Energy-Efficient Cache Coherence Protocol for NoC-based MPSoCs

Tales M. Chaves, Everton A. Carara, Fernando G. Moraes
PUCRS – FACIN – Av. Ipiranga 6681 – Porto Alegre – 90619-900 – Brazil
tales.chaves@acad.pucrs.br, everton.carara@pucrs.br, fernando.moraes@pucrs.br

ABSTRACT
As the number of cores and functionalities integrated in embedded devices increases, the amount of memory used on these devices also increases, justifying the development of memory architectures presenting scalability, low energy consumption and low latency. To implement memory solutions, most works adopting NoC-based MPSoCs only employ basic communication services, such as send/receive, without exploring the services NoCs can offer, for instance connection, priorities and multicast communication. Multicast can be used to optimize the cache coherence protocol, leading to both traffic and energy consumption reduction. The goal of this work is to optimize a directory-based cache coherence protocol exploiting specific NoC services, as multicast and priorities. To demonstrate our proposal, an MPSoC described at the RTL level is used, enabling accurate performance and energy evaluation. Results show a reduction of 17% in the number of clock cycles and a reduction up to 86% (average reduction: 39%) in energy consumption for some memory transactions.

Categories and Subject Descriptors
B.3.2 [**Hardware**]: Design styles – *cache memories.*

General Terms
Algorithms, Performance, Design, Experimentation.

Keywords
MPSoC, NoC, memory hierarchy, caches

1. INTRODUCTION
The increasing number of functionalities and applications of current embedded systems raise the demand for processing, high-speed communication and memory. According to Wolf et al. [1], the most critical component that determines the success of an MPSoC (Multiprocessor System-on-Chip) is its memory architecture.

In the general purpose computing field, the use of cache memories consists of a simple and efficient way to increase software performance [2]. According to [3], caches are and will be one of the best solutions to achieve low latency accesses to data and instructions. The introduction of cache memories reduces the average access latency, since most memory accesses are local. Consequently, the amount of memory transactions issued by processors to the communication infrastructure is reduced, and the performance is increased. One of the disadvantages of using cache memories is the unpredictability introduced for memory accesses. The amount of cycles that a memory operation takes to complete depends on the location of the data being accessed. Thus, designs that have to meet hard real-time deadlines should adopt alternative solutions such as scratchpad memories, which guarantee one-cycle latency.

In addition, cache memories increase the energy overhead. Cache misses are the main source of energy consumption, as several operations might be necessary to bring a block to the cache memory. During this process, cache coherence protocols act to enforce system coherence and, therefore, their energy consumption must be reduced.

Cache coherence protocols, such as invalidate-based ones, might generate several unicast messages on the NoC in order to keep the system coherent. For instance, whenever a line needs to be invalidated in several L1 caches, the L2 cache sends several invalidation messages. The resulting traffic generated in the NoC increases significantly the energy consumption of the MPSoC. An approach that helps reducing the energy overhead is to send a multicast message, targeting all caches that contain the entry to be invalidated. This reduces the traffic and consequently the energy consumption.

The goal of this work is to explore NoC services such as multicast and priorities, and some properties of the NoC, such as duplicated physical links, to optimize a directory-based cache coherence protocol. Exposing the low-level NoC to the protocol allows the reduction of energy consumption and latency of the protocol.

The rest of this paper is organized as follows. Section 2 presents related works. Section 3 details the reference MPSoC architecture adopted by this work. Section 4 details the cache coherence protocol. Section 5 presents the experimental setup and results. Section 6 concludes this paper.

2. RELATED WORK
MPSoCs are commonly employed as the architecture for embedded devices, which require low energy consumption. According to [4], the reduction of both miss latency and traffic generated by the cache coherence protocol are conflicting factors. Protocols designed to decrease miss latency usually generate more network traffic than protocols designed to consume low energy. The increase in network traffic also increases power consumption [5]. This consumption might approach 50% of the overall chip power in some cases. Considering that MPSoCs designs mainly target low energy consumption, there is a necessity of the study and development of energy efficient protocols.

A few works on the literature explore the physical services, which are provided by the NoC to optimize the cache coherence

SBCCI'11, August 30-September 2, 2011, João Pessoa, Brazil.

protocol. Examples of NoC-based MPSoCs adopting memory hierarchy with caches includes [6][7][8], having as common feature the abstraction of the communication infrastructure, adopting only send and receive services.

