Distributed Shared Memory for NoC-based MPSoCs

Tales Marchesan Chaves and Fernando Gehm Moraes

PUCRS – FACIN – Av. Ipiranga 6681 – Porto Alegre – 90619-900 - Brazil
e-mail: tales.chaves@gmail.com, 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 that present scalability, low energy consumption and low latency. Two factors impact the scalability of MPSoC systems: cache access latency and energy consumption. The distance of the processors to the cache banks and the cache coherence protocol influences both factors. This work proposes a physically distributed data L2 cache as the cache architecture for a NoC-based MPSoC, because it allows the concept of clustering, and the implementation of data migration algorithms to reducing cache access latency. Results show that the number of cycles required to execute a given application might reduce 23% with the appropriate number of L2 cache banks. Also, a directory-based cache coherence protocol was implemented, exploiting the NoC physical services to improve performance. 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 cache transactions.

Index Terms: MPSoC, NoC, Cache Coherence, DSM.

1. INTRODUCTION

The architecture of an MPSoC may be composed of elements such as: PEs, memory elements and a communication infrastructure. According to [1], one of the most critical components that determine the success of an MPSoC is its memory system. This assertion justifies itself by the fact that applications might spend several cycles waiting for the conclusion of a given memory operation.

One approach to decrease the gap of speed between processing elements and memory elements, a solution commonly applied in high-end microprocessors is the use of static memories and the concept of memory hierarchy. In general-purpose systems, there usually exist four levels of memory: level 1 cache, level 2 cache, main memory and secondary memory.

Cache memories can provide an acceptable data rate to feed the processor, maximizing the number of instructions that are executed in a certain period. Caches work as temporary, fast access memories that prevent the processors to sit idle while waiting for an instruction or data from slower memories. Another interesting point in the use of caches is the possibility of reducing energy consumption. As most memory accesses are done locally at the cache, transferring data on a bus/network-on-chip from slower memories are avoided.

Despite the advantages, some problems related to the use of cache memories must be addressed. According to [2], data residing in a cache bank located near to the processor could be accessed much faster than data that reside in a bank physically farther from the processor. The nearest processor's cache bank, in a 16-megabyte, on-chip L2 cache, built in a 50-nanometer process technology, could be accessed in 4 cycles, while an access to the farthest cache bank might take 47 cycles. This difference is caused due to wire latency. Therefore, as the manufacturing technology advances, the number of cycles to retrieve data from further cache banks increases. Additionally, the use of cache memories in MPSoCs creates the problem of cache incoherence. To avoid incoherence, a cache coherence protocol must be implemented.

A. Motivation and Objectives

As the number of PEs increase [3], problems, such as scalability, memory bandwidth, cache coherence and data migration, found in high-performance computing will also need to be studied in embedded devices. Other problems that must be considered are: memory access latency, once several processors can simultaneously issue requests to the memory; energy consumption; memory banks positioning inside the chip.

These problems direct to the study of alternative memory solutions for MPSoCs, to satisfy design constraints commonly applied in embedded systems, such as memory access latency, power consumption, area, and programming model (e.g. shared memory or message passing).

The *goal* of the present work is to propose a physically distributed data L2 cache as the cache architecture for a NoC-based MPSoC. NoC services, such as multicast and priorities, and some properties of the NoC, such as duplicated physical links, are used to optimize the directory-based cache coherence protocol. Exposing the low-level NoC services to the coherence protocol may reduce the energy consumption and latency of the cache accesses.

B. Document Organization

This paper is divided in 7 sections. Section II presents the state of the art in the field of memory architecture and cache coherence for NoC-based MPSoCs. Section III presents the platform used as reference for this work. Section IV presents the shared memory architectures developed for this work. Section V presents an optimized cache coherence protocol. Section VI presents the results. Finally, Section VII presents the conclusion.

2. STATE OF THE ART

This section presents state-of-the-art works in two fields: memory architecture and cache coherence protocols. These topics are separately presented because this work proposes solutions for each field.

A. Memory Architecture

The shift from busses to NoCs influences the memory architecture of a system because in NoCs different requests to the memory may travel through different routes. This leads to unordered access to the memory, as well as variable latency due to the NoC load.

Kim et al. [4] affirms that the higher latency presented by routers in a NoC directly affects the way distributed shared memories are designed. The Authors propose a switch architecture for low-latency cache coherence of a distributed shared memory MPSoC platform denoted DCOS (Directory Cache On a Switch). The memory system is physically distributed but logically shared between PEs. The directory cache coherence protocol adopted is the modified-shared-invalidate (MSI). The tests and evaluations of the DCOS architecture were done using a modified version of the RSIM MPSoC simulator. The results show a substantial reduction on the average read latency and execution time compared to a platform in which directory caches are not embedded into the switches. Although, according to the results presented by the authors an important decrease in execution time is obtained only when the directory contains 2048 entries.

Monchiero et al. [5] present a NoC-based MPSoC composed by ARM7 PEs, memory elements and interrupt controllers. The memory elements can be of two types: shared memory banks or L2 caches. To access one of the shared banks, a given PE must send a request to a unit that is responsible for managing the shared memory, named HW Memory Management Unit (HWMMU). The authors simulated several scenarios varying parameters of the memory architecture (number of shared memory banks) and NoC topology (RING, SPIDERGON AND MESH). The results showed that a decrease in latency is obtained with the increase on the number of shared memory banks. Although, for more than four memory banks the communication cost overcomes the gains obtained from the increase of memory banks. According to the authors, latency reduction is due the decrease on the memory contention and the decrease of transactions on the network.

Man et. al [6] discuss the problem of limited bandwidth when using a unique, centralized Memory Management Unit (MMU). As the number of PEs increases, the number of memory references also tends to increase, therefore the sequential structure of a centralized MMU can become the bottleneck of the system. To tackle this problem, Man et al. proposes a distributed MMU scheme, which takes several MMUs as resources on the NoC to handle memory access requests. According to the authors, with a proper number of MMUs and reasonable network placement, this design would lower both memory bandwidth requirements and NoC communication traffic. PEs are grouped into Translation-Sharing Partitions (TSP). Each partition has: a set of PEs and a MMU responsible of handling requests of the PEs belonging to that partition. The problem with this approach is that it limits a TSP to only one MMU.

B. Cache Coherence Protocols

According to [7], 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 [8]. This consumption might approach 50% of the overall chip power in some cases. Considering that MPSoC designs mainly target low energy consumption, there is a necessity of the study and development of energy efficient protocols.

Few works on the literature explore the physical services, which are provided by NoCs to optimize the cache coherence protocol. Examples of NoC-based MPSoCs adopting memory hierarchy with caches includes [9][10][11], having as common feature the abstraction of the communication infrastructure, adopting only send and receive services.

