PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DISTRIBUTED MEMORY ORGANIZATION WITH SUPPORT FOR DATA MIGRATION FOR NOC-BASED MPSOCS

TALES MARCHESAN CHAVES

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science at Pontifícia Universidade Católica do Rio Grande do Sul.

Advisor: PhD. Fernando Gehm Moraes

Porto Alegre, Brasil
2011

FICHA CATALOGRÁFICA

TERMO DE APRESENTAÇÃO

AGRADECIMENTOS

Logo de início, o mestrado parece fácil. Muitos colegas, coisas novas, tudo divertido. Até que ele começa mostrar sua verdadeira face, torna-se difícil e cansativo. Perdemos alguns colegas que abandonam a batalha. Mas guerreiro que é guerreiro persiste, confia em deus, e acaba a sua jornada com sucesso.

A minha jornada foi de grande aprendizado, alegrias, tristezas, cansaços, esforços, mas no fim, valeu muito a pena. Eu faria novamente. Desta forma, gostaria de agradecer a todos que me ajudaram e apoiaram durante estes dois anos (2010-2012). A meus pais, Carlos e Joseane, muito obrigado pelo incentivo, apoio e educação que me deram. A minha companheira, Bruna, obrigado pela enorme paciência, amor e compreensão que teve e ainda tem comigo. Gostaria também de agradecer a meus avós paternos, Ocir e Rodi, que sempre me apoiam, em qualquer momento da minha vida. Também aos meus tios Toni e Lea, que durante a minha infância e adolescência foram como meus segundos pais. Gostaria também de agradecer ao meu primo Rafael, que sempre me ajudou, principalmente desde que me mudei para Porto Alegre. Ao restante da família, meus sinceros agradecimentos pelo pensamento positivo e os momentos de descontração.

Gostaria de agradecer enormemente aos professores do PPGCC, principalmente, ao meu orientador, Fernando Moraes, por me aceitar como aluno do mestrado, me dar a oportunidade de conhecer um mundo novo composto por NoCs e MPSoCs. Gostaria de agradecer também aos professores que me acompanharam e avaliaram durante as várias bancas do mestrado, dentre estes: Prof. Ney Calazans, Prof. Gustavo Fernandes, Prof. Cézar Marcon e o Prof. José Luiz Güntzel.

Gostaria de agradecer aos colegas mestrado, companheiros de noitadas na PUC, fazendo trabalhos. Alguns destes são membros do grupo GAPH, outros não: Eduardo Antunes, Everton Carara, Thiago Raupp, Carlos Petry, Eduardo Wachter, Marcelo Mandelli, Julian Pontes, Rafael Soares, Matheus Moreira e a todos os demais.

Não poderia deixar de citar e agradecer a compreensão, amizade e força dos colegas e amigos do laboratório de R&D da Hewlett-Packard, empresa a qual sou parte desde o início do segundo ano do mestrado. Na empresa, conheci muitas pessoas, e por muitos momentos, estas tiveram enorme compreensão quando precisei me ausentar da HP para me dedicar ao mestrado. Aos demais agradeço pela força. Dentre os principais cito: Alan Souza, Marlon Parizzotto, Joan Alminhan, Leandro Farinati, e a minha querida gerente Maria Bortolotti.

ORGANIZAÇÃO DE MEMÓRIA DISTRIBUÍDA COM SUPORTE À MIGRAÇAO DE DADOS PARA MPSOCS BASEADOS EM NOCS

RESUMO

O avanço da tecnologia de semicondutores possibilitou o desenvolvimento de sistemas intra-chip (SoCs) que integram, em um mesmo chip, diversos elementos de processamento (PEs) e módulos de memória. SoCs que possuem mais de um PE são denominados de sistemas intra-chip multiprocesados (MPSoCs).

À medida que o número de PEs aumenta em um MPSoC, torna-se necesso o uso de técnicas que resultem em um baixo consumo de energia, baixa latência na comunicação e escalabilidade. Em MPSoCs baseados em redes intra-chip, com blocos de memória compartilhada (cache L2), o aumento no número de PEs culmina no aumento da quantidade de acessos realizados a estes módulos. Isto torna a organização de memória um dos componentes mais críticos destes sistemas, tendo em vista que a mesma pode apresentar um alto consumo de energia e alta latência de resposta. Fatores como estes podem limitar o uso e a escalabilidade destes sistemas.

Dentre os fatores que afetam o consumo de energia e a latência da organização de memória de um MPSoC estão: o protocolo de coerência de cache e o mapeamento dos dados de aplicações. Este trabalho propõe a utilização de uma organização de memória cache L1 que possui latência de acesso não uniforme, onde acessos realizados podem ser destinados a diferentes bancos de memória cache L2 (*NUCA – non uniform cache access architecture*). Além disso, o presente trabalho explora os serviços físicos providos pela rede intra-chip, tais como *multicast* e prioridades, para otimizar a implementação de um protocolo de coerência de cache baseado em diretório. Os resultados obtidos mostram uma redução média de 39% no consumo de energia de comunicação e 17% de latência em transações do protocolo de coerência quando explorando serviços físicos oferecidos pela rede intra-chip.

Além disso, a fim de melhorar o posicionamento de dados de aplicações, é proposto um protocolo de migração de dados que posiciona os dados utilizados por uma região de PEs em um banco de memória cache L2 próximo. Com a utilização deste protocolo, é possível obter uma redução média de 29% no consumo de energia nos acessos à memória cache L2.

Palavras Chave: MPSoCs baseados em NoC, Sub-sistema de Memória, Coerência de Cache, Migração de Dados.

DISTRIBUTED MEMORY ORGANIZATION WITH SUPPORT FOR DATA MIGRATION FOR NOC-BASED MPSOCS

ABSTRACT

The evolution in the development of semiconductor technology has enabled the development of System-on-Chip (SoCs) that integrate several processing elements (PEs) and memory modules in a single chip. SoCs that integrate several PEs are refered as Multiprocessor System-on-Chip (MPSoCs).

As the number of PEs increases in an MPSoC, techniques that present low energy consumption, low latency and scalability become necessary. In NoC-based MPsoCs that adopt the Shared Memory model in the L2 cache, as the number of PEs increases, the number of accesses done to the memory modules also increases. This makes the memory organization one of the most critical components of the system because it can present high energy consumption and high latency. Such factors may limit the use and scalability of MPSoC systems.

Among the factors that contribute to increasing latency and energy consumption in the memory organization are: the cache coherence protocol and the mapping of application's data. This work proposes the use of a cache memory organization that presents non-uniform access latency, where accesses done to the L1 cache can target different L2 cache banks (*NUCA – non uniform cache access architecture*), as a function of the address being accessed. Additionally, the present work proposes the exploration of the physical services provided by the network-on-chip, such as multicast and priorities, to optimize the implementation of a directory-based cache coherence protocol. The obtained results show an average reduction of 39% in communication energy consumption and 17% reduction in latency for transactions of the cache coherence protocol when exploring NoC services.

To improve placement of application's data, a data migration protocol is proposed. The goal of the protocol is to approximate L2 cache blocks to PEs that are mostly accessing it, moving the blocks to a closer L2 cache bank, if available. By using the data migration protocol, an average reduction of 29% was obtained in the energy consumption of cache accesses.

Keywords: NoC-based MPSoCs, Memory subsystem, Cache Coherence, Data Migration.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

2D	Two dimensional
ACK	Acknowledgment
API	Application Programming Interface
CMP	Chip Multi Processor
CPU	Central Processing Unit
CSM	Centralized Shared Memory
DCOS	Directory Cache On Switch
DDR	Double Data Rate
DM	Distributed Memory
DMA	Direct Memory Access
DRAM	Dynamic random access memory
DSM	Distributed Shared Memory
DSP	Digital Signal Processor
DVFS	Dynamic Voltage Frequency System
FBS	Free Block Searcher
FFT	Fast Fourier Transform
FSM	Finite State Machine
GPS	Global Positioning System
I/O	Input/Output
IC	Integrated Circuit
IP	Intellectual Property
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
KPN	Kahn Process Network
MMU	Memory Management Unit
MOESI	Modified Owned Exclusive Shared Invalid
MPI	Message Passing Interface
MPSoC	Multiprocessor System on Chip
MSI	Modified Shared Invalid Protocol
NI	Network Interface
NoC	Network on Chip
NUCA	Non Uniform Cache Access
PC	Program Counter
PCB	Printed Circuit Board
PE	Processing Element
POSIX	Portable Operating System Interface
QoS	Quality Of Service
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
RTOS	Real Time Operating System
SOC	System on Chip
TCB	Task Control Block
TSP	Translation Sharing Partition
UART	Universal asynchronous receiver/transmitter

VCI	VCI interface
VHDL	VHSIC Hardware Description Language
VTC	Virtual Tree Coherence

SUMMARY

1 INTRODUCTION

During the last decades, the field of embedded systems evolved significantly. Most of the evolution is based on the increase of the amount of transistors that can be placed at the same silicon surface. This evolution allowed modifications on the architecture of embedded systems, which evolved from discrete elements on a PCB (Printed Circuit Board) to SoCs (System on Chip).

A SoC is an integrated circuit that implements most or all of the functions of a complete electronic system [JER05]. SoC designs are based on IP (Intellectual Property) modules integrated on silicon chips, as described by Gupta et al. [GUP97] and Bergamaschi et al. [BER01]. Gupta et al. [GUP97] define an IP as a pre-designed, pre-verified hardware piece that can be used as a building block for large and complex applications on an Integrated Circuit (IC). The design of systems based on IPs reuse is an attempt to reduce the time-to-market, which is the time that a new product takes to enter the market. SoCs are usually denoted MPSoCs (Multiprocessor Systems on Chip) when constituted by more than one processing element (PE). According to [JER05] most SoCs are in fact MPSoCs because it is too difficult to design a complex SoC without making use of multiple PEs.