Bolotin et al. [9] attributes different priorities to packets transmitted by the cache coherence protocol. Operations such as read and exclusivity request (short data packets) are transmitted using high priority. Long packets, such as packets containing data to be written in the memory or a block just read from the memory are transmitted using low priority. This strategy increases performance, but does not addresses energy issues.

Barroso et al. [10] propose optimizations to an invalidation-based directory protocol. One of the proposed optimizations is named clean-exclusive, in which an exclusive copy of a block is returned to a read if there are no sharers. An important enhancement presented by this work is the avoidance of NAK messages (negative acknowledgement). This is possible due to the use of 3 virtual channels (I/O, L, H). L channel is a low priority channel and H is a high priority channel. Also, to decrease the traffic in the network, a technique called cruise-missile-invalidates is used for sending a unique invalidation message to several nodes.

Jarger et al. [11] implement a cache-coherence protocol, named Virtual Tree Coherence (VTC). It is based on a virtual ordered interconnection tree, which keeps a history of nodes sharing a common region of memory. For each region, a virtual tree is created containing the nodes that share that region. Every time one of the nodes accesses a given region, a request is sent to the root of the tree, which in turn, requests the data to the node holding the most updated copy. This request is done through a multicast message that traverses the tree.

This paper explores the optimization of a well-known cache coherence protocol, by exploring the physical services offered by the NoC. Most works focus on a specific service provided by the NoC, such as priority, while we propose to use a combination of services and NoC features to optimize the cache coherence protocol. Additionally, few works provide RTL implementation of the proposed solutions. The present work adopts a RTL model for the MPSoC, resulting in accurate performance results.

3. MPSOC ARCHITECTURE

This work adopts as reference a homogeneous NoC-based MPSoC [13], described in synthesizable VHDL. Each PE contains a MIPS-like processor (Plasma), a local memory (RAM), a DMA controller, a Network Interface and a L1 data cache. The MPSoC can also have L2 data cache banks connected to a given node of the NoC. Figure 1 shows an architectural view of a 2x2 instance of the HeMPs Platform.

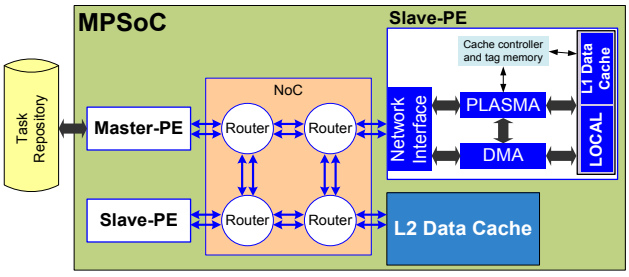


Figure 1 - MPSoC with a two level memory hierarchy – shared memory and caches (only Slave-PEs have caches).

Two types of PEs are used: slave and master. Slave-PEs are responsible for executing application tasks. The Master-PE is responsible for mapping tasks into the Slave-PEs. The Master-PE can also receive debug messages from Slave-PEs, transmitting them to an external host through an Ethernet interface.

The external memory, named *task repository*, contain the object code of all application tasks. Tasks are described in C language. Each Slave-PE runs a tiny operating system (named microkernel whose memory footprint is around 20 KB), responsible for managing task execution and task communication. This microkernel is a preemptive operating system where each task uses the CPU for a pre-defined period.

The network interface and DMA modules are responsible for sending and receiving packets, while the processor performs computation tasks and the management of NI and DMA modules. The local RAM is a true dual port memory allowing simultaneous processor and DMA accesses. This local memory is organized in a parameterizable number of pages. The microkernel uses the first pages and application tasks use the remaining ones.

3.1 NoC

The 2D mesh NoC used in the MPSoC has the following features to support QoS: (*i*) Hamiltonian routing, which enables multicasting; (*ii*) packet and circuit switching; (*iii*) priorities; (*iv*) duplicated physical channels (two 16-bit bidirectional links).

Each router of the NoC is labeled with a number, from 0 to N-1, where N is the number of routers. The NoC is logically divided into two acyclic and disjoint, sub-networks. In one sub-network, the routers are numbered in decreasing order, whilst in the other; routers are numbered in increasing order. Each packet is routed according to the Hamiltonian algorithm, traversing only one sub-network. For instance, when router 2 sends a packet to router 6, the increasing sub-network is used. In the other hand, when the router 6 sends a packet to router 2, the decreasing sub-network is used.