Bolotin et al. [12] 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. [13] propose an invalidation-based directory protocol, with some optimizations. One optimization proposed is named clean-exclusive, in which an exclusive copy is returned to a read if there are no sharers. An important enhancement is that this protocol avoids the use 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. [14] 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. The authors do not evaluate the size of the tree, which keeps a history of the nodes sharing a common region, and not discuss the energy gains when using the VTC protocol.

C. Conclusions on the state of the art

Most works adopt DSM (Distributed Shared Memory) architecture for NoC-based MPSoCs, due to the increasing number of PEs. Latency and contention are still an open problem, even when more than one MMU is employed. Data migration algorithms help reducing latency by migrating data closer to the PEs mostly accessing them. This fact may decrease the network contention as the traffic tends to be restrained to a given region, which means that different applications will use different regions of the chip. Therefore, only enhancing the DSM architecture is not a scalable solution.

The cache coherence protocol also plays an important role as the system scales. This is because a significant amount of traffic is required to enforce system coherence, such as invalidate messages and write-backs. To avoid huge traffic loads, the exploitation of NoC physical services is required in order to both reduce traffic, latency and energy consumption.

Considering this discussion, this work proposes a DSM architecture for a NoC-based MPSoC, which supports cache coherence. Data migration is out of the scope of the present work.

3. HeMPS MPSoC

The HeMPS Platform [15], shown in Figure 1, is a homogeneous MPSoC in which PEs are interconnected through the Hermes NoC. Each PE contains a RISC microprocessor, a local memory, a DMA controller, and a Network Interface (NI). These modules are wrapped by the Plasma-IP module, that is then connected to the NoC.

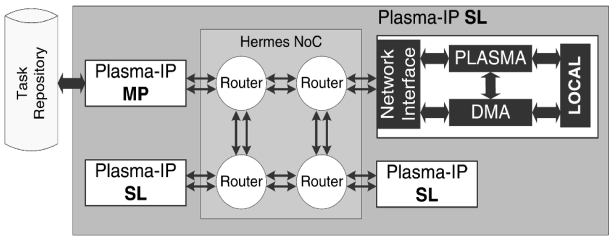


Figure 1. HeMPS platform block diagram.

The following reasons justify the adoption of the HeMPS platform: (*i*) all modules are implemented in RTL; (*ii*) it adopts a NoC to interconnect PEs; (*iii*) it follows the trends of the state of the art of embedded systems; (*iv*) it has a framework that allows the parameterization of PEs, application tasks and NoC; (*v*) it is open-source.

Next sections detail these modules. Additionally, section F presents the *microkernel* and section K presents the framework enabling the MPSoC generation.

A. Hermes QoS

An improved version of the Hermes NoC, named Hermes QoS, was proposed in [16]. This version has mechanisms to provide QoS (Quality of Service), to reduce latency and energy consumption. Hermes QoS uses the Hamiltonian routing algorithm, two physical channels, and a priority scheme. According to [16], it is possible to define several Hamiltonian paths on a bi-dimensional mesh by labeling the routers from 0 to N-1, being N the number of routers. After that, the network can be divided into two disjoint and acyclic sub-networks. One sub-network contains the channels that go from the smallest label router to the highest label router and the other contains the channels that go from the highest label router to the smallest label router. Thus, packets sent from a router labeled 0 targeting a router labeled 3 will take the ascending path.

Hamiltonian paths provide a good support for implementing efficient multicast/broadcast algorithms. For example, multicast messages can be used by cache coherence protocols when invalidation messages need to be sent to all processors that are caching a given block. Without multicast, an invalidation message would have to be sent individually to all processors in the system, increasing the number of transactions in the network, as well as energy consumption and congestion. Thus, it is possible to assert that the interconnection infrastructure plays an important role on the design and implementation of a cache coherence protocol.

B. Plasma-IP

Plasma-IP is a modified version of an open-source soft core, described in VHDL, freely obtained from OpenCores [17]. It is a 32-bit RISC architecture, based on the MIPS-I ISA (Instruction Set Architecture). The most important modifications performed on the original Plasma architecture are: insertion of a paging mechanism; addition of a DMA module and a NI (Network Interface). Concerning the instruction set, the Plasma processor originally did not include the *syscall* instruction, which allows for user applications to ask the kernel to perform low-level functions such as accessing I/O devices or controlling an external memory. The *syscall* instruction was added to the Plasma implementation.

C. Network Interface (NI)

The NI acts as a wrapper between the Plasma processor and the NoC. It is responsible for adapting data coming out from the processor to NoC standards. The length of the data channel of the network is 16 bits (flit size). The Plasma word is 32-bit wide. So, for every word sent from the processor to the network, two flits are sent through the NoC. The NI is responsible for breaking up words into flits and re-assembling them at the receiving node.

The NI contains two finite-state machines: *Receive* and *Send*. The *Receive* machine is a 4-state machine responsible for receiving data coming from the network and buffering them (up to 16 32-bit words). Furthermore, this machine is responsible for controlling the reading of data from Plasma core. The *Send* machine is a 6-state machine responsible for sending packets to the NoC. The contents of the packets sent on the NI are fetched from memory by the DMA module, described in the next subsection.

D. DMA

The DMA module executes two operations: transfer of packets stored on memory to the NI and transfer of incoming packets from NI to memory. DMA programming is done inside the microkernel, whenever a new interrupt event originated from the NI module is detected, or when a packet must be sent to the NoC. The configuration of the DMA module is done through a set of memory-mapped registers.

E. Memory System

Each Plasma-IP follows the Von-Neumann organization, having only one private memory that stores both data and instructions. To allow simultaneous access from DMA and processor, the private memory is a true dual-port memory, implemented as a BRAM memory when synthesized on a Xilinx FPGA. Section 4 presents a shared cache architecture proposed to the HeMPS Platform.

F. Microkernel

The microkernel running on Plasma processors is a small operating system, whose main goal is manage task execution on each core. The microkernel implements a preemptive scheduler that allows applications to use the CPU during a pre-defined period of time called *timeslice*.

Preemptive multitasking requires support from the hardware, otherwise there might exist situations where it is not possible to remove a given task from the CPU. The PLASMA core provides a memory-mapped tick counter, which is incremented by one at each clock cycle. The tick counter generates an interrupt event when it reaches a threshold.

Timeslice is defined as the time interval which a task occupies the CPU. When the *timeslice* ends, an interrupt is dispatched. This interrupt is handled by the interrupt handler, which in turn, calls the scheduling algorithm, which will select a new task to be executed on the CPU.

The microkernel has two versions: master and slave. The master version runs on PLASMA master whose main goal is to coordinate task mapping and manage their execution. The master does not execute any application task and its memory footprint is about 4,3KB. The slave microkernel runs on PLASMA slave and provides support to multitasking and software interrupts (*traps*). The microkernel slave occupies the initial pages of the memory.

G. Task allocation

The master PE, according to the task mapping heuristic, performs task allocation on HeMPS Platform. HeMPS Platform supports two types of task mappings, static and dynamic.