1.1 MPSoCs

The architecture of an MPSoC system is composed by a combination of: PEs, memory elements and a communication infrastructure. Considering processing, MPSoCs are classified in two classes:

- Homogeneous: in this class all PEs of the MPSoC have the same architecture. The fact that all PEs have the same architecture facilitates task migration, for instance, because there is no need for translating the binary code of a given application to other architecture. Examples of homogeneous systems can be found in [SAI07][CAR09];
- Heterogeneous: in this class the architecture of at least one PE is different from the others. The advantage of this class is, for example, to develop real time systems. For instance, an MPSoC system can have an ARM processor for handling system tasks and a DSP processor for handling 3G signals. Examples of heterogeneous systems can be found in [MON08][LIM09].

Considering the communication infrastructure used to interconnect PEs and memory elements, three infrastructures are commonly employed in MPSoC: dedicated wires, shared busses and networks-on-chip (NoCs).

Dedicated wires are effective for systems with a small number of cores, but the number of wires around the core increases as the number of PEs in the system grows.

Therefore, dedicated wires have poor reusability and flexibility. A shared bus is a set of wires common to multiple cores. This approach is more scalable and reusable, when compared to dedicated wires. However, busses allow only one communication transaction at a time, thus all cores share the same communication bandwidth in the system and scalability is limited to a few dozen of IP cores [KUM02]. Using separate busses interconnected by bridges or hierarchical bus architectures may reduce some of these constraints. Nonetheless, scalability remains a problem for hierarchical bus architectures.

MPSoC designs are mostly based on Networks-on-Chip (NoCs) as they provide scalability, high bandwidth, energy efficiency, reliability, parallel communication and scalable design exploration space [MIL04]. A NoC is an on-chip network composed by switches, which are connected among themselves by communication channels.

1.2 Memory system in MPSoCs

According to [KAN05], one of the most critical components that determine the success of an MPSoC-based architecture is its memory system. This assertion is justified by the fact that applications might spend several cycles waiting for the conclusion of read/write memory operations.

The technology employed to develop processing elements advances faster than that used in the development of memory components, which enables PEs to operate at higher frequencies. This disparity allows PEs to consume data at rates not possibly achieved by DRAM memories, creating a *performance gap*. To decrease this gap, a solution commonly applied in high-end microprocessors is the use of static memories and a memory hierarchy. In a hierarchy, several levels of memories are used to decrease average memory access latency. The main idea is to place faster but smaller memories closer to processors and slower but larger memories in further levels. The smaller memories contain a subset, which consists usually of the most accessed data from the data stored in the adjacent further level. 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 stall while waiting for an instruction or data from main memory. Another interesting point in the use of caches is that they can reduce energy consumption, once memory accesses are local, avoiding transactions on a bus/network-on-chip that would be necessary to bring a block of data from main memory/secondary memory.

In multiprocessor systems, the use of cache memories might give raise to the problem of cache incoherence, which refers to the consistency of data stored in local

caches of a shared resource. To avoid incoherence, a cache coherence protocol must be implemented.

1.3 Data Migration in MPSoCs

MPSoC systems tend to have hundreds of elements [ITR11]. As the system size increases, there is a need to develop mechanisms that optimize these systems in several aspects, such as: energy consumption, latency and resource allocation. To decrease energy consumption, techniques such as DFVS may be applied to decrease the energy consumption of PEs when they are executing low priority tasks or are idle. Also, in multiple memory bank systems, there is the possibility of migrating data from a bank to another to approximate them to the processors that are mostly accessing those data. Accesses done to the data after migration tend to consume less energy in communication and take less time as the distance between the memory bank and the PEs decreases.

1.4 Goals

The goals of the current work are divided in strategic and specific. As a strategic goal this work proposes solutions to optimize the use of distributed memory blocks in a NoC-based MPSoC through the use of data migration technique. The use of data migration techniques in these systems brings two main benefits: smaller energy consumption in the communication infrastructure (NoC) and also decrease the traffic in certain areas of the MPSoC leading to a smaller probability of congestion.

The specific goals of this work are:

- Development of a memory controller for the HeMPS platform. This requires the development of a network interface for integrating the memory with the NoC. The memory controller stores data from applications and can be initialized at design time.
- Development of a data cache controller and its integration with the PLASMA processor.
- Development of a cache coherence protocol for the HeMPS platform, exploring the benefits (decrease in latency and energy consumption) that physical services, such as multicast and priorities, can provide to the implementation of the protocol.
- Evaluate the implementation of the cache coherence protocol through the use of synthetic and real applications.
- Development of parallel benchmarks for the HeMPS platform. The benchmarks allow the evaluation of the techniques and architectures proposed in the current work.

- Implementation of a cache API, at the application level, in the HeMPS microkernel, which allows applications to access the data cache.
- Development of a data migration protocol for a NoC-based MPSoC. Evaluate the benefits related to energy consumption and latency of memory accesses when using the data migration protocol.

1.5 Contributions

The contributions of the current work include:

- The development of a distributed memory hierarchy for NoC-based MPSoC (Chapter 4);
- The development of an energy-efficient cache coherence protocol for NoC-based MPSoCs (Chapter 5);
- Development of a data migration protocol for NoC-based MPSoCs (Chapter 6).

1.6 Document Organization

This document is organized as follows. Chapter 2 presents the state of the art for three areas of NoC-based MPSoCs: memory organization; cache coherence protocols and data migration. Chapter 3 presents the MPSoC platform used as reference for the development of this work. Chapter 4 presents the cache controller developed for the HeMPS Platform. Chapter 5 presents an energy efficient cache coherence protocol for the platform described in Chapter 3. Chapter 6 presents the data migration protocol proposed to the HeMPS Platform. Chapter 7 presents results related to Chapters 4, 5 and 6. Finally, Chapter 8 presents conclusions and directions for future works.

2 RELATED WORKS

This Chapter presents state-of-the-art works for 3 topics related to the memory architecture of NoC-based MPSoCs: memory organizations, cache-coherence protocols and data migration algorithms.

2.1 Memory Organization

Memory organization determines the structure, location and number of memory banks in the MPSoC architecture. The reviewed literature presents in its majority three types of memory organizations: centralized shared memory (CSM), distributed shared memory (DSM) and distributed memory (DM). In the CSM organization, there exists only one physical memory bank and a unique (global) address space. The memory bank is shared among all PEs of the system, and communication between PEs is done through a common region of memory. The problem presented by the CSM organization is scalability. As the number of PEs increases, the number of requests issued to the memory also increases [MAN10], which induces congestion on a given region of the NoC, resulting in unpredictable access latency. Very few works have been found in the literature using the CSM organization. Consequently, it is omitted in the next subsections.

DSM [HEN03] is a memory organization that allows multiprocessors to support a single shared address space that is implemented with physically distributed memory. This organization is more scalable when compared to CSM, as the overall bandwidth of the memory is increased. Although, in case several PEs access the same memory address, one of the memory controllers will be overloaded.

In a non-shared memory system, such as DM, each PE has a private memory, which is accessed exclusively by the PE it is associated with. Communication among PEs occurs through message-passing only.

The next subsections present state-of-art works for DSM and hybrid organizations. In hybrid organizations, concepts of both shared and distributed organizations are applied to the memory organization.

2.1.1 Distributed Shared Memory

According to Kim et al. [KIM06] the higher latency presented by the routers in a NoC directly affects the way distributed shared memories are designed. To address the latency issue, the authors propose a switch architecture for providing low-latency for cache coherence operations on a distributed shared memory MPSoC platform. The authors denote the platform as DCOS, which stands for Directory Cache On a Switch. The platform is composed of a three-level memory hierarchy, where a private L1 cache, which stores data and instruction, constitutes the first level cache. The second level is a shared

L2 cache, which is a write-back cache with write-allocate. The third level memory is a shared memory that can be either located on-chip or off-chip.

To address the cache coherence problem, a directory based MSI (modified-shared-invalidate) protocol is used. To decrease latency of cache coherence operations, the authors propose the insertion of both Shared Memory directory and L2 cache directory cache inside the switch, as shown in Figure 1. The results show a substantial reduction of 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. This may represent an important area overhead of the router, which is not evaluated in the work.

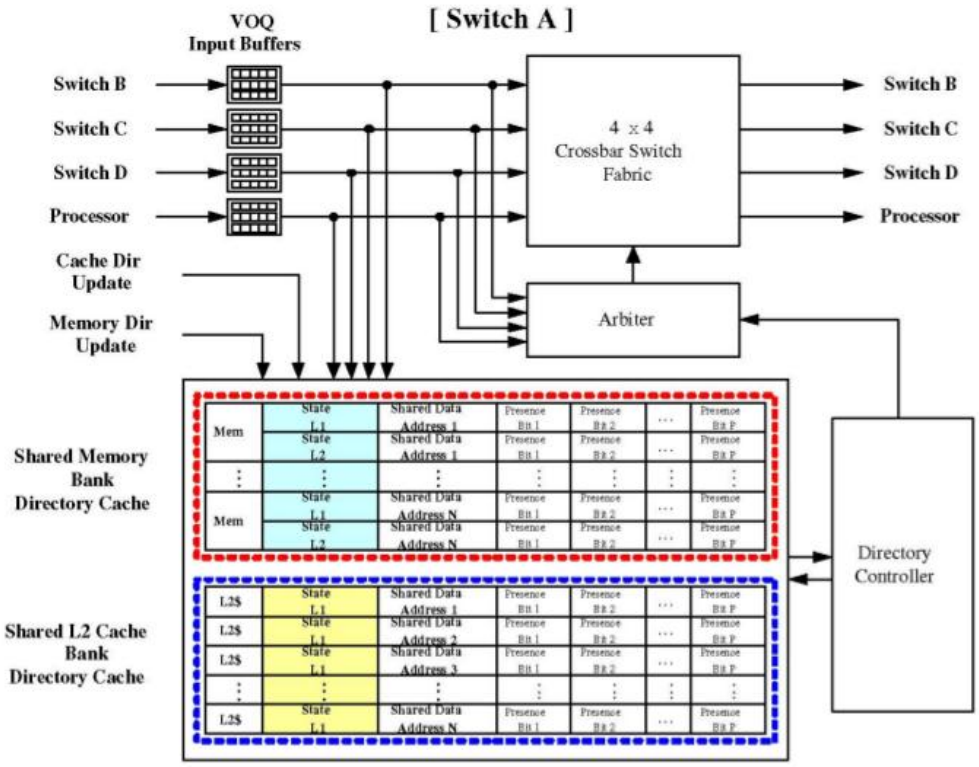


Figure 1 – DCOS switch architecture proposed by [KIM06].