The dual-path multicast algorithm is used to transmit multicast messages. In this algorithm, the header of the packet contains all target routers for that packet. When a multicast message is sent, at most two packets are transmitted. The first packet is transmitted to all PEs with addresses higher than the multicast source address. The second one is transmitted to all PEs with addresses lower than the source PE. If all targets have either higher addresses or lower addresses than the multicast source, only one packet is sent.

There are two levels of priorities physically implemented in the NoC: high and low. High priority packets may be transmitted in both physical channels, while one of the channels only transmits low priority packets. The use of priorities allows the distinction of long and short packets, so that one physical channel might be used for sending short packets only, whilst the other transmits long packets only, which increases memory throughput.

3.2 Memory Architecture

The MPSoC has a two-level memory hierarchy, as illustrated in Figure 1. Only Slave-PEs support L1 data caches. In Slave-PEs, the address space is divided into two disjoint address spaces: one address space is used for accessing data from the cache memory and the other is used for the local memory. The local memory stores the microkernel, object code of tasks received from the Master-PE and private variables. It can be seen as a scratchpad memory because it guarantees one-cycle latency for every access, which never results in miss. The L1 cache memory stores temporary copies of blocks from a shared L2 cache memory, which is used only for storing data.

A cache controller and a tag memory (right upper corner of Figure

1) compose the L1 cache. The cache adopts the direct mapping scheme, due to the minimum hardware support required to implement it. The tag memory stores for each line: (i) tag, the most significant bits of the block address stored in that line; (ii) valid bit, informs if the line is valid; (iii) modified bit, informs that the line was modified. Each line stores a full block of the L2 cache, which size is 128 32-bit words.

The L2 cache is shared by all PEs. It is composed by: (i) a memory controller, responsible for executing read/write operations according to the cache-coherence protocol; (ii) a directory memory, which stores for each block its status (modified, shared, invalid or transition) and the addresses of PEs holding a copy of the block; (iii) a network interface (NI); (iv) the memory bank, logically divided into blocks of 128 32-bit words. Note that more than one L2 cache bank can be used in the system, resulting in a distributed shared memory (DSM) organization. In the scope of this work, only one bank is used.

All necessary instructions to execute a given task are stored in the local memory. The L2 cache is exclusively used for storing shared data. Therefore, there is no traffic in the network due to fetching instructions from L2 cache to PEs. One of the advantages of this memory hierarchy is the smaller traffic inside the NoC. In addition, the native reference MPSoC communication mechanism, message passing, is extended to shared memory communication.

The priority mechanism included in the NoC enables the transmission of short memory requests using high priority. Memory responses, which contain data from a block, are transmitted using low priority. The NI of the L2 cache interfaces has two buffered input ports, one for each channel of the NoC (port 0 and port 1). The output ports of the NI are not buffered. The advantage of using two physical channels avoids the contention of the L2 cache for long periods, such as in situations when the L2 cache is receiving a write-back block. In this situation, the other channel is free to receive and buffer service requests that will be handled as soon as the write operation ends.

3.3 Cache Coherence Protocol

According to the MSI cache-coherence protocol, any shared block can be in 3 states: *modified* – a copy of the block has been modified, therefore the L2 cache does not contain a valid entry of that block; *shared* – zero or several caches might contain an identical copy of the block which is stored in the L2 cache; *invalid* – block data is not valid. In addition to the three states of the MSI protocol, we propose the creation of the transition state (T). This state indicates that a given block is not updated in the L2 cache, but a write-back request has been already issued to the L1 cache which has modified the block. Requests that arrive at the L2 cache when the block is in T state are forwarded to the L1 cache that has modified the block. Section IV presents a more detailed explanation of the benefits of the transition state.

Our work adopts a hybrid implementation of the protocol, being part of it implemented in hardware (in the cache controller) and part in software (in the microkernel). This design choice simplifies the hardware implementation because some mechanisms, such as DMA programming, are handled in software, by the microkernel.

The cache controller is responsible for: (i) detecting and signaling hit/miss when the address accessing the cache changes; (ii) updating the tag memory; (iii) executing read and write operations. The microkernel is responsible for: (i) exchanging messages with the shared memory; (ii) replacing blocks when necessary; (iii) handling write-back operations.