Static mapping: the user defines, at design time, in which Plasma-IP each task is going to be executed. When the execution starts, tasks are sent to PEs according to the user definition.

Dynamic mapping [18][19]: a dynamically allocated task is only transferred to the Plasma-IP that will execute it when another task in the system needs to send a message to it. This situation works as follows: (*i*) a task *Ti*, running on processor *Pi* needs to send a message to task *Tj*; (*ii*) task *Ti* calls the send primitive, which requests the microkernel to send the message; (*iii*) the microkernel searches for task *Tj* location on its internal table, but the task *Tj* has not been allocated yet; (*iv*) as the microkernel do not find the location of task T*j*, it sends a message to the master requesting the allocation to be done.

H. Interrupts

Interrupts in the microkernel can be generated after the occurrence of one of the following events: arrival of a new packet at the NI; *timeslice* counter reaches up its limit, indicating a new task must be allocated; and a call to the *syscall* primitive (software interrupt – *trap*).

I. Inter-task communication

Different tasks running on a distributed system need to communicate in order to coordinate and synchronize task execution. Inter-task communication inside the processor is done through a memory area called *pipe*. A *pipe* is a communication channel in which messages are consumed at the same order they were produced. There exists only one global pipe per processor that is shared between all tasks.

According to the KPN communication model [20], adopted by HeMPS microkernel, a channel read operation must be blocking. Thus, when a *ReadPipe()* instruction is executed, the task is blocked until it receives the requested message from the pipe. At the opposite, write operations must be non-blocking, which means that after every *WritePipe()* operation, the task keeps executing (in case the pipe is not full).

When a task wants to receive a message, a read operation on the pipe occurs. Whenever a task wants to send a message it executes a write operation (*WritePipe()*). If both sending and receiving task are located at the same processor, no message is sent through the network. Although, in case tasks reside at different processors, a packet is assembled and sent to the target task processor. This process is shown in Figure 2, where in (a) task 2 (*t2*) located at Processor 1 writes a message to task 5 (*t5*) on the *pipe* and continues its execution. Subsequently, task 5 makes an explicit call to *request_msg* that sends a message to Processor 1 requesting the reading of the pipe. Then, the NI interrupts Processor 1 when the request arrives, and sends a message (*msg*, Figure 2(b)) to processor 2 that receives the message and unblocks task 2.

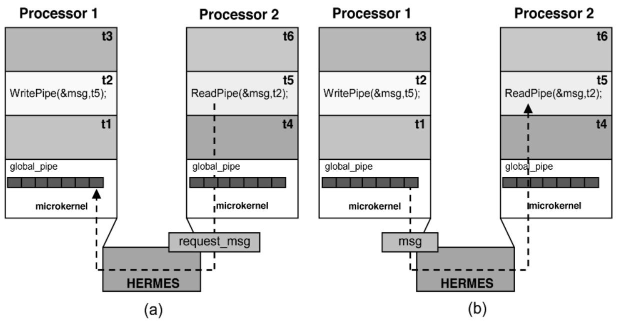


Figure 2. Intertask communication between tasks located at different processors.

J. Memory management

As already mentioned, each Plasma-IP contains a private dual-port memory, which is divided into pages. The first pages store the microkernel, and the subsequent ones store user tasks. Each task is associated to a page number, which is kept by a CPU internal register, denominated *page*. Every memory address (*mem_address_wop*) generated by memory controller (*Mem_ctrl*) of the Plasma core does not includes the page, only the logical offset inside it. Thus, before putting the address on the memory bus, it is necessary to concatenate the page initial address in memory with the address contained in *mem_address_wop*. This mechanism prevents a given task located at page 2, for instance, from accessing data from a task located at a different page. An important consideration is that this mechanism also prevents tasks from accessing the kernel memory area, avoiding a crash caused by a malicious user task.

K. HeMPS Generator

The HeMPS Generator is a framework that allows the customization of an MPSoC system for simulation of applications tasks. It is possible to parameterize: the number of PEs; the size of the NoC; the page size; the maximum number of tasks per processor; the abstraction level of the PEs: ISS or RTL. Finally, the user can insert application tasks and statically map them to PEs.

4. SHARED MEMORY ARCHITECTURE

This section details the architecture of the L2 cache module. This module is responsible for handling requests, such as read and writes, from the L1 data cache. Additionally, the L2 cache has support for a cache coherence protocol. The L2 cache is connected to the NoC through a NI, as shown in Figure 3. The L2 cache module can be instantiated one or more times in a system, resulting in a centralized or distributed shared L2 data cache. The next subsections provide a detailed view of the services supported by the L2 cache module and its internal architecture. Next, examples of centralized and distributed systems are presented. Finally, it is presented the API that allows application tasks to manipulate data from the cache structure.

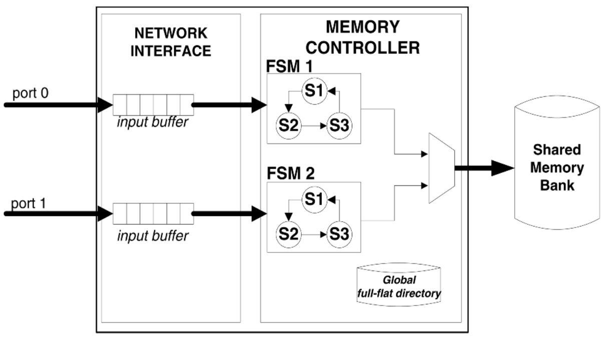


Figure 3. Cache L2 Controller architecture.

A. Supported Services

The services supported by the current implementation of the L2 cache controller include:

- READ_BLOCK, returns a copy of a block;
- WRITE_BACK, writes the content of an entire

block to the L2 cache;
- INVALIDATE_BLOCK, invalidates cache lines at the L1 cache of PEs. This service is required before granting exclusivity of write to a given PE;
- ASK_EXCLUSIVITY which is sent from a PE to the L2 cache to get an exclusive copy of a given cache line;
- GRANT_EXCLUSIVITY service where the L2 cache guarantees to only one PE the exclusive right to modify a certain block;
- FLUSH_BLOCK that has the same effect that a WRITE_BACK, but it is only used when an application task finishes its execution.

B. L2 Cache Controller Architecture

The controller is divided in two main modules, as illustrated in Figure 3: Network Interface (NI) and Memory Controller (MC). The NI is responsible for interfacing the two physical links of the NoC. The use of two separated ports to communicate with the NoC allows the differentiation of cache requests. Short packets, such as read requests, are addressed to port 0 of the NI, whilst long packets, such as write-back packets are addressed to port 1.

Packets arriving at the NI are buffered in the *input_buffer* (shown in Figure 3). Write-back packets are partially buffered due to the limited size of the buffer. Output ports are not buffered. As soon as the NI notices that a new packet is being stored in its input buffer, it signalizes the arrival of a new packet to the MC through an interrupt signal.

