology. Subsequently, Flores *et al.* [11] propose to combine a protocol-level technique (called *Reply Partitioning*) with the use of a simpler heterogeneous interconnect. Their work however is focused on directory-based protocols and their proposal has not been evaluated with the higher traffic generated by a broadcast-based protocol. Eisley *et al.* [12] propose in-network cache coherence, an implementation of the cache coherence protocol within the network based on embedding directories in each switch node that manage and steer requests towards nearby data copies. This approach enables in-transit optimization of memory access delay and shows better scalability than full-map directories stored in the

last-level caches; still, this proposal is based on directories, which limit the actual scalability of the system. In [13], it is presented a priority-based NoC, which differentiates between short control signals and long data messages to achieve a significant reduction in cache access delay. Additionally, the authors propose to use more efficient multicast and broadcast schemes instead of multiple unicast messages in order to implement the invalidation procedure and provide support for synchronization and mutual exclusion. The paper however does not address the problem of gathering multiple acknowledgements with the same destination node. Walter *et al.* [14] explore the benefits of adding a low-latency, customized shared bus as an integral part of the NoC architecture. The bus is used for some transactions such as broadcast of queries, fast delivery of control signals, and quick exchange of small data items. More recently, Vantrease *et al.* [15] advocate nanophotonic support for building high-performance simple atomic cache coherence protocols.

All previous proposals assumed a directory-based coherence protocol, which imposes a boundary to the scalability of the system due to the area overhead of the directory. On the other hand, broadcast-based cache coherence protocols can completely remove the important overhead that the directory structure would entail in a many-core CMP system. The AMD's Coherent HyperTransport (TH) [4] implements the Hammer broadcast-based protocol enabling the construction of small-scale multiprocessors. Subsequently, the HyperTransport Assist [16] developed by AMD for the 12-core AMD Opteron processor-based system code-named Magny Cours, added a directory cache to reduce the frequency of broadcasts, and therefore, to enable larger core counts. Also, the Intel's QuickPaht Interconnect (QPI) implements two different protocol modes [17]. In one of them (*source snoop* protocol mode) coherence transactions are broadcasted and every core must respond to the home with a *snoop response* that indicates the state of the block at that core. This mode allows for lower-latency coherence transactions at the expense of not scaling well. JETTY [18] and Blue Gene/P [19] are two proposals to filter the broadcast requests that would miss at destination nodes in order to reduce energy consumption due to cache look-ups. Filtering has also been proposed at source nodes [20], [21] to save energy and bandwidth. Agarwal et al. proposed to move the filters into the interconnect [22].

None of the filtering proposals however considered the possibility of a NoC support to

gather the acknowledgements. Proposals for efficient multicast support in on-chip networks have also appeared [23]. Additionally, it has been evaluated the case of using this kind of support in combination with a cache coherence protocol implementing imprecise directories (the Hammer protocol could be seen as using an inexact directory), demonstrating that multicast support is not enough to completely remove the performance degradation that the inexact sharing codes introduce [24].

Recently Krishna et al. [25] proposed a fabric that performs efficient forking and aggregation of messages. In their proposal ACKs are still transmitted as messages through the NoC and the switches are in charge to aggregate the ACKs ith the same destination node, which requires more complexity at the switches compared to the SAGN.

Chapter 6

Conclusions

In this work we presented a dedicated control network that allows fast recollecting of the acknowledgements to a multicast message sent on a CMP's on-chip network. We used this network to optimize the invalidation process of a MOESI invalidation-based cache coherency protocol. Two variations of the basic protocol have been evaluated, which differ on the source node of the invalidation messages: the home L2 cache or the L1 cache that issued the write request. We implemented the control network, which result to have a low area overhead and to need 2 clock cycles for a 4×4 network. Then, we implemented and simulated the coherency protocols. Simulation results show that by using the gather network the average store miss latency is reduced up to 20%, thus reducing the overall execution time. In addition, we have proposed a sequential implementation of the logic to reduce wire requirements. The solutions have been adapted to a Hammer protocol where broadcast operations are needed. Results demonstrate performance benefits of 8% in execution time with a reduction of network traffic up to 80%.

6.1 Current and Future Work

Current efforts aim to use the gather network to speedup the search phase of the home LLC bank in a system that employs dynamic mapping of the blocks to the LLC banks. In common systems this mapping is done statically, so when an access misses in the L1 cache the LLC bank to which a request has to be sent is known a priori. In case of dynamic mapping however, since the home bank may be any of the LLC banks, a search

phase is needed. One way to implement this search phase is to broadcast the request to all LLC banks and wait for all banks to answer, both with an acknowledgement or with the requested data. In this case too, the SAGN can be used to speedup the collection of acknowledgment messages and reduce NoC traffic, enabling efficient implementation of mapping policies that reduce the hop distance between an L1 requestor and the LLC home bank.