4. CACHE COHERENCE PROTOCOL OPTIMIZATIONS

4.1 Invalidating cache lines

Whenever a PE needs to modify a given block, the L2 cache must invalidate all valid entries of this block to prevent cache incoherence. An invalidation message is then sent to every PE sharing this block. Finally, the L2 authorizes the modification of the block by the requesting PE.

In unicast-only NoCs, a unicast packet must be sent for each PE. Figure 2(a) shows a scenario where PE01 requests exclusivity of a block, which is in shared state. Two other PEs are currently holding a copy of this block (PE02 and PE03). Therefore, an invalidation message is sent to PE 02 and PE 03. Using unicast messages, the traffic generated on the NoC in this case increases according to the number of PEs sharing the block. Figure 2(b) shows a scenario where multicast is exploited. In this case, the L2 cache issues a multicast message targeting several processors, reducing network traffic. The traffic reduction decreases the switching activity of the routers, therefore reducing energy consumption.

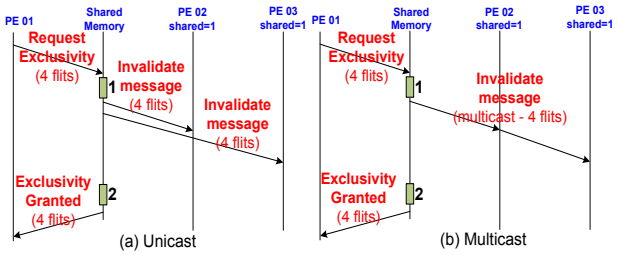


Figure 2 - Sequence diagram for a request of exclusivity on a shared block.

4.2 Read request optimization

The use of multicast messages might optimize a read operation on a block that is in modified state. The non-optimized operation occurs as shown in Figure 3(a). After receiving the modified block from PE02 (event 3), the L2 cache first writes the block into the memory bank, and then sends a copy of it to the requesting PE (PE01). In the optimized operation, the PE containing the modified block (PE02) sends a multicast message to both the requesting PE (PE01) and the L2 cache.

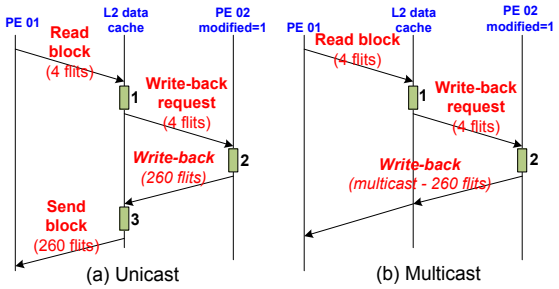


Figure 3 - Sequence diagram for a read operation of a modified block.

4.3 Write request optimization

To write on a block, the processor must read it beforehand. If the block that a given PE wants to modify is already in modified state, the PE holding the modified copy must execute a write-back

operation. Suppose PE01 wants to write on a modified block, only cached by PE02. PE01 sends a read-with-exclusivity request to the L2 cache, which, in the non-optimized implementation (Figure 4(a)), sends a write-back request to PE02. After receiving the write-back response, the L2 cache sends a copy of the block to PE01 and updates the directory.

In the optimized implementation (Figure 4(b)), after receiving the read-with-exclusivity request from PE01, the L2 cache updates the directory, setting PE01 as the holder of the modified copy of the block. Then, it sends a special write-back to PE02, which will send a copy of the block to PE01, and invalidate its copy of the block. Additional requests for this block may arrive at the L2 cache before finishing this operation. To ensure sequential consistency, these operations must be blocked at PE01 until it finishes the operation on this block.

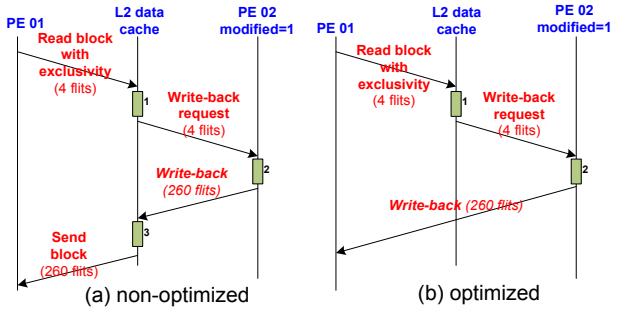


Figure 4 - Sequence diagram for a write-back after a write request.