The MC is implemented based on two independent FSM. FSM 1 is responsible for handling packets from port 0, which may contain services such as: READ_REQUEST, ASK_EXCLUSIVITY, INVALIDATE_BLOCK, and GRANT_ EXCLUSIVITY (described in Section V). FSM 2 is responsible for handling packets from port 1, which may contain services such as WRITE_BACK and FLUSH_BLOCK. The MC maintains a centralized directory that stores, for each block, its status according to the cache coherence protocol.

C. Message Exchange Default Format

The messages exchanged between PEs and the L2 cache controller follow the format shown in Figure 4. The fields of this packet corresponds to:

- *TargetNetAddr*: network address of target PE;
- *Size*: packet size, in *flits* (each flit has 16 bits);
- Service: specifies the operation to be executed and can be any of the services described in Section 4.A;
- *SourceNetAddr*: router network address of the sender of the message;
- *TargetBlock*: address of the block where the operation will be performed;
- *SourceTaskId*: identifier of the application task which triggered the operation;
- *Payload*: optional field, being 256-flit wide, with the data of an entire block. It is only used in WRITE_BACK and FLUSH_BLOCK services.

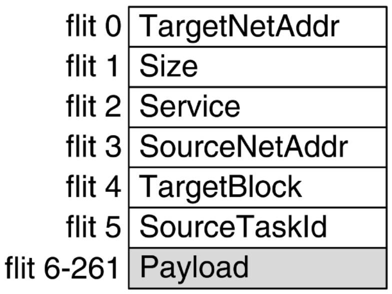


Figure 4. Message format for communication with the L2 cache controller.

D. L1 Cache Controller Architecture

The L1 cache module is located inside the Plasma entity. It contains a cache controller, responsible for storing and retrieving cache lines, detect hits/misses and manage the tag memory (TM). The TM stores, for each cache line, the following information: *tag address* which corresponds to the higher bits of the block address mapped to a cache line; *modified*, which informs if the cache line has been altered, *valid*, which informs if the cache line is valid.

The cache adopts the direct mapping scheme, due to the minimum hardware support required to implement it. Each cache line stores a full block of the L2 cache, which size is 128 32-bit words. The size of the cache line was chosen based on the size of a pipe slot in the current implementation of the HeMPS Platform.

E. Centralized L2 Data Cache

Figure 5 shows an MPSoC system with a centralized shared L2 data cache. All PEs of the system have access to the L2 cache. This fact facilitates the development of application tasks as the cache address space is seen by all PEs. Although, a central L2 cache architecture has two main negative points. First, the bandwidth required by the L2 cache controller increases with the increase of PEs. The cache can easily become a bottleneck of the system as the platform grows. Second, the average distance of the PEs to the L2 cache controller increases as the size of the system increases. An increase in the average distance culminates in the increase of the latency of the cache requests. Additionally, the energy consumed by cache operations increases as the requests must traverse a longer distance to get from a PE to the L2 cache and vice-versa.

The L2 cache can be initialized at design time. The HeMPS Generator allows the user to load a file with initial values for each line of the cache for simulation matters.

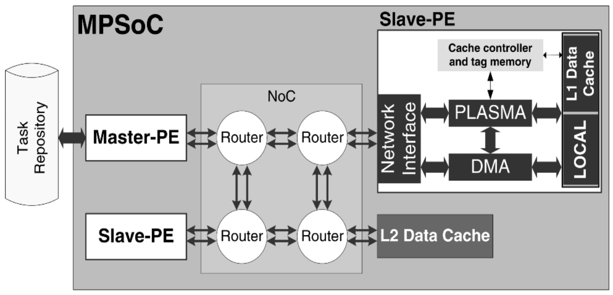


Figure 5. Example of an MPSoC with a centralized shared L2 data cache.

F. Distributed L2 Data Cache

Figure 6 shows an MPSoC with a distributed shared L2 cache. In the literature it is commonly referred as a Non-Uniform Cache Access (NUCA) system, due to the fact that the access latency varies from PE to PE because of the variable distance to the L2 cache banks.

The advantage of the distributed approach is that it allows the concept of clustering, which allows applications to be mapped to access one cache bank. In a system with four L2 cache banks for example, the application tasks may be divided so that each task accesses one L2 cache, decreasing the bandwidth required for each bank and also decreasing energy consumption and latency as the applications may be mapped in PEs close to the bank that they are mostly accessing. Another advantage of the distributed approach is that it allows data migration. When an given application task is mapped to a PE which is not close to the L2 bank that it mostly accesses, the data migration algorithm may migrate the data being mostly accessed by that PE to a closer bank, decreasing latency and energy spent in communication.

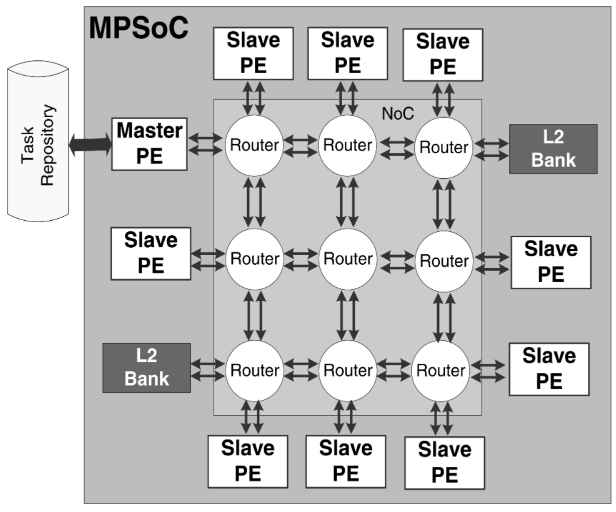


Figure 6. Example of an MPSoC with distributed shared L2 cache banks.

G. Programming API

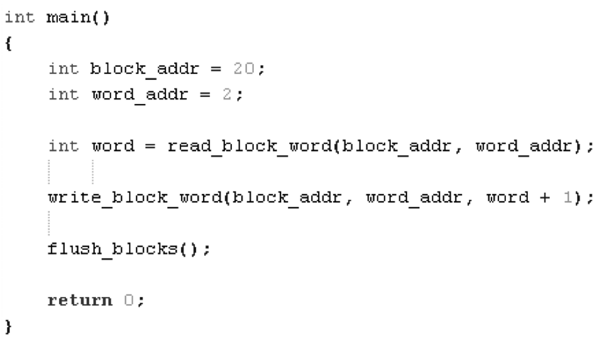
This section details the cache programming API, which provides access to the data cache structure at the application level. This API is linked with the application tasks and the operating system after compilation.