Yuang et al. [YUA08] discuss the implementation of a distributed shared memory and control mechanisms, such as synchronization and consistency model, for a NoC-based MPSoC. There are two types of nodes in the system: master and slave. Master nodes, shown in Figure 2, contain: a microprocessor, a network interface, and a private memory that stores local data and instructions. All these modules are attached to a local bus. Slave nodes may be used for storing shared data and for inter-processor communication. Slave nodes contain: a shared memory, a semaphore memory and a network interface. The semaphore memory has two main purposes: synchronize inter-processor communication and control memory consistency, which corresponds to handling of memory accesses and

in order execution of the accesses. To ensure synchronization in memory accesses, a lock mechanism is employed. To ensure memory consistency, a write with response scheme (ACKs) is provided. To evaluate performance of the implemented memory system, the authors simulated the execution of two applications (FFT and Matrix Pipeline) on a SystemC cycle accurate NoC Platform simulator. The results show a speed up of three when the size of processed data increases. The authors do not provide execution time and energy consumption for applications.

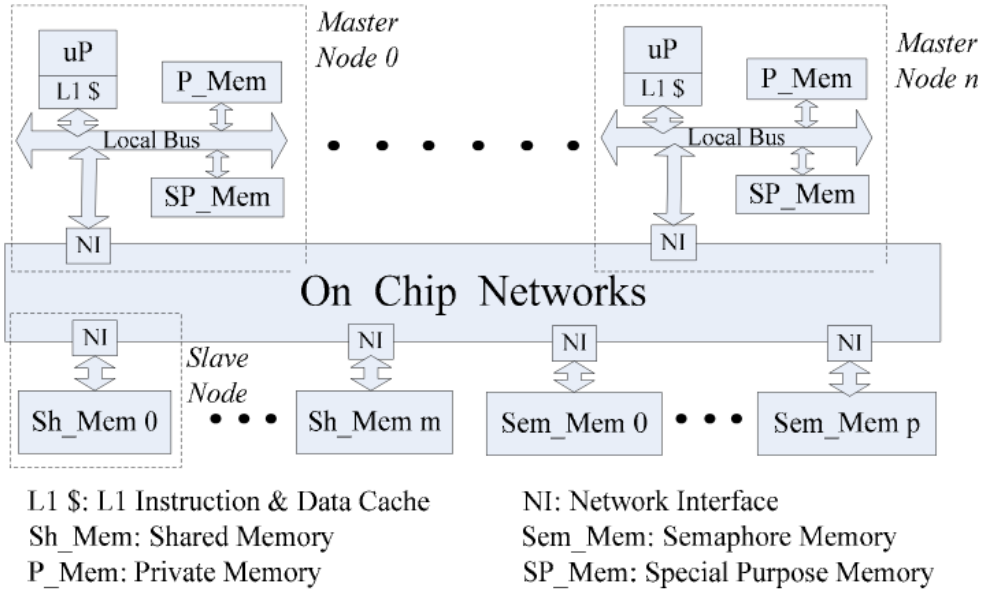


Figure 2 – Distributed Shared Memory architecture proposed by [YUA08].

Man et. al [MAN10] discuss the problems of using a unique, centralized Memory Management Unit (MMU), shared by PEs in a NoC-based Chip Multiprocessor (CMP). According to the authors, as the number of PEs increases, the number of memory references also tends to increase. Consequently, the sequential structure of a centralized MMU can become the bottleneck of the system. To tackle this problem, a distributed MMU is proposed. Distributed MMU scheme 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 of these modules, both memory bandwidth requirements and NoC communication traffic may be decreased. In the design proposed by [MAN10], PEs are grouped into a Translation-Sharing Partition (TSP) that contains a MMU responsible for handling requests of PEs belonging to that partition. Figure 3 shows an example of a 5x5 2D mesh NoC having 3 TSPs and 4 MMUs.

The authors in [MAN10] conducted simulations to evaluate throughput and delay in a 5x5 mesh network as a function of different number and distribution of MMU nodes. The traffic generated on the experiments was synthetic and based on a Poisson distribution. Results showed that the increase of MMU nodes decreases the delay of PE nodes significantly. Therefore, it is possible to conclude that network latency of distributed MMU is very sensitive to the size of TSPs and number of MMUs.

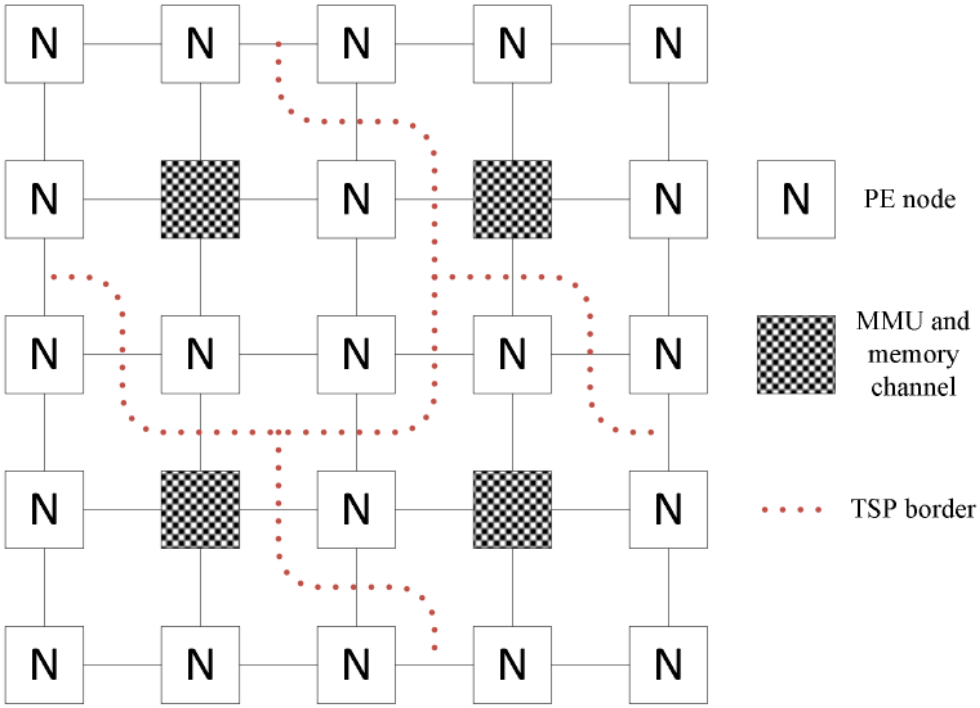


Figure 3 – NoC view of TSP partitions and positioning of MMU and PE nodes.

Monchiero et al. [MON06] present a NoC-based MPSoC platform, modeled at the GRAPES Plataform [MON08]. The platform is composed of ARM7 PEs, memory elements, memory and interrupt controllers, as shown in Figure 4. Each PE has a L1 data and instruction cache. Memory elements can be of two types: shared memory banks or a L2 cache, centralized in a unique node connected to the NoC. To access one of the shared banks, a given PE must send a request to a HW Memory Management Unit (HWMMU) that is responsible for managing shared memory access. The operations supported by HWMMU are: memory allocation (MMU_MALLOC); memory release (MMU_FREE); copy and move.

The authors performed experiments to evaluate latency and energy consumption at the GRAPES Platform executing a set of applications from SPLASH-2 and PARMACS benchmarks. Different scenarios where tested by varying parameters of memory architecture (number of shared memory banks) and NoC topology (ring, spidergon and mesh). The results showed that a decrease in latency is obtained when there is an increase on the number of shared memory banks. Although, when there are more than four memory banks the communication cost overcomes the gains of having more memory banks. According to the authors, latency reduction is obtained due to the decrease on the memory contention and decrease of transactions on the network. Among the three NoC topologies evaluated (mesh, spidergon and ring), mesh presented the shortest execution time.

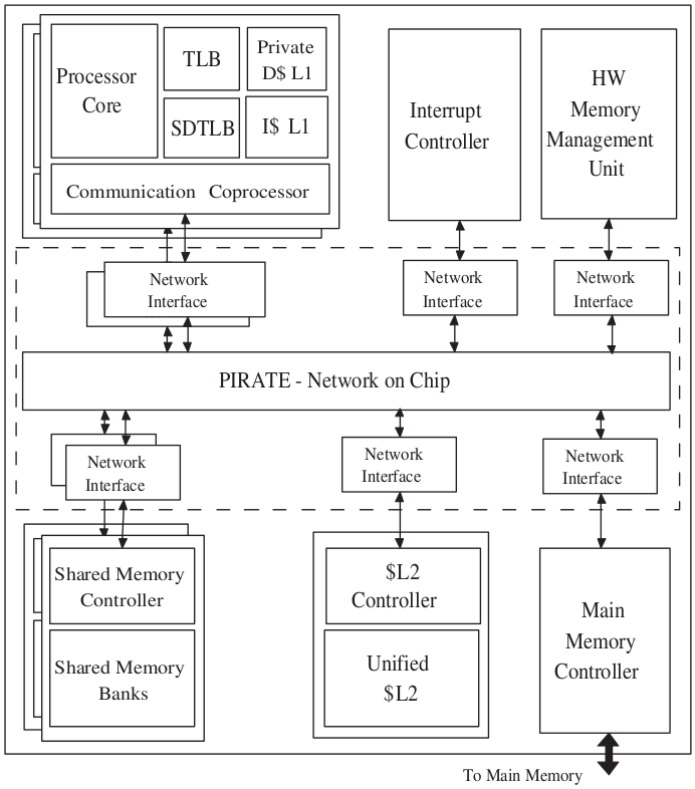


Figure 4 – Multiprocessor architecture proposed by [MON06].