4.4 The Transition State

The benefit of having a new state in the cache coherence protocol is the possibility of decreasing latency in some cases. This gain is achieved when forwarding requests that arrive at the L2 cache to the L1 cache that has an updated copy of the block being requested.

Figure 5 presents the MPSoC configuration that will be used to illustrate a scenario that explains the operation of the transition state. Suppose PE03 holds a given block in modified state, and PE00 wants to read this block. A read miss occurs, resulting in a read request to the L2 cache (event 1 in Figure 6). When the L2 cache receives the request of PE00, it searches for the address of PE whose L1 cache holds the block in modified state. This search is performed in the local directory of the L2 cache. A write-back request is issued to PE03 (event 2), setting the block from M (modified) to T (transition) state. Next, if another request arrives at the L2 cache, such as from PE05, for instance (event 3), instead of blocking the request until the block is updated, the L2 cache issues a read request of this block to PE03 (step 4). PE03 then sends a packet containing a copy of the cached block of its local cache to PE05 (step 7). This reduces the load of the L2 cache.

This optimization is possible because, although after writing-back the block to the L2 cache and PE00 (events 5 and 6), PE03 still has a valid copy of that block. Therefore, it might serve a copy of the block to PE05. When the L2 cache receives the write-back packet (event 5) it sets the block as shared.

Without the transition state, the standard coherence protocol would have to buffer in the L2 cache the read request from PE05 and wait for: (i) the arrival of the write-back packet in the L2 cache; and (ii) the update of the block in the L2 cache. In the proposed optimization, the PE holding the modified block (PE03) sends the block directly to PE05 just after finishing the write-back operation. This optimization tends to reduce the number of cycles required to send a copy of the block to the second PE which requested the read, as it does not require the read request to be block on the L2 cache until the finish of the write-back operation.

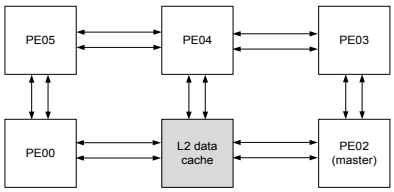


Figure 5 - MPSoC configuration: 3x2 NoC, 5 PEs (1 master, 4 slaves) and 1 L2 cache.

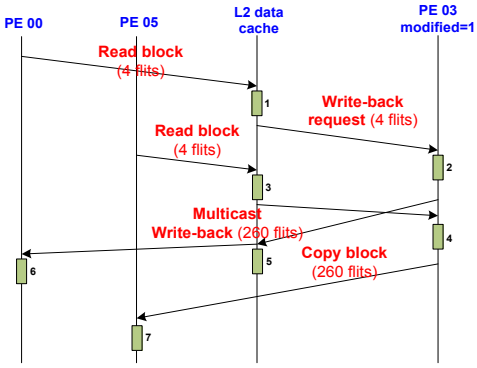


Figure 6 - Sequence diagram for the T state.

5. EXPERIMENTS

This section presents experiments performed to compare the optimized cache-coherence protocol presented in Section 4, against a non-optimized MSI directory-based protocol.

5.1 Experimental Setup

Two different implementations of an MPSoC platform were simulated in RTL-level using the ModelSim simulator. The platform used as a case study is configured as: 5x5 NoC mesh topology, containing 24 PEs (1 master and 23 slaves) and 1 L2 cache bank. The first implementation, named OPT, employs the four optimizations described in Section 4. The second implementation, named NO-OPT, adopts a standard MSI directory-based protocol based on unicast messages only. In all experiments, the results evaluate the number of clock cycles, and the energy spent in communication between the PEs and the L2 cache.

The packets containing memory operations are generated by application tasks. To prevent the programmer from dealing with low-level aspects related to the memory architecture, a cache library was developed. Figure 7 shows an example of application, which uses two functions of the cache library: *cache_read_block_word*, which reads a word of a block (indicated by number 1 in Figure 7); *cache_write_block_word*, which writes to a given word of a shared block (indicated by number 2 in Figure 7). In the current example, the value 0x44 is written to the first word of block 1.