1. Functions description

- *initialize_cache()* – this function initializes the cache structures, such as freeing all data structures present in the *kernel* to manipulate the L1 data cache. Also, it resets the bits of hit and tags of the L1 cache controller.
- *read_block_word(int block_addr, int word_offset)* – this function is responsible for performing the process of reading a block from the data cache. As the block being accessed might not be copied at the L1 cache, firstly the function must verify if the block is present at the L1 cache. If the block is not present at the L1 cache, a READ_REQUEST message is assembled and sent to the L2 cache controller. The task is set to WAIT state until the L2 cache returns the block to the L1 cache.
- *write_block_word(int block_addr, int word_offset, int datum)* – this function is responsible for performing the write of a single datum to a single word of a line present at the L1 cache. If the line being accessed is not present at the L1 cache, the procedure to retrieve the line is the same executed during a *read miss*.
- *flush_blocks()* – this function flushes all lines present at the L1 cache to the L2 cache. It is used by the microkernel every time a given application finishes in order to updating the L2 with the data of the application. This function might also be used for debugging purposes.

2. API usage in an example application

The application shown in Figure 7 is an example application that calls functions of the Cache API. The first call is to *read_block_word* function, which returns a 32-bit integer corresponding to the word of the block



```
int main()
{
    int block_addr = 20;
    int word_addr = 2;

    int word = read_block_word(block_addr, word_addr);

    write_block_word(block_addr, word_addr, word + 1);

    flush_blocks();

    return 0;
}
```

Figure 7. Example application that uses the cache API.

being requested. This return can be casted to other type, such as a char. Next, a call to the *write_block_word* is performed, incrementing by one the value read previously. The call to the *flush_blocks* is not necessary in a real application as the kernel calls this function automatically at the end of each application task.

5. SHARED MEMORY ARCHITECTURE

This section presents a directory-based cache coherence protocol implemented in the HeMPS Platform. 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. This state indicates that a write back request has been issued to the PE previously holding exclusivity on this block, but it has not been written in the shared memory yet. The addition of this state optimizes the coherence protocol, as explained next.

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). 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.

A. Protocol optimizations

Most protocols, such as invalidate protocols, generate several unicast messages whenever a block needs to be invalidated in several caches. This traffic increases significantly the energy consumption introduced by the cache. One way to minimize this overhead is to send multicast messages to reduce the traffic induced by the invalidation. Exposing the low-level NoC features may reduce the energy consumption and increase the performance of cache coherence protocols for NoC-based MPSoCs. The next subsections present individual optimizations applied to the MSI protocol through the exploration of NoC physical services. All control packets of the coherence protocol are sent through the NoC using high priority packets, enabling their fast transmission even in congested NoC regions.

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 8(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 8(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.

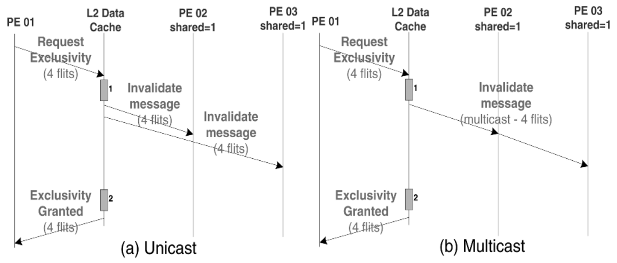

PE 01
L2 Data Cache
PE 02 shared=1
PE 03 shared=1
Request Exclusivity (4 flits)
Invalidate message (4 flits)
Invalidate message (4 flits)
Exclusivity Granted (4 flits)
(a) Unicast
Invalidate message (multicast - 4 flits)
(b) Multicast

Figure 8. Sequence diagram for a request of exclusivity on a shared block.

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 9(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.

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

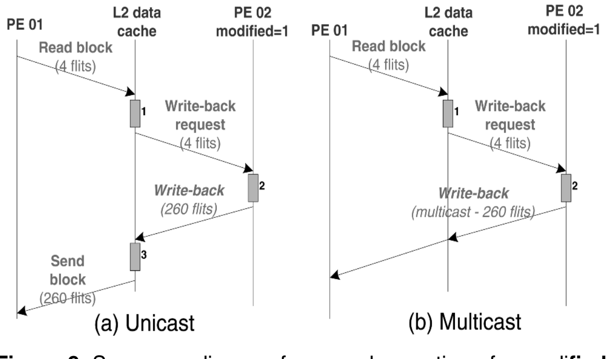


Figure 9. Sequence diagram for a read operation of a modi**fied block.**

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 10(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.

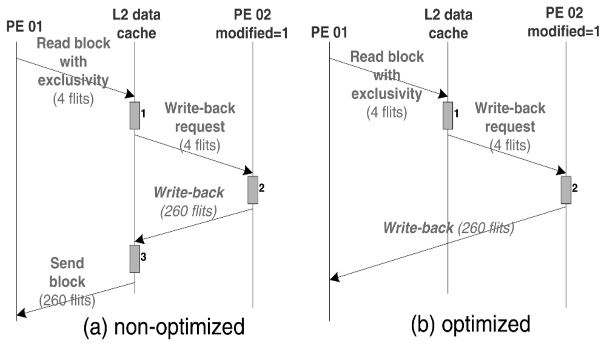


Figure 10. Sequence diagram for a write-back after a write request.

In the optimized implementation (Figure 10(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.

4. The Transition State

The benefit of having a new state in the cache coherence protocol is the possibility of decreasing latency. This is achieved by transferring memory requests to a PE that also has an updated copy of the block being read.

Figure 11 presents a scenario illustrating the use 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 11). The L2 cache upon receiving PE00's request, search in the directory the processor holding the block in modified state, issuing a write-back request to PE03 (event 2), setting the block from M (modified) to T (transition) state. Next, PE05 also requests a read in the same block (event 3). Instead of blocking the request of PE05, the L2 cache issues a read request of this block to PE03 (step 4), which sends a packet containing a copy of the cached block stored in its cache (step 7).

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.

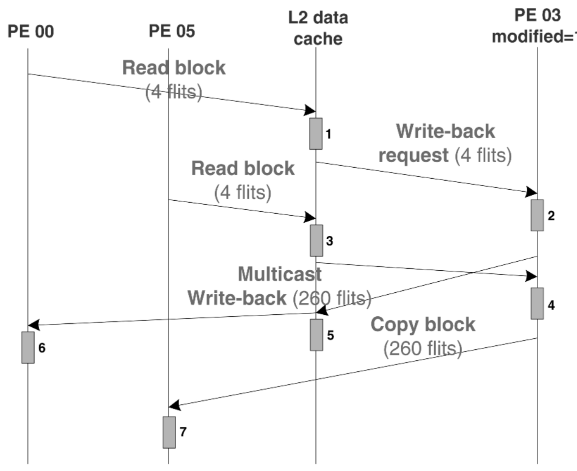


Figure 11. Sequence diagram for the T state.

6. RESULTS

This section is divided into three subsections: the first one presents results obtained with the optimizations in the cache coherence protocol; the second one presents results for mostly-read and mostly-write synthetic applications; finally, the third section presents results obtained from two real applications.

A. Cache coherence protocol evaluation

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 2-D 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 A. 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 evaluate the consumed energy per memory transaction, the present work adopts the volume-based energy model proposed by Hu et al. [21]. Equation 1 computes the communication energy spent to transit *1 bit* through a distance of *n* hops.