Silva et al. [SIL09] present an evaluation of four different memory organizations for a NoC-based MPSoC to compare their performance and energy efficiency. The evaluated organizations are: (i) a distributed memory that includes processors with local private memories; (ii) a shared memory accessed by all processors; (iii) a distributed shared memory; (iv) a physically shared but logically distributed memory (nDMA). The experiments conducted by the authors evaluate the behavior of the four memory organizations when the traffic on the network increases, what induces higher packet latency. Results showed that for applications with high communication demands, the distributed memory organization presents the highest tolerance to communication latency. Although, applications with low communication workload this organization seems to suffer more than others due to network latency. Regarding performance, the nDMA memory shows the smallest variation of performance reduction (22% to 33%) when NoC traffic increases from 10% to 20%.

2.1.2 Hybrid organizations

Tota et al. [TOT10] present the MEDEA framework, which consists of a configurable NoC-based MPSoC modeled in SystemC. This framework supports two programming models: shared memory and message passing. Instructions fetch and load/store operations adhere to the standard shared memory model whereas synchronization and data exchange among cores may occur, for performance or cost reasons, by means of explicit low-latency message passing technique. The system is composed of three basic

elements: a NoC, RISC-like PEs and the Multiprocessor Memory Management Unit (MPMMU) that interfaces an external DDR memory. The NoC has a torus topology and implements a deflection-routing (Hot Potato) algorithm that allows each flit to be routed independently. Each PE consists of a configurable Tensilica XtensaLX, which was used to implement the MPI message passing interface. The MPMMU is a special processor that handles shared-memory transactions (reads/writes). The global shared-memory is divided into two logic segments: shared and private area. Each core has a private segment and can access the shared segment through the use of a lock mechanism. Experiments were conducted by the authors using the use of the Jacobi algorithm. The authors reached the conclusion that the hybrid approach scales better and uses silicon area for additional cores instances in a more efficient way compared to a standard shared memory approach.

2.1.3 Memory organizations comparison

Table 1 shows a comparison of the reviewed works in memory organizations considering four topics. Most works adopt a 2D Mesh NoC. Only one of the reviewed works ([MON06]) evaluates the impact of the NoC topology in the memory performance. Most works consider the evaluation of the proposed platform in SystemC or higher-level simulation platforms. Although high-level models accelerate implementation, they do not provide high accuracy. It is also possible to notice that most works do not evaluate the energy consumption considering the placement of the memory in the MPSoC, and neither evaluate the cost of transactions on the network. This work proposes the exploration of DSM organization, modeled in synthesizable VHDL, where L2 data cache banks characterize the shared memory.

2.2 Cache coherence protocols for NoC-based MPSoCs

The use of cache memories in multiprocessor systems may raise the cache coherence problem. This problem occurs when PEs have local copies from a shared resource, such as a shared memory, which is modified by other PEs. In this way, local copies may become inconsistent. If any modification is done locally to a cached copy of the shared resource, changes must be propagated to other caches; otherwise PEs might process inconsistent data. To avoid this problem, there are two possible solutions: do not allow caching of shared data or implement a cache coherence protocol. MPSoC systems are mostly designed for parallel applications. In these applications, some data are shared between several PEs, and communication may occur through accessing a common region of memory. Additionally, applications that perform a significant amount of communication may suffer from NoC latency [TOT10]. In these cases, the use of shared memories and cache coherence protocols may help decreasing the communication overhead between PEs.

Table 1 - Comparison of the reviewed works considering memory organization.

Memory Organizations for NoC-based MPSoCs				
Author	**NoC topology**	**Memory Organization**	**Modeling Level / Simulator**	**Goal**
[KIM06]	2D Mesh	L1 and L2 + Shared Memory banks	RSIM Simulator	Decrease latency of the cache coherence protocol
[YUA08]	2D Mesh	SM banks + semaphore memory	SystemC	Explore Synchronization and Memory consistency
[MAN10]	2D Mesh	Distributed MMU (Memory Management Unit)	-	Evaluate the placement of MMUs
[MON06]	Ring, spidergon, 2D Mesh	L1 and L2 + SM banks	Grapes Platform	Implement a HWMMU
[SIL09]	2D Mesh	4 types: SM, DSM, nDMA and private memory	SystemC	Evaluate memory performance under latency
[TOT10]	Torus	L1 + L2	RTL and SystemC	Explore a hybrid architecture
This work	2D Mesh	L1 + L2	RTL	Evaluate DSM and allow data migration algorithm

Cache coherence protocols can roughly be classified in two classes: software-based protocols and hardware-based protocols. Hardware protocols include snoopy-based protocols, which are not appropriate for NoC-based MPSoCs, as they require that all accesses to the memory must be visible for all memory banks and PEs of the system. Global visibility is very efficient for busses, but NoCs are composed of segmented links interconnected through routers. To provide global visibility all memory accesses need to be broadcasted to all processors, considerably increasing the amount of traffic in the NoC. In addition, according to the energy model proposed by Hu et al. [HU03], the amount of traffic in the NoC causes an increase in the energy consumption of the NoC.

Directory-based protocols are another type of hardware protocol. In such protocols, before modifying a given block of the memory, PEs must access a centralizing point, the directory, to obtain the status of the block. According to the status, invalidation/update messages are sent to guarantee system coherence. Directory-based protocols are more attractive for NoC-based MPSoCs as they do not require global visibility for all memory accesses.

Software-based protocols, usually, rely on explicit synchronization, and do not ensure cache coherence on the granularity of individual memory accesses, but on groups of memory accesses. Additionally, software protocols rely on the use of compiled-time information to predict run-time behavior. The performance of such protocols is significantly degraded when predicted hits do not happen at runtime [ADV91].

Yuang et al. [YUA08] proposes a hierarchical cluster based cache coherence protocol for large-scale NoC-based shared memory architectures. The nodes of the NoC are grouped into units named clusters. Each cluster consists of a group of L1 cache banks and one L2 cache bank, which is named HEAD and contains a local directory. The HEAD node is located at a NoC router that has the average minimum distance to all members of the cluster. Figure 5(b) shows a 5x5 mesh NoC, where each circle comprises a cluster. The HEAD of each cluster is painted in black. The shared memory is located off-chip and maintains the global directory, as shown in Figure 5(a). Cache coherence is enforced hierarchically. Similarly to the directory-based cache coherence protocol, the global directory is a flat, full-map directory, which stores the status of each block. The difference is that instead of keeping which PEs have copies of the blocks, the global directory keeps which clusters have a copy of each block. Intra-cluster, the local directory stores which nodes of the cluster (L1 caches) contain a copy of each block.

The advantages of using such protocol for enforcing coherence on the platform is the reduction on the number of hops traversed by messages of the coherence protocol (Longest Manhattan Distance and Long Distance Travel). Another advantage is the reduction on the space required for storing the directory. A disadvantage of the clustered approach is that each application must be mapped onto a single cluster, which is not always possible, what can degrade performance significantly.

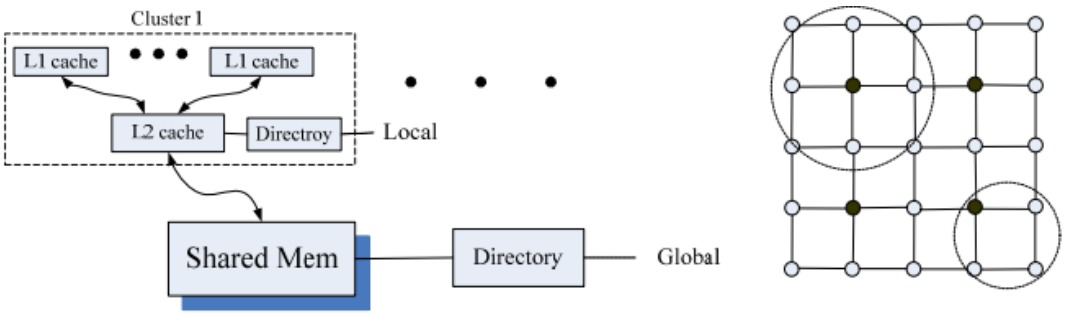


Figure 5 – In the left, it is shown the architecture of the HEAD node. In the right, each circle comprises a cluster region [YUA08].

Jhalani et. al [JHA11] propose the inclusion of a data pattern, called migratory data, to the directory-based cache coherence protocol to make the protocol adaptive. Migratory data is a data pattern exhibited by parallel programs where blocks move between PEs and at each PE the block is in exclusive state, which means that each PE tries to write on the block. To detect that a block follows the migratory pattern, one of the following conditions

must arrive: (i) at the time of a write-hit, a block appears in shared state with exactly two cached copies and the processor currently requesting the invalidation operation is not the same which requested invalidation previously; (ii) at the time of a write-miss, there is only one copy of this block and it is modified.

By detecting that a given block is migratory, the protocol used to maintain its coherence is the adaptive cache coherence protocol, otherwise a regular version of the invalidate protocol is used. The adaptive protocol reduces execution time of applications and reduces the traffic generated on the NoC by the cache coherence protocol. To evaluate the protocol, experiments were conducted with the use of the RSIM simulator, with sizes of 32KB (2-way associative) for L1 cache and 1MB (4-way associative) for L2 cache, both having block sizes of 32 bytes. Five SPLASH benchmarks were simulated. The results showed that the adaptive protocol performed better in some benchmarks, but worse in others compared to the regular MESI protocol.

Jerger et al. [JER08b] proposes a method for cache coherence in multicore architectures, 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. Figure 6 shows the architecture of each node. For each region, a virtual tree of the nodes that shared that region is created. The routers used information stored in the VTC Table (Figure 6) to obtain information about the virtual trees and their members. Every time a given region is accessed by one of the nodes, a request is sent to the root of the tree, which in turn, requests the data to the node holding the most updated copy of it. This request is done through a multicast message traversing the tree.

The implementation of the multicast according to the tree topology decreases latency when compared to unicast-based implementations. The authors compare their implementation to a directory-based implementation and to another implementation based on a greedy algorithm. Results show a performance improvement of 25% of this implementation over the directory-based one and 11% over the greedy one.