To evaluate the consumed energy per memory transaction, the present work adopts the volume-based energy model proposed by Hu et al. [12]. Equation 1 computes the communication energy spent to transit *1 bit* through a distance of *n* hops.

$$E_{bit}^{hops} = n_{hops} * E_{S_{bit}} + (n_{hops} - 1) * E_{L_{bit}} \quad (1)$$

In Equation 1: E_{Sbit} (20.58 pJ/flit), E_{Lbit} (2.84 pJ/flit) and n_{hops} correspond to the energy consumption of the router, in the interconnection wires and the number of hops to transmit 1 flit, respectively (65 nm technology).

```
#include <task.h>
#include <stdlib.h>
#include <cache.h>          ← Cache Libary

int main() {

    int i, word;

    Echo("Test Cache Memory - task A started.");

(1) word = cache_read_block_word(1, 0);

    /* Simulates that computation is performed */
    for (i=0; i<300; i++);

(2) cache_write_block_word(1, 0, 0x44);

    return 0;
}
```

Figure 7 - Example of an application reading and writing to the shared memory.

The energy model was calibrated using the ST/IBM CMOS 65 nm technology at 1.0 V, adopting clock-gating, 100 MHz clock frequency and injection rate of 10% of the available link bandwidth. The PrimePower tool generates the power and energy values used in Equation 1.

5.2 Invalidating cache lines

In situations where more than one cache is sharing the same block of the L2 cache, the memory controller needs to send invalidation messages to invalidate these copies before granting exclusivity to a PE. To evaluate the benefits of using multicast to propagate these messages, the number of caches sharing a copy of the same L2 block varies. Table 1 shows the number of clock cycles required to send invalidation messages to 3, 5 and 8 caches, respectively. Although with a smaller number of targets to invalidate, the first scenario (3 caches sharing a block) presents higher gain compared to the non-optimized implementation. This is due to the task mapping on the platform, which allowed the sending of only one multicast message (see Section 3.1), which significantly reduces the amount of data transmitted on the NoC. For the other scenarios (5 and 8 caches sharing a block), the use of multicast messages saves energy and improves performance at most 17.53%.

Table 1 – Number of clock cycles and energy consumption of invalidate messages depending on the number of caches sharing a block.

	Platform	*3 caches*	*5 caches*	*8 caches*
Energy (pJ)	NO-OPT	1635	2584	3798
	OPT	685	2073	2916
	OPT gain vs NO-OPT	58.07%	19.76%	23.20%
Clock Cycles	NO-OPT	141	154	147
	OPT	129	127	129
	OPT Gain vs NO-OPT	8.51%	17.53%	12.24%

5.3 Read request optimization

To evaluate the read optimization, a task after a cache-miss, must issue a read request to a modified block. Upon receiving the request, the L2 cache issues a write-back request to the PE, which holds exclusivity on the block being requested. In the OPT implementation, after receiving the write-back request, the PE sends a multicast message containing a copy of the block, both to the L2 cache and to the requesting PE.

The experiments varied the distance, in hops, between the PE reading the block and the L2 cache. Figure 8 presents the results. The average energy reduction offered by the optimization is 12%. However, the NO-OPT implementation is slightly faster than the OPT implementation (in average 30 clock cycles), due to the higher complexity to treat multicast packets at each router, and the non-minimal path taken by these packets.

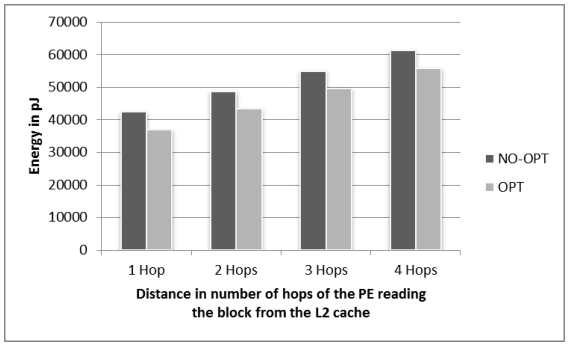


Figure 8 - Energy consumption of the read operation on a modified block as the number of hops increases.

5.4 Write request optimization

To evaluate the write optimization, a task after a write cache-miss, must issue a read with exclusivity request to a modified block. Upon receiving the request, the L2 cache issues a write-back request to the PE, which holds the modified copy of the block being requested. In the OPT implementation, after receiving the write-back request, the PE sends a unicast message containing a copy of the block, only to the requesting PE, bypassing the L2 cache. To evaluate this optimization, the placement of the L2 cache is defined in Figure 11(a). The PE holding the modified copy of the block is fixed at PE00. The evaluated scenarios varied the position of the block writing in the cache.