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

where: 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.

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.

1. 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, significantly reducing 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	NO-OPT	1635	2584	3798
(pJ)	OPT	685	2073	2916
	OPT gain vs NO-OPT	**58.07%**	**19.76%**	**23.20%**
Clock	NO-OPT	141	154	147
Cycles	OPT	129	127	129
	OPT gain vs NO-OPT	**8.51%**	**17.53%**	**12.24%**

2. 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 12 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.

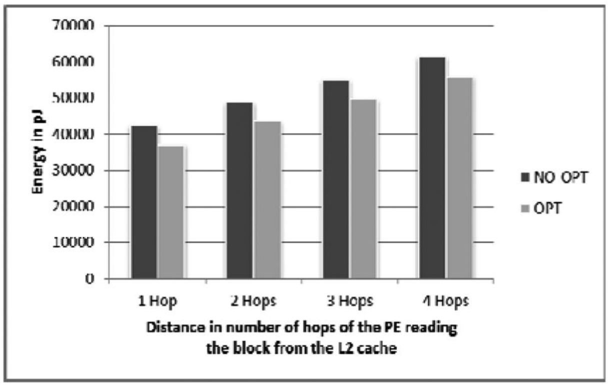


Figure 12. Energy consumption of the read operation on a modified block as the number of hops increases.

3. 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 15(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 13 shows that there is an average reduction of 17% in the number of cycles required to finish the write operation. Also, Figure 14 shows that there is a

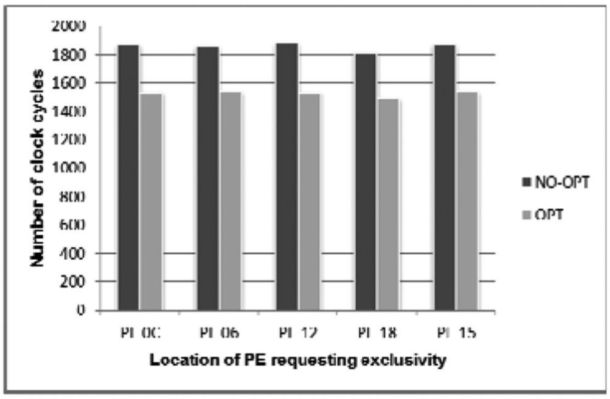


Figure 13. Number of cycles required to execute a read operation on a modified block varying the location of the modified block.

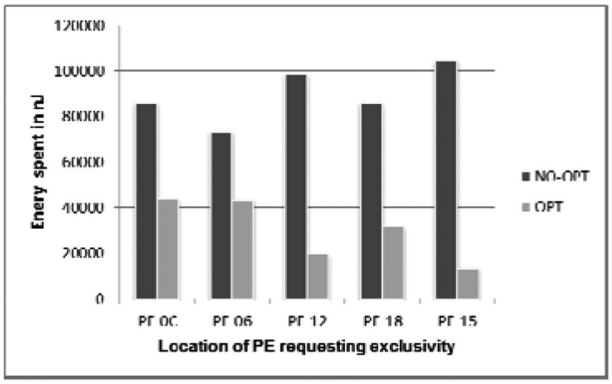


Figure 14. Energy consumed to execute a read operation on a modified block varying the location of the modified block.

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.

4. 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). The first PE which 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 15(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 15(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%.

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.

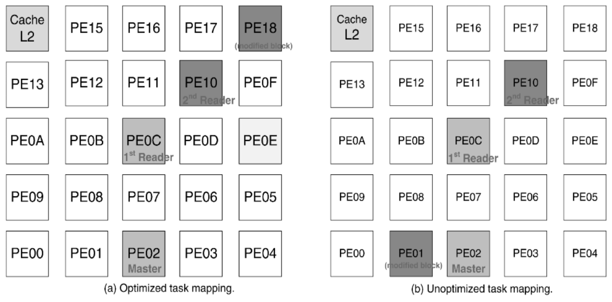


Figure 15. Task mappings for the T state optimization.

B. Evaluation through access patterns

According to [22], the way that applications access the memory can be classified in: *mostly-read*, *mostly-write* and *mixed*. Applications are classified as *mostly-read* when the majority of memory accesses are reads; *mostly-write* when the majority of accesses are writes; and *mixed*, where read and write operations occur with the same probability. Considering the fact that in the HeMPS Platform, before writing to a block of the cache, the application must read it, we assume that mostly-write and mixed sets can be contemplated by the same set of applications.

To evaluate the benefits of DSM architecture in a NoC-based MPSoC, three sets of application were developed. Each set access the memory according to a given pattern, as described previously.

For each pattern there are 4 different implementations. Each implementation differs from the others in the number of tasks, which can be 4, 6, 8 and 10 tasks. Each task accesses four blocks of the L2 cache. In the mostly-read set, each task reads 4 blocks. For each block read, 40% of it is locally used by the application before requesting the next block. In the mostly-write set, each task writes to 4 blocks. For each block, 40% of each block is locally modified by the task before starting writing in the next block.

Each set of application was simulated in a 4x4 instance of the HeMPS Platform, having one task per PE (up to 10 PEs) and three L2 cache bank configurations: 1 L2 cache bank, 2 L2 cache banks and 4 L2 cache banks. The parameters evaluated during the simulation are presented per application: average miss latency; number of misses; and energy consumed with communication.

1. Mostly-read set

The results for mostly-read application set are shown in Figure 16. According to the results, the average cache miss latency decreases as the number of L2 cache banks increases, due to two factors: (*i*) the increased cache bandwidth; and (*ii*) the possibility of mapping applications tasks to PEs located near to the cache bank which will be mostly accessed by the task. The cache bandwidth is increased because each bank is independent and can handle a different cache request, which reduces the number of packets that are queued before being handled. In current manufacturing technologies, wire delay is considered a major problem to cache designs, therefore allocating tasks close to cache banks helps reducing the latency that a given operation takes to be executed.

As shown in Figure 16, cache miss latency for the configuration which uses only 1 L2 cache bank is always higher, even when being accessed by a small number of tasks. The two banks configuration presents an average latency similar to the latency presented by 4 banks (except for the 4 tasks case). Although, the worst latency (shown as extension of the graph – thinner lines) is high for 10 tasks, which means that some of the tasks experience high miss latency in this configuration.

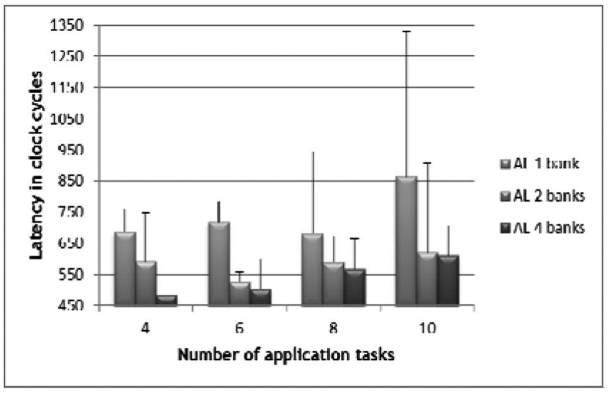


Figure 16. Cache miss average latency (AL), expressed in number of clock cycles, for a given number of L2 cache banks when executing the mostly-read set.