Other solutions that can be explored are the use of the SAGN to deliver unicast acknowledgements (e.g. writeback acknowledgements that are sent from the LLC to an L1 after a writeback) or to efficiently implement synchronization primitives (e.g. barriers) at hardware level.

6.2 Publications

The following papers related with this work were submitted and accepted for publication in different international conferences and journals:

- M. Lodde and J. Flich, "Memory Hierarchy and Network Co-design through Trace-Driven Simulation", Proc. of the 7th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems. July 2012.
- M. Lodde, T. Roca, and 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, January 2012.
- M. Lodde, J. Flich, and M. Acacio, Heterogeneous noc design for efficient broadcast-based coherence protocol support, in Proc. of the 6th Intl, Symposium on Networks on Chip (NOCS), May 2012.
- M. Lodde, T. Roca, and J. Flich, Built-In Fast Gather Control Network for Efficient Support of Coherence Protocols", to appear in IET Computers and Digital Techniques INA-OCMC 2012 Special Issue.

In addition, also in national conferences some related papers have been published:

- M. Lodde and J. Flich, "Memory Hierarchy and Network Co-design through Trace-Driven Simulation", XXII Jornadas de Paralelismo (pages 571-578). Tenerife, Spain. 7-9 September 2011.

- M. Lodde, J. Flich, M.E. Acacio, "A NoC-Level Support for Broadcast-Based Coherence Protocols", XXIII Jornadas de Paralelismo (pages 440-454). Elche, Spain. 19-21 September 2012.

References

[1] Jose Flich and Davide Bertozzi. *Designing Network On-Chip Architectures in the Nanoscale Era*. Chapman & Hall, 2010.

[2] D.J. Sorin, M.D. Hill, and D.A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011.

[3] M.M.K. Martin, M.D. Hill, and D.J. Sorin. Why on-chip cache coherence is here to stay. *Duke University Department of ECE Technical Report TR-2011-1*, August 2011.

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

[5] 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.

[6] E.S. Shin, III Mooney, V.J., and G.F. Riley. Round-robin arbiter design and generation. In *System Synthesis, 2002. 15th International Symposium on*, pages 243 –248, oct. 2002.

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

[8] Samuel Rodrigo, Jose Flich, Jose Duato, and Mark Hummel. Efficient unicast and multicast support for cmps. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 364–375, 2008.

[9] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –12, jan. 2010.

[10] Liqun Cheng, Naveen Muralimanohar, Karthik Ramani, Rajeev Balasubramonian, and John B. Carter. Interconnect-aware coherence protocols for chip multiprocessors. *SIGARCH Comput. Archit. News*, 34(2):339–351, May 2006.

[11] A. Flores, J.L. Aragon, and M.E. Acacio. Heterogeneous interconnects for energy-efficient message management in cmps. *IEEE Transactions on Computers*, 59(1): 16–28, January 2010.

[12] Noel Eisley, Li-Shiuan Peh, and Li Shang. In-network cache coherence. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 321–332, 2006.

[13] Evgeny Bolotin, Zvika Guz, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. The power of priority: Noc based distributed cache coherency. In *Proceedings of the First International Symposium on Networks-on-Chip*, NOCS '07, pages 117–126, 2007.

[14] I. Walter, I. Cidon, and A. Kolodny. Benoc: A bus-enhanced network on-chip for a power efficient cmp. *IEEE Computer Architecture Letters*, 7(2):61–64, July-Dec .2008.

[15] D. Vantrease, M. Lipasti, and N. Binkert. Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols. In *In Proc. of the 17th International Symposium on High Performance Computer Architecture*, pages 132–143, February 2011.

[16] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30 (2):16–29, March/April 2010.

[17] R.A. Maddox, G. Singh, and R.J. Safranek. Weaving high performance multiprocessor fabric: Architecture insights into the intel quickpath interconnect. *Intel Press*, 2009.

[18] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. Jetty: filtering snoops for reduced energy consumption in smp servers. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 85 –96, 2001.

[19] V. Salapura, M. Blumrich, and A. Gara. Design and implementation of the blue gene/p snoop filter. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 5 –14, feb. 2008.

[20] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. *SIGARCH Comput. Archit. News*, 33(2):246–257, May 2005.

[21] A. Moshovos. Regionscout: exploiting coarse grain sharing in snoop-based coherence. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 234 – 245, june 2005.

[22] N. Agarwal, Li-Shiuan Peh, and N.K. Jha. In-network coherence filtering: Snoopy coherence without broadcasts. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 232 –243, dec. 2009.

[23] Samuel Rodrigo, Jose Flich, Jose Duato, and Mark Hummel. Efficient unicast and multicast support for cmps. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 364–375, 2008.

[24] A. Ros and M.E. Acacio. Evaluation of low-overhead organizations for the directory in future many-core cmps. In *Proc. of the 4th Workshop on Highly Parallel Processing on a Chip*, pages 87 – 97, September 2010.

[25] Tushar Krishna, Li-Shiuan Peh, Bradford M. Beckmann, and Steven K. Reinhardt. Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 71–82, 2011.