Figure 9 shows that there is an average reduction of 17% in the number of cycles required to finish the write operation. Also, Figure 10 shows that there is a reduction of up to 86.8% on the energy spent during this operation by the OPT implementation over the NO-OPT. The reason of this significant reduction is that long messages, containing data blocks, are transmitted only once, from PE to PE. The memory can be bypassed because its copy of the block would be altered right after.

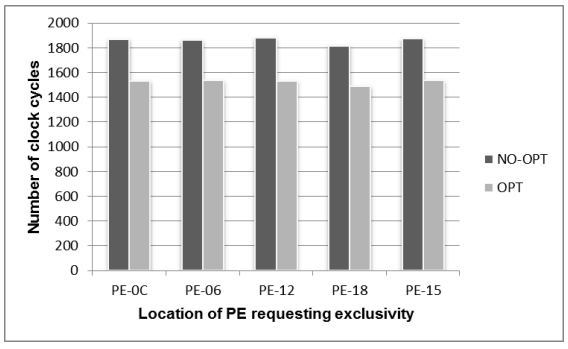


Figure 9 - Number of cycles required to execute a read operation on a modified block varying the location of the modified block.

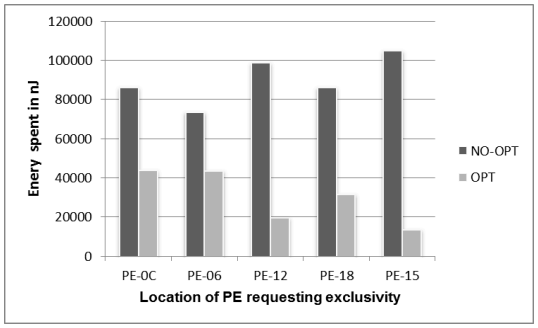
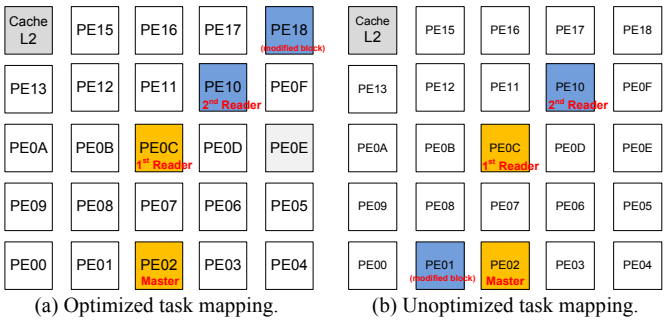


Figure 10 - Energy consumed to execute a read operation on a modified block varying the location of the modified block.

5.5 The Transition state

To evaluate the addition of the transition state, a scenario where 2 PEs issue subsequent reads to a modified block of the L2 cache is analyzed (this optimization was presented in Section 4.4). The first PE that issues a read request will benefit from the Read request optimization, whilst the second PE will benefit from the addition of the T state. The NoC feature enabling this optimization is the duplicated physical channels, because while the L2 cache controller monitors of the channels waiting for a write-back packet, the other channel can receive requests, such as a read request.

The results show that the gains against the standard MSI protocol, in this case, are sensitive to the task and L2 cache mapping. In scenarios where the PE that issues the second read request is closer to the PE previously holding the modified copy of the block, there are gains both in performance of the protocol (decrease in clock cycles) and also a save on the energy spent during the operation. Figure 11(a) shows a scenario where PE18 holds the modified copy of the block being accessed, PE10 is the second reader and the L2 cache is located at the upper left corner of the platform. In this case, the second read operation consumes, in the OPT version of the platform 19.035 pJ, against 42.893 for the NO-OPT version. This represents a 55% decrease in energy consumption. The number of clock cycles required is decreased by 7%. In scenarios where the PE that issues the second read request is closer to the L2 cache, the addition of the T state increases the number of cycles, and the consumed energy. In Figure 11(b), the second reader is mapped on PE10, PE01 holds the modified copy being accessed and the L2 cache is located at the upper left corner. For this case, the energy consumed during this operation by the OPT implementation is 37.583 pJ, against 30.621 for the NO-OPT. It represents a 22% increase on the energy consumed. The number of clock cycles is increased by 5%.