2. Mostly-write set

The results for the mostly-write application set are shown in Figure 17. The difference from the mostly-read application set is that the 4 cache banks configuration shows a more significant decrease in average cache miss latency when compared to the other configurations. The reduction in number of cycles is of up to 23% for the 8 tasks application.

It is possible to conclude that the average cache miss latency increases fast as the number of applications accessing that cache also increase. Therefore, the DSM architecture can help reducing latency by distributing cache access packets across the NoC and also, by increasing access bandwidth.

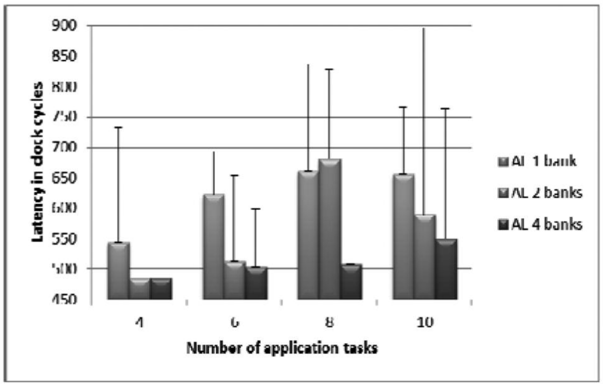


Figure 17. Average write latency (AL), shown in number of clock cycles, for a given number of L2 cache banks. Also, worst write request latency (WL) for a given configuration of cache banks.

C. Application benchmarks

This subsection describes the modeling and development of two applications, which were adopted as benchmarks to evaluate the benefits of DSM over CSM architecture.

1. Matrix multiplication (MM)

The implementation of the matrix application follows the traditional algorithm, with two-nested loop and no optimizations. Each task can iterate over part of the lines of the first input matrix, while the inner loops must be replicated in all tasks. The matrix multiplication can be characterized as a mostly-read application due to the fact the innermost loop has two reads, while the write operation is only executed once per column.

In the implementation for the HeMPS Platform, the input matrices are statically set into the L2 cache banks at design time. The size of both matrices is 10x10.

Figure 18 presents the task graph of two MM applications. In both cases, two tasks are used to execute the multiplication; each task is responsible for generating half of the resulting matrix. The difference is that in task graph *(a)*, both tasks access the same L2 cache bank, while in task graph *(b)*, each task accesses a different L2 cache bank.

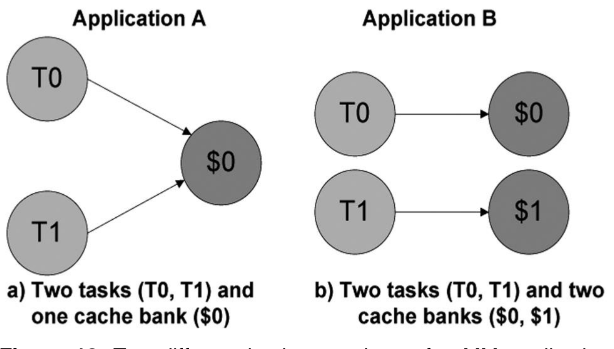

Application A
Application B
T0
T1
$0
T0
$0
T1
$1
a) Two tasks (T0, T1) and one cache bank ($0)
b) Two tasks (T0, T1) and two cache banks ($0, $1)

Figure 18. Two different implementations of a MM application using 2 tasks.

Table 2 shows that the average miss latency is smaller in application B, due to the use of 2 cache banks. The energy consumed in communication is also smaller, because applications can be mapped to PEs closer to the L2 bank that it is going to access mostly. This reduces the number of hops that each message sent from a PE to the cache must traverse. The execution time decreases when using 2 cache banks because fewer requests are queued in the L2 cache banks.

Table 2. Results obtained from simulation of scenario 1 of matrix multiplication. (c/c means clock cycles)

	Application A	Application B
Average miss latency	630 c/c	580 c/c
Hit count	13489	13082
Energy consumed (Communication)	3.45µJ	2.28µJ
Execution time	218133.5 c/c	212093.5 c/c

Figure 19 presents two task graphs for a MM application that uses 3 tasks. The situation is similar to the one presented in Figure 18. The difference is that each task is responsible for generating 1/3 of the resulting matrix. Table 3 presents the results. Application A uses only one cache bank, while application B uses two cache banks. As the average miss latency is lower for application B, the total execution time of the application decreases when compared to application A. Compared to *Scenario 1*, the execution time is significantly smaller because of the increase in the level of parallelization. The energy spent in communication is near the same due to the fact that the memory accesses are the same in both cases.

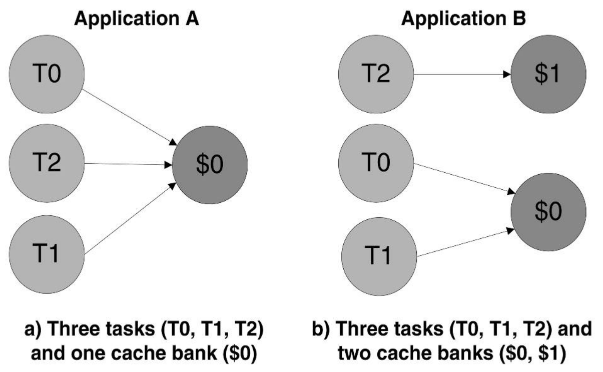

Application A
T0
T2
T1
$0
Application B
T2
$1
T0
T1
$0
a) Three tasks (T0, T1, T2) and one cache bank ($0)
b) Three tasks (T0, T1, T2) and two cache banks ($0, $1)

Figure 19. Two different implementation of a MM application using 3 tasks.

Table 3. Results obtained from simulation of scenario 2 of matrix multiplication. (c/c means clock cycles)

	Application A	Application B
Average miss latency	625 c/c	538 c/c
Hit count	5362	5233
Energy consumed (Communication)	1.69 µJ	1.68 µJ
Execution time	108376 c/c	102898 c/c

2. Equation Solver

The equation solver kernel solves a simple partial differential equation on a grid, using what is referred to as finite differencing method. The kernel sweeps over the grid, updating each point by using the values of its neighbors. The algorithm stops when the generated value for all points converges over a pre-defined tolerance. Figure 20 shows 3 task graphs that implement the equation solver application. In all task graphs, the problem is divided into 4 tasks; each task is responsible by updating 25% of the lines of the grid per sweep. For each sweep, the Sync task is responsible for implementing a barrier that guarantees that each task waits for the others to finish until all the grid is updated. The difference between task graphs *a*, *b* and *c* is the number of L2 cache banks used.