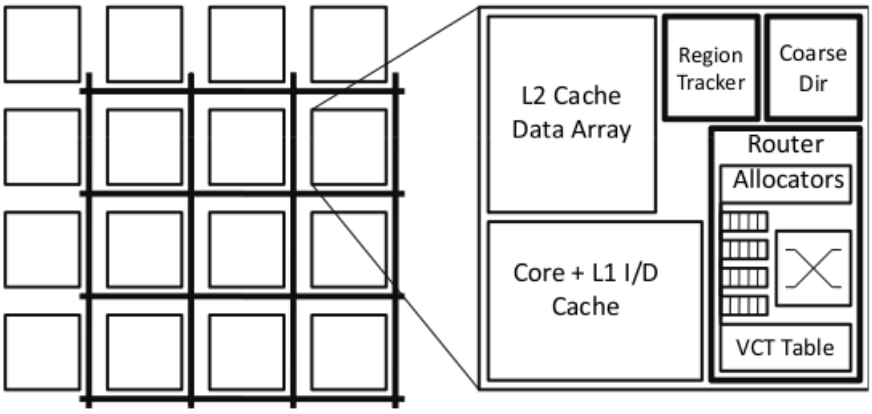


Figure 6 – PE architecture with L2 cache and VTC modules [JER05].

Chtioui et al.

[CHT09] present the development of a dynamic hybrid cache coherence protocol for shared-memory NoC-based MPSoCs shown in Figure 7. According to the authors, existing regular protocols such as invalidation and update, do not take into account the patterns of data accesses performed by applications during runtime, leading to unnecessary operations of update/invalidate on a given block. This might affect the amount of traffic on the NoC (e.g. generated by unnecessary update operations) and, consequently, increase the energy consumption. In response to that, the authors propose a protocol based on the traditional directory-based protocol that adapts itself to the way in which the data is used, alternating between an update to invalidate protocol and vice-versa. It is considered as dynamic because the threshold value that determines if the protocol should be alternated changes during runtime. Operations of the cache coherence protocol are transmitted to PEs through the use of a dedicated bus, as shown in Figure 7.

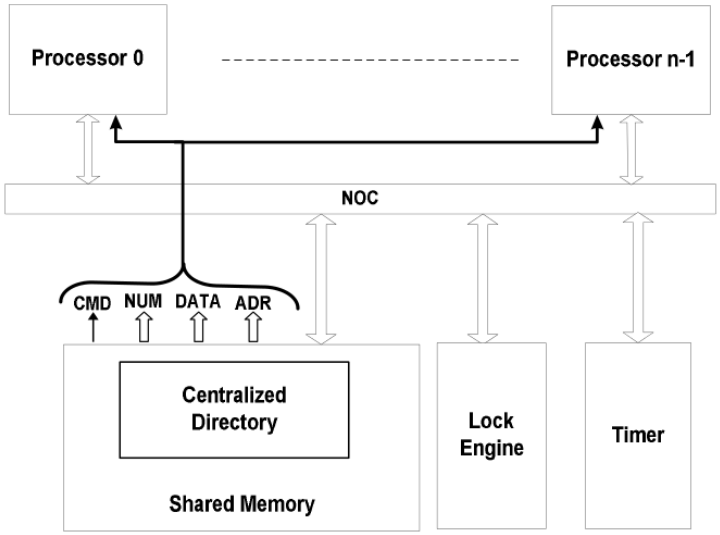


Figure 7 – NoC-based MPSoC architecture with an exclusive bus for cache coherence transactions [CTH09].

The experiments were conducted with the use of the SoCLib platform running two benchmarks: Fast Fourier Transform and matrix multiplication. The simulated architecture is composed by MIPS R3000 processors, containing I-Cache and D-Cache, two banks of shared memory, one for instruction and the other for data. All modules are connected to a crossbar NoC through a VCI interface. Results show that the most significant gain of the hybrid protocol is the reduction on energy consumption when compared to regular invalidate and update protocols. Also, the hybrid protocol reduces both cache misses, compared to invalidation protocol, and unnecessary updates, compared to update protocol. The use of a dedicated bus to cache coherence operations affects the scalability of the system.

Petrot et al. [PET06] present research for the problems of cache coherency and memory consistency in NoC-based shared-memory MPSoCs. According to the authors, in

SoC architectures, software-oriented cache coherence protocols provide the best trade-off between complexity and performance as they incur no hardware cost at all. The software-solution used in [PET06] makes shared data uncached and local, non-shared data, cached. Each application is written as a set of POSIX threads. The memory is partitioned into two types of segments: local segments, which store private data for each processor and shared segments, which store shared variables such as communication buffers. Memory consistency is guaranteed with the use of the release consistency model. The experiments conducted by the authors compare the proposed solution to a fully uncached solution. The results show a performance improvement of a factor of 3, versus the fully uncached solution when running a Motion-JPEG decoder for 48x48 pixels movies of 24 images. Another software solution for a cache coherence protocol in MPSoCs is presented by Ophelders et. al [OPH09].

Most of the works of the state of the art, shown in Table 2, adopt a shared memory organization and the platform evaluated is usually modeled in high abstraction levels. Some works propose the use of traditional protocols for NoC-based MPSoCs, such as the directory-based protocol. Most NoC-based MPSoCs designs including a memory hierarchy do not explore the services offered by NoCs to improve performance, using the NoC as simple bus. An exception to this remark is the work of [JER08a] which proposes a protocol based on trees implemented in the NoC. Several NoCs implement multicast messages at the physical level [JER08a][CAR08]. The use of multicast messages by cache-coherence protocols has several advantages, as the reduction on the number of cycles required to perform write-invalidate operations, as well as to reduce the traffic generated by these protocols on the NoC.

Table 2 - Reviewed works in cache coherence protocol for NoC-based MPSoCs.

Cache-Coherence Protocol in NoC-based MPSoCs			
Authors	**Abstraction Level**	**Memory Organization**	**Cache-Coherence Mechanism**
[YUA08]	SystemC	Distributed Shared Memory	Atomic writes
[JHA11]	RSIM Simulator	Shared Memory	Adaptive directory-based
[JER08a]	PHARMsim	-	Virtual tree coherence
[CHT09]	SystemC (SoCLib)	Shared Memory	Dynamic Hybrid Cache Coherence based on a shared directory
[PET06]	Cycle Accurate Level	Shared Memory	Shared data is not cached avoiding the need for a coherence mechanism
This work	RTL	Distributed Shared Memory	Multicast optimized directory-based cache coherence

2.3 Data Migration Algorithms

Wire delays play an important role in cache design [BEC04]. As semiconductor technology advances the number of cycles required to transmit data in the chip increases. Therefore, wire latency increases fast. To reduce this latency in cache design, designers have split L2 caches into several cache banks. Such architectures are named NUCA (Non Uniform Cache Access) and have varying access latency. Latency is a function of the distance between the processor requesting data and the cache bank that will provide the data.

Some of the key design points of data migration algorithms include:

- when the migration occurs, which directly influences the way that cache accesses are accounted;
- the granularity of the migration, for instance an algorithm could migrate one or more blocks of a cache at once;
- which bank (target) will receive the migrating block once detected that the migration should occur;
- how blocks are retrieved in the system before and after migration (search algorithm).

Beckmann et al. [BEC04] argue that block migration is less effective for CMPs (chip multiprocessors) because 40-60% of L2 cache hits in commercial workloads are satisfied in central banks, which are equally far from all processors. Based on this assumption, the Authors propose a combined solution which uses a data migration protocol combined with optimized layout techniques. The main goal of their work is to optimize performance. In our work, we consider MPSoC applications, which may not present the same pattern that is present in commercial workloads. Additionally, the main focus of our work, which is not addressed by [BEC04], is the energy consumption of cache accesses.

Nafziger et. al. [NAF10] discuss the cache access latency problem in NoC-based systems as the number of PEs increases. According to the authors, methods proposed in the literature do not consider massively parallel architectures, with hundreds of cores, and also do not take into consideration the dynamic patterns of accesses to caches. According to [NAF10], three factors must be considered to reduce latency in cache accesses: NoC layout; data migration algorithm; and future access prediction. The layout adopted by the authors is a hybrid layout in which a cluster of PEs is connected to a L2 cache bank through a bus, shown in Figure 8. The L2 cache banks are connected to a 2D mesh network. The system is composed by a varying number of clusters. Data are migrated between L2 cache banks according to the Directional Migration Algorithm (DMA). In this approach, a block is migrated from a bank to its neighbor bank that is one hop closer to the PEs accessing it. To avoid unnecessary migrations, it only occurs once the number of

accesses done by a cluster of PEs reaches a threshold of 4 accesses. Additionally, an algorithm called Active Neighbor Migration is implemented on top of DMA to migrate blocks that will be likely accessed in the future by a group of PEs. A problem in this approach is that applications may not be entirely mapped into a cluster. This may cause the problem of migrating a cache block several times between cache banks due to data sharing between two tasks of the same application.

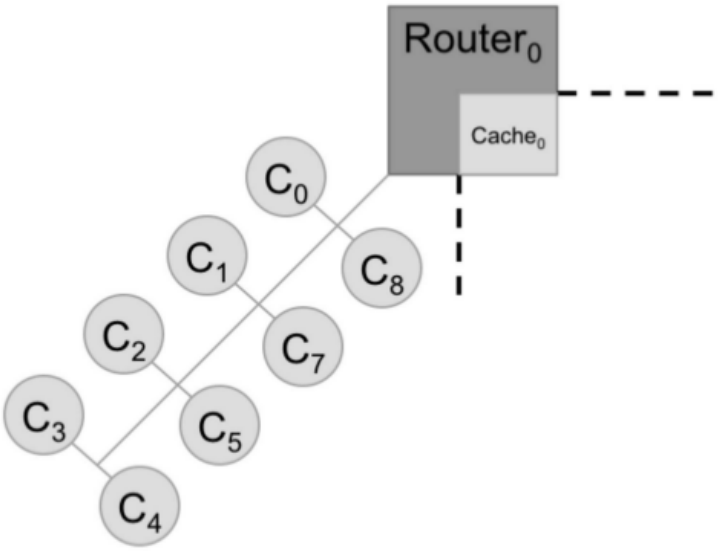


Figure 8 – Cluster architecture proposed by Nafziger et al. [NAF10].