(a) Optimized task mapping. (b) Unoptimized task mapping.

Figure 11 –Task mappings for the T state optimization.

To reduce energy consumption for all scenarios, this optimization must be activated dynamically according to the task mapping. Upon receiving a read request, a module of the L2 cache calculates the *Manhattan distance* between PEs (PE reading and PE holding the modified block) and L2 cache, and chooses if it is best to use the T state optimization or block the request until finishing the write-back operation for this block.

6. CONCLUSIONS

This work is one of the first attempts to explore the benefits NoCs can bring to cache-coherence protocols, evaluating a complete system at the RTL level (PEs and the NoC), including the software (microkernel and applications) running on top of it.

By using the proposed protocol optimizations, results show that it is possible to reduce the energy consumed by the operations up to 86.8% (average reduction: 39%) and to achieve an improvement of 17.53% in the execution time (clock cycles). All optimizations, except the Transitions state, always reported energy reduction. The Transition state optimization is sensible to the task mapping.

Future works include: (*i*) couple the proposed techniques to mapping heuristics that consider the memory position; (*ii*) data migration policies to optimize the memory performance; (*iii*) use of parallel benchmarks to characterize performance; (*iv*) extend the number of memory-IPs in the memory hierarchy, resulting in a DSM architecture; (*v*) study and implement a way to distribute (or to hierarchize) the directory used by the cache coherence protocol.

7. ACKNOWLEDGMENTS

The Authors acknowledge the support of CNPq and FAPERGS, projects 301599/2009-2 and 10/0814-9, respectively.

8. REFERENCES

[1] Wolf, W.; Jerraya, A. A.; Martin, G. “Multiprocessor System-on-Chip (MPSoC) Technology”. IEEE Transactions on Computer-Aided Design of ICs and Systems, vol. 27(10), pp.1701-1713, 2008.

[2] Petrot, F., Greiner, A., and Gomez, P. “On Cache Coherency and Memory Consistency Issues in NoC Based Shared Memory Multi-processor SoC Architectures”. In: EUROMICRO, pp. 53-60, 2006.

[3] Leverich, J.; Arakida, H.; Solomatnikov, A.; Firoozshahian, A.; Horowitz, M.; Kozyrakis, C. “Comparing memory systems for chip multiprocessors”. In: ISCA, pp. 358-368. 2007.

[4] Ros, A.; Acacio, M; Garcia, J. DiCo-CMP: “Efficient Cache Coherency in Tiled CMP Architectures”. In: IPDPS, pp. 1-11. 2008.

[5] Magen, N.; Kolodny, A.; Weiser, U.; Shamir N. “Interconnect-power dissipation in microprocessor”. In: SLIP, pp. 7-13. 2004.

[6] Monchiero, M.; Palermo, G.; Silvano, C.; Villa, O. “Exploration of Distributed Shared Memory Architectures for NoC-based Multiprocessors”. Journal of Systems Architecture: the EUROMICRO Journal, vol. 53(10), pp.719-732, 2006.

[7] Silva G. G. B.; Barcelos, D.; Wagner, F. R. “Performance and Energy Evolution of Memory Hierarchies in NoC-based MPSoCs under Latency”. In: IFIP VLSI-SoC, 2009.

[8] Tota, S. V; et al. “MEDEA: a hybrid shared-memory/message-pass-ing multiprocessor NoC-based architecture”. In: DATE, pp. 45-50, 2010

[9] Bolotin, E.; Guz, Z.; Cidon, I.; et al. “The Power of Priority: NoC Based Distributed Cache Coherency”. In: NOCS, pp.117-126. 2007.

[10] Barroso, L. A.; et al. “Piranha: a scalable architecture based on single-chip multiprocessing”. In: ISCA, pp 282-29. 2000.

[11] Jerger, E. N. D.; Peh, L.; Lipasti, M. H. “Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence”. In: MICRO, pp. 35-46, 2008.

[12] Carara, E., Oliveira, R., Calazans, N., Moraes, F. HeMPS - a Framework for NoC-based MPSoC Generation. In: ISCAS, 2009, pp.1345-1348.

[13] Hu, J; et al. Energy-aware mapping for tile-based NoC architectures under performance constraints. In: ASP-DAC, pp. 233-239, 2003.