Table 4 shows the results for the three applications. In this case, the only gain using more cache banks is the reduction of the average cache misses, which is of approximately 7% comparing the 4 L2 cache banks configuration over the 1 bank configuration. There is an increase in energy consumption of communication when the number of cache banks increases. The execution time for all applications is nearly the same, which means that the load of the application is not problematic for the only one cache bank scenario.

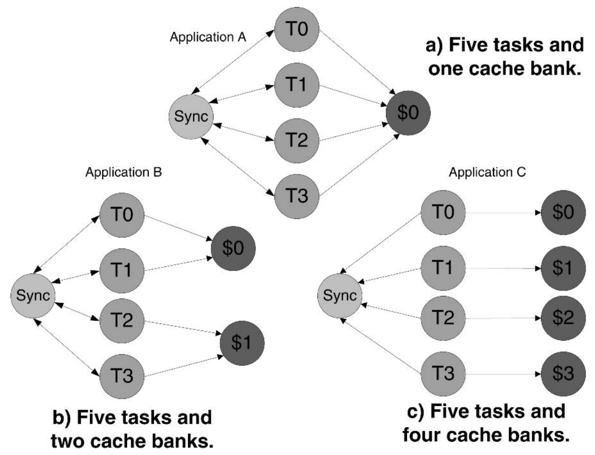

Application A
T0
T1
T2
T3
Sync
$0
a) Five tasks and one cache bank.
Application B
T0
T1
T2
T3
Sync
$0
$1
b) Five tasks and two cache banks.
Application C
T0
T1
T2
T3
Sync
$0
$1
$2
$3
c) Five tasks and four cache banks.

Figure 20. Tasks graphs for equation solver application.

Table 4. Results obtained from simulation of the equation solver application. (c/c means clock cycles)

	Application A	Application B	Application C
Average miss latency	537.25 c/c	510.5 c/c	502.15 c/c
Energy consumed (Communication)	0.17 µJ	0.29 µJ	0.29 µJ
Execution time	0.85ms	0.85ms	0.84ms

7. CONCLUSIONS AND FUTURE WORKS

State-of-the-art designs must cope with the increasing wire delay. Distributed shared memory architectures, such as a split L2 design, help alleviating this problem. The current work presented a distributed shared memory architecture with a cache coherence protocol.

The implemented cache coherence protocol explores 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. This fact points to several future works, as couple the proposed techniques to mapping heuristics that consider the memory position in the MPSoC, and data migration policies to optimize the memory performance.

The exploration of several L2 cache banks allow the reduction on energy consumption as the distance traversed by packets can be decreased by placing application tasks in PEs near a L2 cache bank. Latency can also be decrease as the overall L2 cache bandwidth increases when using a higher number of banks. Additionally, the presence of several cache banks enables the implementation of a data migration mechanism.

Future work also includes: (*i*) study and evaluation of data migration protocols; (*ii*) study and implement a way to distribute the directory used by the cache coherence protocol; (*iii*) evaluation and implementation of a memory consistency model at the software level.

ACKNOWLEDGEMENTS

The author Fernando Gehm Moraes received research support from CNPq and FAPERGS, project numbers 301599/2009-2 and 10/0814-9, respectively.

REFERENCES

[1] M. Kandemir; N. Dutt. "*Memory Systems and Compiler Support for MPSoC Architectures*". Multiprocessor Systems-on-Chips, Kluwer Academic Publishers, 2005, pp. 251–281.

[2] C. Kim; D. Burger; S. W. Keckler. "*Non uniform cache architectures for wire-delay dominated on-chip caches*". IEEE Micro. 2003, pp. 99-107.

[3] ITRS. "*The International Technology Roadmap for Semiconductors. International Technology Roadmap for Semiconductors 2008 Update overview*". http://www.itrs.net, 2008.

[4] K. Daewook; M. Kim. ; G. E. Sobelman. "*DCOS: cache embedded switch architecture for distributed shared memory multiprocessor SoCs*". In: International Symposium on Circuits and Systems (ISCAS), 2006.

[5] M. Monchiero; G. Palermo;C. Silvano; O. Villa. "*Exploration of Distributed Shared Memory Architectures for NoC-based Multiprocessors*". Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS, 2006, pp.144-151.

[6] C. Man; et al. "*Distributed Memory Management Units Architecture for NoC-based CMPs*". In: 10th IEEE International Conference on Computer and Information Technology (CIT), 2010, pp. 54-61.

[7] A. Ros; M. Acacio; J. Garcia. "DiCo-CMP: *Efficient Cache Coherency in Tiled CMP Architectures*". In: International Parallel & Distributed Processing Symposium (IPDPS), 2008. pp. 1-11.

[8] N; Magen; A. Kolodny; U. Weiser; N. Shamir. "*Interconnect-power dissipation in microprocessor*". In: System Level Interconnect Prediction (SLIP), 2004, pp. 7-13.

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

[10] G. B. Silva; D. Barcelos; F. R. Wagner. "*Performance and Energy Evolution of Memory Hierarchies in NoC-based MPSoCs under Latency*". In: International Conference on Very Large Scale Integration (VLSI-SoC), 2009.

[11] S. V. Tota; et al. "*MEDEA: a hybrid shared-memory/message-passing multiprocessor NoC-based architecture*". In: Design, Automation, and Test in Europe (DATE), 2010, pp. 45-50.

[12] E. Bolotin; et al. "*The Power of Priority: NoC Based Distributed Cache Coherency*". In: International Symposium on Networks-on-Chips (NOCS), 2007, pp.117-126.

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

[14] E. N. D. Jerger; L. Peh; M. H. Lipasti. "*Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence*". In: International Symposium on Microarchitecture (MICRO), 2008, pp. 35-46.

[15] E. A. Carara; F. G. Moraes. "*Deadlock-Free Multicast Routing Algorithm for Wormhole-Switched Networks-on-Chip*". In: Computer Society Annual Symposium on VLSI (ISVLSI), 2008, pp. 341-346.

[16] E. A. Carara. "*Estratégias para Otimização de Desempenho em Redes Intra-Chip - Implementação e Avaliação sobre a Rede Hermes*". Dissertation Thesis. PPGCC-PUCRS, 2008, 89p.

[17] OpenCores. http://www.opencores.org, 2012.

[18] C. Ewerton. "*Mapeamento dinâmico de tarefas em MPSoCs heterogêneos baseados em NoC*". Thesis. PPGCC-PUCRS, 2009, 170p.

[19] M. Mandelli. "*Mapeamento dinâmico de aplicações para MPSoCs heterogêneos*". Dissertation Thesis. PPGCC-PUCRS, 2011, 106p.

[20] G. Kahn. "*The semantics of a simple language for parallel programming*". In: Information Processing, 1974, pp. 471-475.

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

[22] M. Verma; P. Marwedel. "*Advanced Memory Optimization Techniques for Low-Power Embedded Processors*". Springer Publishing Company, 1st Edition, 2007, 161p.