Eisley et al. [EIS08] propose the use of data cache migration to reduce off-chip memory access. Every time a block is evicted from L1 cache, a migration algorithm uses information embedded in the routers of the NoC to steer the migrating cache blocks towards underutilized L2 caches. The information present in the routers (Score Table in Figure 9) is updated periodically from immediate neighbors and is based on estimates of L2 cache utilization in a particular direction. The authors justify this action by saying that evicted blocks probably will be used in the near future. In this way, a cache miss is avoided. The problem with this approach is that as any L2 cache is eligible for receiving an evicted cache block, evicted blocks may interfere in the performance of other applications.

Kandemir et al. [KAN05] models the optimal data placement in the L2 cache space problem as a two dimensional post-office placement problem. For each cache block that is shared by more than one PE, the location of the PEs and the number of times that each has accessed the block (*weight*) are used to calculate an approximate optimal placement for the data. The main goal of finding the proper location to a given cache block is to decrease the number of migrations that are required to place a given block in its proper location. The problem with this approach is the additional complexity for finding the optimal location for each block, which can incur additional area overhead. Although, preventing several unnecessary data migrations may help reducing energy consumption.

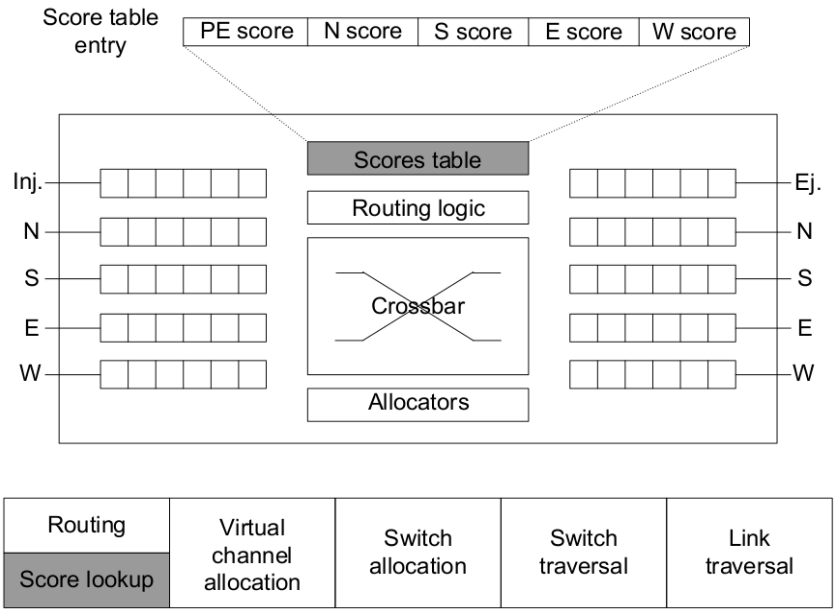


Figure 9 – Router architecture proposed by Eisley et. al [EIS08] to guide evicted blocks to unused cache blocks.

2.3.1 Static and Dynamic NUCA

NUCA architectures are sub-divided in two categories: S-NUCA and D-NUCA. In S-NUCA architectures, blocks are mapped to a given cache bank based on its address. The blocks are not moved between cache banks. In D-NUCA cache architectures, a block may be migrated to other banks towards processors that mostly access it, reducing dynamic energy consumption and latency. Despite the benefits of migration, one of the main challenges of D-NUCA cache architectures is to implement an energy-efficient and low latency mechanism to search for a block. It is necessary because as the blocks can migrate from one bank to another, it is not possible to determine the location of a block based only on its address, such as in an S-NUCA.

[LIR11] proposes a search algorithm named HK-NUCA for D-NUCA cache architectures. The algorithm has three stages to find a block in the L2 NUCA cache and is activated after a miss occurs in the first level cache. In the first stage, the cache controller of the requesting core accesses the closest NUCA bank; in the second stage, the home bank is accessed. The home bank corresponds to the bank determined by the bank bits of the address being accessed; and in the third stage, parallel accesses are sent to all banks that possibly hold the block. In case none of the stages finds the block, a miss request is sent to the upper-level memory.

2.3.2 Data Migration Methods Comparison

Table 3 shows a summary of the reviewed works on data migration. Most works adopt gradual migration based on the directional migration algorithm that migrates a cache block towards the PEs that mostly accesses it gradually. None of the reviewed works

adopts as the target platform NoC-based MPSoC. Additionally, most of the solutions only optimize the latency accesses, but do not consider the energy consumption. As one of the main goals of MPSoC designs is to consume low energy, our work proposes a data migration protocol whose main goal is to reduce energy consumption after performing a data migration.

Table 3 - Summary of the reviewed for data migration protocols.

Data Migration Protocol				
Authors	**Migration Algorithm**	**Goal**	**Platform**	**Communication Infrastructure**
[BEC04]	Direction Migration Algorithm	Optimize performance	CMP	2D Mesh NoC
[NAF10]	Direction Migration Algorithm + Prediction based Migration	Reduce cache access latency	CMP	2D Mesh NoC between routers / Bus between PEs
[EIS08]	Directional Migration guided by usage information	Reduce off-chip memory accesses	CMP	2D Mesh NoC
[KAN05]	Gradual Migration Algorithm	Decrease the number of necessary migrations	CMP	2D Mesh NoC
This Work	Region based migration	Decrease energy consumption	MPSoC	2D Mesh NoC

2.4 General evaluation of the state of the art

Most papers consider the modeling of the platform in cycle-accurate level or higher-levels of abstraction. Although these models require less effort and allow faster simulation, the RTL level still must be considered. The RTL level is necessary because some problems, concerning either the memory architecture or the traffic generated on the NoC are only observable at this level or at lower abstraction levels. Common examples of these problems are: congestion, burstiness and jitter.

In this work, we propose a distributed shared memory architecture, with support to data migration and cache coherence, implemented in an RTL NoC-based MPSoC.

3 HEMPS-Q PLATFORM

The HeMPS-Q Platform, developed in the scope of Carara's Thesis [CAR11] (Figure 10), is a homogeneous MPSoC in which PEs are interconnected through a Network-on-Chip (NoC) named Hermes QoS. Each PE contains a RISC microprocessor, a local (scratch pad) memory, a DMA controller and a Network Interface (NI). These modules are wrapped by the Plasma-IP module, which is then connected to the NoC. The next sections detail each of these modules. Section 3.6 presents the microkernel responsible for managing task allocation/execution and communication on the system. Concluding this Chapter, Section 3.8 presents the HeMPS Generator framework, which allows automatic generation of a parameterized architecture at RTL level. The user can manually configure platform parameters, such as NoC size, memory size and number of memory pages.

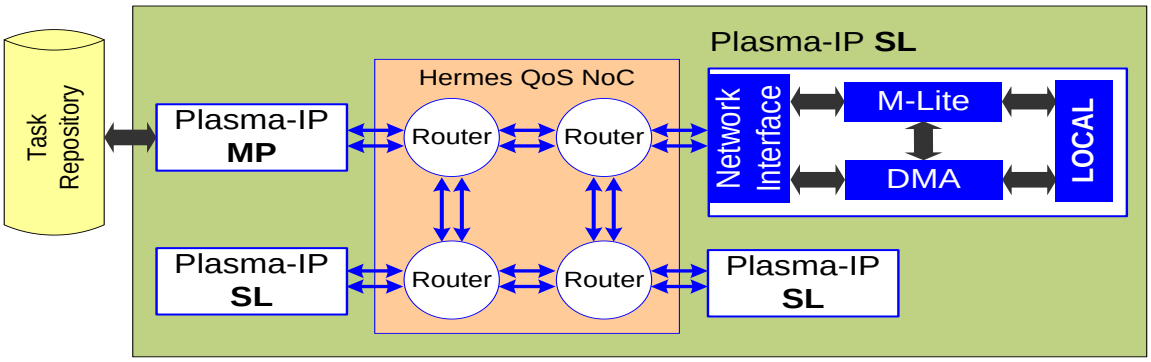


Figure 10 – HeMPS-Q platform architecture.

There are two types of Plasma-IPs: slaves (SL) and master (MP). Plasma-IP SLs are responsible for executing application tasks. Plasma-IP MP is responsible for managing task mapping and system debug. The external memory, named *task repository*, contain all application tasks. According to some mapping heuristic, the Plasma-IP MP maps the tasks into the Plasma-IP SLs. The Plasma-IP MP can also receive debug messages from Plasma-IP SLs, transmitting them to an external host through an Ethernet interface (not shown in Figure 10).

3.1 Hermes QoS

The Hermes QoS enhances the original Hermes NoC proposed in [MOR04] providing support to *Quality of Service* (QoS). Hermes QoS adopts a bi-dimensional (2D) mesh topology, Hamiltonian routing algorithm, replicated physical channels and a priority scheme to transmit packets. Two features of the Hermes QoS are important to support the implementation of the cache coherence protocol and data migration protocol: Hamiltonian routing and priorities.

3.1.1 Hamiltonian routing

According to [HAR72], 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, as shown in Figure 11. A Hamiltonian path is a path through which it is possible to traverse all nodes of a graph passing only once for each node. After defining the Hamiltonian paths, the network can be divided into two disjoint and acyclic sub-networks, as defined by the dual-path algorithm. One sub-network contains ascending paths that go from the smallest label router to the highest label router and the other contains descending paths 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. Figure 11 shows a network with 16 routers, labeled from 0 to 15, having two disjoint networks. Ascending packets are sent using the network depicted by the lines, whilst descending packets are sent using the network depicted by the dotted lines.

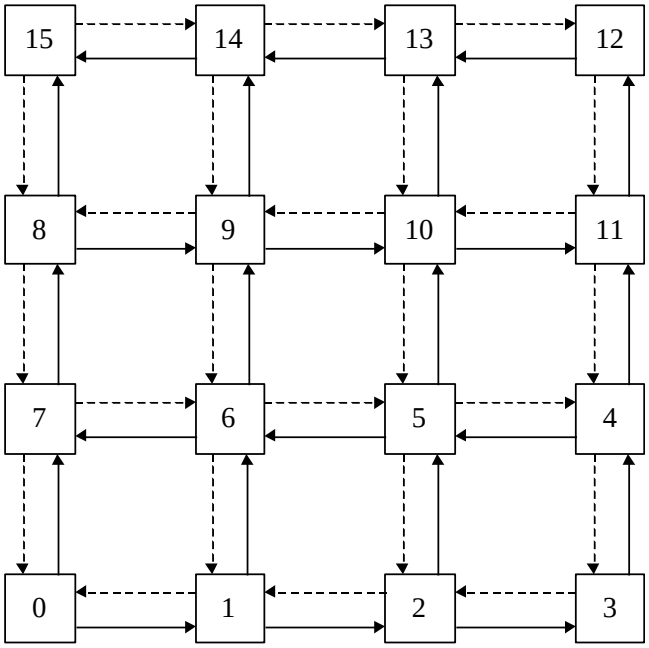


Figure 11 - Example of label assignment based on a Hamiltonian path in a 4x4 mesh.

Hamiltonian paths provide a good support for implementing efficient multicast/broadcast algorithms [CAR08]. Multicast messages may be used in the implementation of cache coherence protocols to optimize invalidation messages that target several PEs 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 mean plays an important role on the design and implementation of a cache coherence protocol.

The Hermes QoS supports multicast for both circuit-oriented connection and packet switching, based on the dual path algorithm. Multicast messages carry in the header the network address of all routers to which the message must be delivered. When a multicast

message is sent from a lower labeled router to a higher labeled router, the target network addresses are sorted in decreasing order, as shown in Figure 12(b). On the other hand, when a multicast message is sent from a higher labeled router to a lower labeled router, network addresses are sorted in increasing order in the header, as shown in Figure 12(a). When the message reaches each target router, the address of the current address (first flit of the header) is removed from the header and a copy of the message is forwarded to the next network address present in the header, and to the local port of the router.

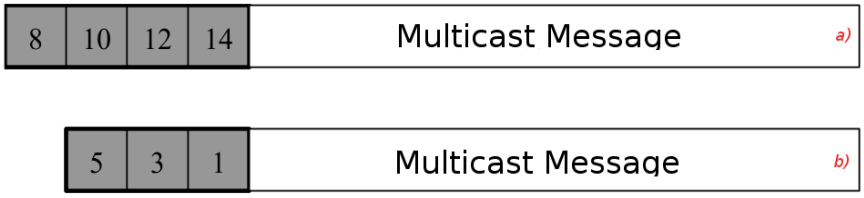


Figure 12 – Two examples of multicast messages. The first shows an example of a multicast message to network addresses 8, 10, 12 and 14. The second shows a multicast message that target network addresses 5, 3 and 1.

3.1.2 Physical channels and priority

The NoC Hermes QoS uses two 16-bit channels (physical links) to interconnect the routers. The advantage of this approach is that it allows allocation of one physical path for sending only high priority packets. In the NoC Hermes, channel 0 is used to transmit both high and low priority packets, whilst channel 1 only transmits high priority packets.

3.2 Plasma-IP

Plasma-IP is a modified version of an open-source soft core, described in VHDL, freely available at OpenCores [OPE11]. It consists of a 32-bit RISC architecture, strongly based on the MIPS-I ISA (*Instruction Set Architecture*).

The Plasma-IP core is implemented with a configurable pipeline that can be set to two or three stages, with an additional optional stage for memory accesses. The entity that controls the core converts a 32-bit instruction opcode to a 60-bit opcode and sends control signals to the other entities.

The most significant modifications performed on the original Plasma architecture were: insertion of a paging mechanism; exclusion of the UART module; addition of a DMA module and a NI (Network Interface). Concerning the instruction set, which is based on MIPS-I ISA, Plasma originally did not include the *syscall* instruction. *Syscall* allows the implementation of software traps. Traps are an entry point that allows 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.

3.3 Network Interface (NI)

The Network Interface main function is to serve as a wrapper of the Plasma processor to the NoC. It is responsible for splitting 32-bit words that are written in the memory by the processor to 16-bit flits that are transmitted in the NoC. 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 the words at the receiving end.

The NI is constituted by two finite-state machines (FSM): Receive and Send. The Receive is a 4-state machine responsible for buffering incoming packets from the network and signaling to PLASMA that a new packet may be handled. Furthermore, the Receive machine is responsible for controlling the reading of data from Plasma core. The Send 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. The NI controls DMA operations, and copy to/from memory.

3.4 DMA

The implementation of the Plasma processor available at *OpenCores* did not include a DMA module in its architecture. Without a DMA module, the processor must handle incoming packets buffered at NI. This operation is very expensive for the processor as it cannot execute any application task during this time, which causes a reduction on the system performance.

To address this problem, a DMA module was designed to execute two operations: transfer packets stored on memory to the NI and transfer incoming packets from NI to memory. DMA programming is done inside the microkernel (Section 3.6), whenever a new interrupt event originated from the NI module is detected or, every time a packet must be sent to the NoC. The configuration of the DMA module is done through a set of memory-mapped registers:

- DMA_OP: defines the operation to be performed by the DMA module, and can be either read or write.
- DMA_ADDRESS: defines the initial address for a given operation.
- DMA_SIZE: defines the size of the data to be transferred.
- DMA_START: initiates the operation assigned to the DMA.

To check the availability of the DMA module, a memory-mapped register, named DMA_ACTIVE, is used.

3.5 Memory system

Each Plasma-IP follows the Von-Neumann organization, having only one private memory, which stores both data and instructions. To allow simultaneous access from DMA and processor, the private memory is implemented as a dual-port memory. The memory space is divided into pages. The first pages store the microkernel, and the remaining ones store application tasks.

Application tasks are fetched from an external, off-chip memory, during system initialization (static mapping) or at run-time (dynamic mapping). The master PE accesses the external memory through a repository interface, in order to fetch object code of all tasks that are going to be executed in the system. This interface is also present on slave Plasma-IPs, and can be used to connect the core to another type of memory, such as a cache memory, with the advantage of not having to modify the core architecture.

3.6 Microkernel

The *microkernel* that runs on PLASMA core is a tiny operating system responsible for managing and supporting task execution on each core. The HeMPS microkernel is a preemptive OS, where each task occupies the CPU for a pre-defined period of time called *timeslice*. The microkernel has two versions: one runs on PLASMA master whose main objective is to coordinate task distribution and management. The master does not run any application task. The other runs on PLASMA slaves and gives support to features such as multitasking and software interrupts (*traps*). The memory of both master and slave is paginated. The kernel occupies the first pages, whilst the tasks occupy the remaining ones. The microkernel is divided into layers as shown in Figure 13.

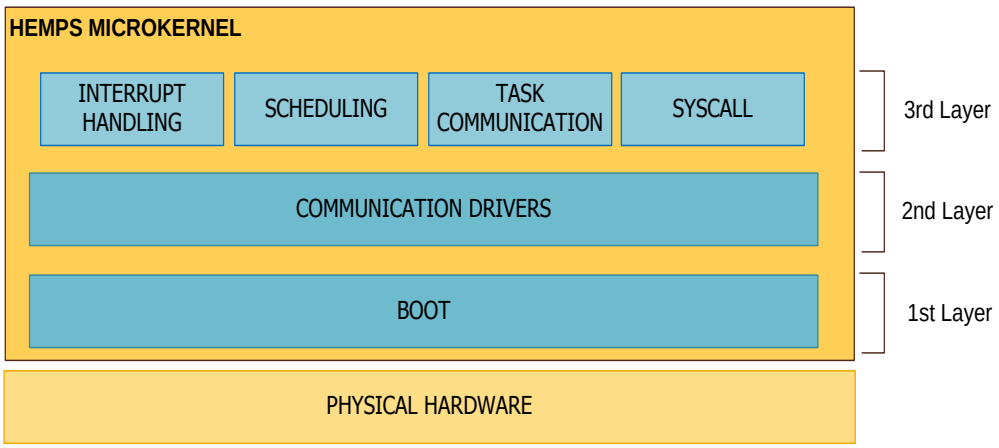


Figure 13 - HeMPS Microkernel Layers.

Upon finishing the execution of the boot code, the main function of the microkernel is called through the *main* global label that represents the main function of the C code. The microkernel initializes internal structures that are used later to maintain tasks status/context and configures all interrupts.

3.6.1 Boot (1st Layer)

The boot layer contains a few routines written in assembly and in C. The assembly code is responsible for initializing segments of global and static (.bss) data, global and stack pointers.

The Plasma processor starts its execution by fetching an instruction from the first address in memory (**0x**00000000). The first positions of memory contain the assembly code.

3.6.2 Communication drivers (2nd Layer)

The communication drivers are accessed only inside kernel address space, preventing user tasks from dealing directly with hardware modules. Although this limitation exists, a user task can have access to communication drivers through high-level macros, defined in the microkernel, which cause a trap and call to the communication driver routines. The communication routines are *Send()* and *Receive()*. Both routines write/read messages from the pipe (communication vector, described in Section 3.6.7). The implementation of the communication model in the HeMPS Platform is based on the read request protocol, while the computation model is based on the KPNs (Kan Process Networks). The *Receive()* blocks task executes until the moment that the message to be received is available on the pipe, the *Send()* routines is non-blocking.

3.6.3 Task execution (3rd layer)

To manage tasks execution, the microkernel keeps a TCB (*Task Control Block*) structure for each task. This structure contains the entire context of each task, so that it must be updated before switching execution to another task (*context saving*). Figure 14 shows the TCB structure that is currently implemented inside the microkernel in C. The registers saved on TCB structure are: temporary registers ($t0 - $t9); saved registers ($s0 - $s8); argument registers ($a0 - $a3); function result registers ($v0 - $v1); return address ($ra); stack pointer ($sp); global pointer ($gp); and registers used to store multiplication/division results ($hi and $lo). Beyond these registers, for each task is maintained its *id* (identifier), *pc* (program counter), *offset* (indicating its initial memory position) and *status*. The status of a task can be:

- READY : means that the task is ready to be executed;
- RUNNING : indicates that the task is currently executing in the CPU;
- TERMINATED : indicates that the task finished its execution;
- WAITING : indicates that the task requests a message and is waiting for response;
- FREE : indicates that the TCB is free and can be allocated;
- ALLOCATING: indicated that the TCB is being allocated.

```
typedef struct {
    unsigned int reg[30];
    unsigned int pc;
    unsigned int offset;
    unsigned int id;
    unsigned int status;
} TCB;
```

Figure 14 - TCB structure described in C language.

3.6.4 Multitasking

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 maintains a memory-mapped tick counter, implemented as an internal memory-mapped register. This register is incremented each clock cycle, and causes an interrupt event when it reaches a *threshold*. The time interval that starts at zero and the moment when the counter reaches the *threshold* is denominated *timeslice*. The interrupt event is handled by the interrupt handler, which in turn, calls the scheduling algorithm that allocates another task to be executed in the CPU.

The microkernel implements a round-robin scheduling algorithm, which assigns time slices for each task in equal portions and in circular order, handling all tasks without priority. Round-robin scheduling is both simple to implement and starvation-free.

3.6.5 Task allocation

The master PE, according to the task mapping, executes task allocation on HeMPS Platform. HeMPS Platform supports two types of task mappings:

- **Static**: 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**: When a task is dynamically allocated, it is sent to the Plasma-IP that will execute it only when another task in the system requests the allocation. This situation happens whenever a task T_i, running on processor P_i, needs to send a message to task T_j. Task T_i searches for task T_j location on its internal table, but it finds out that task T_j has not been allocated yet. Consequently, it sends a message to the master requesting the allocation to be done.

3.6.6 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 *threshold*, indicating a new task must be allocated; and a call to the *syscall* primitive (software interrupt – *trap*).

When a hardware interrupt event happens, the PC register receives the 0x3C address causing the execution of a full context saving function. When a software interrupt event happens, the PC register receives the 0x44 address causing the execution of a partial context saving function.

3.6.7 Task communication

Different tasks running on a distributed system need to communicate to coordinate and synchronize task execution. Task communication inside the processor is done through an 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 model adopted by HeMPS microkernel, a channel read operation must be blocking. Thus, when a *Receive*() instruction is executed, the task is blocked until it receives the requested message from the pipe. On the opposite, write operations must be non-blocking, which means that after every *Send*() operation, the task keeps executing.

When a task wants to receive a message, a read operation on the pipe occurs. Whenever a task wants to send a message it does a write operation (*WritePipe*()). If both sending and receiving tasks 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 15, 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, Processor 1 is interrupted by the NI when the request arrives, and sends a message (*msg,* Figure 15(b)) to processor 2 which receives the message and unblocks task 2.

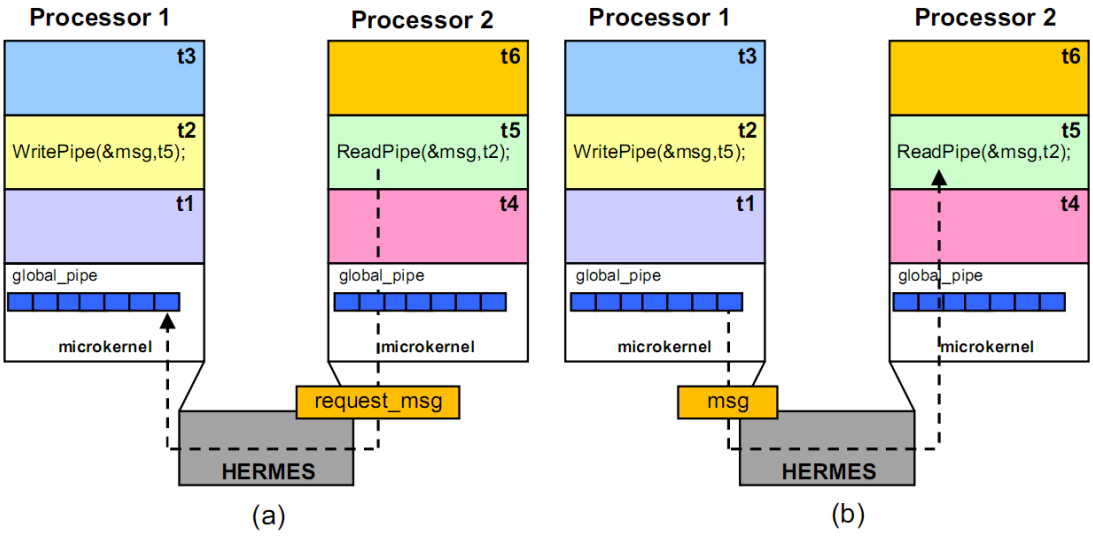


Figure 15 - Intertask communication between tasks located at different processors.

3.7 Memory management

As already mentioned in section 3.5, each Plasma-IP contains only a dual-port private 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 a given page to access 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.

3.8 HeMPS Generator

The HeMPS Generator is a framework that allows platform customization, where the user can define the number of processors connected in a mesh NoC through the parameters X and Y. The maximum number of tasks per slave can be configured, and is a function of two parameters: page size and memory size. For performance evaluation purposes, processors and local memories are modeled using cycle accurate instruction set simulators (ISSs) and C/SystemC models, respectively. This enables faster design space exploration. The left panel in Figure 16 presents two applications (mpeg, and communication) along with their task composition. Drag and drop actions allow to perform the initial static task mapping to slave processors. The master processor receives the remaining tasks (the applications that are still on the left panel when the platform is generated), which correspond to the contents of the task repository.

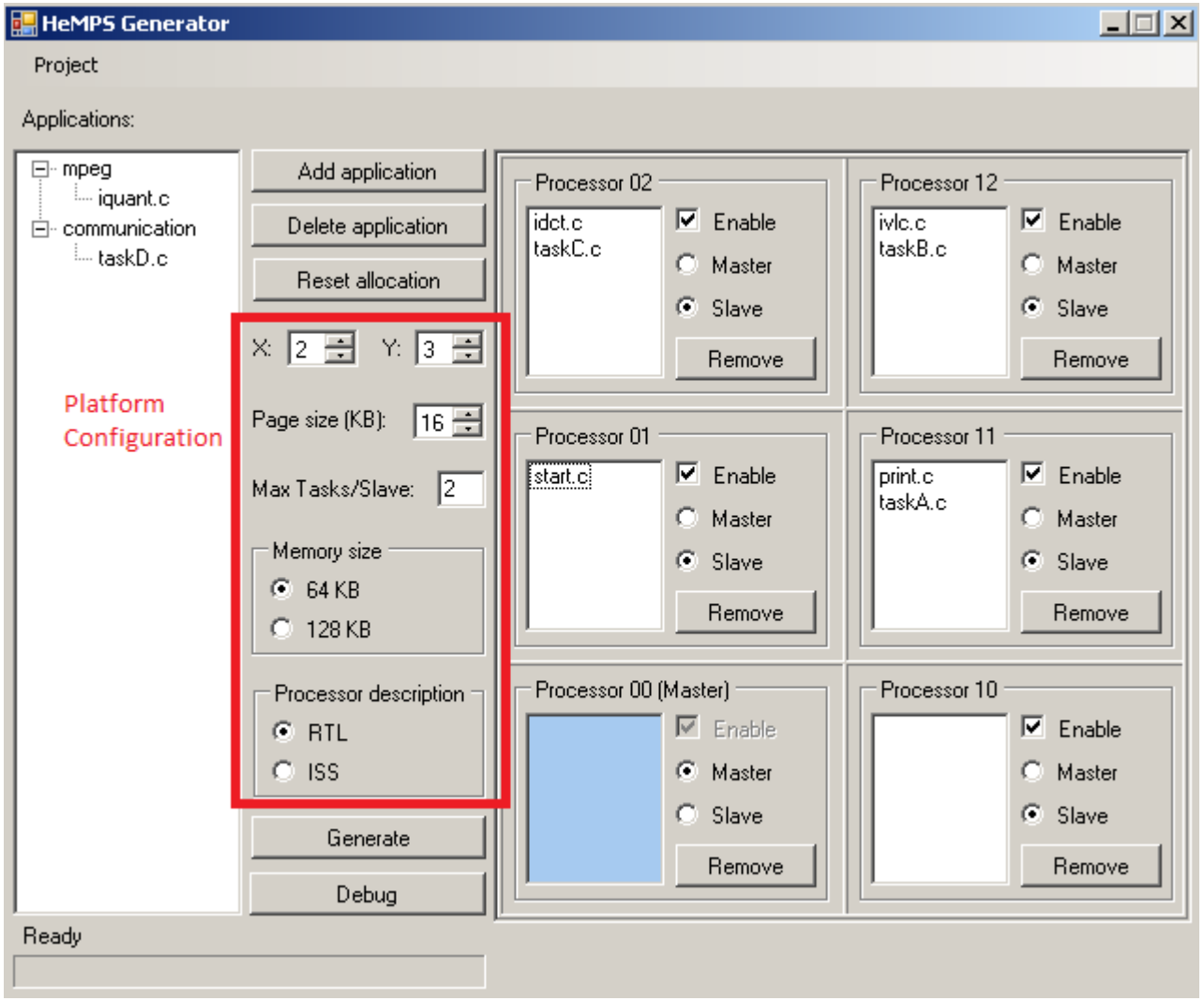


Figure 16 - HeMPS Generator Framework.

4 MEMORY ORGANIZATION FOR NOC-BASED MPSOCS

The increasing complexity of software applications executed in embedded devices requires the study and development of energy efficient and scalable mechanisms for processing, communication and memory. According to [KAN05], the memory subsystem is the bottleneck of an MPSoC in terms of energy consumption and performance. Solutions to memory organizations for NoC-based MPSoCs must address both aspects. Additionally, they must cope with the expanding number of on-chip cores and more complex software applications.

The HeMPS Platform, described in Chapter 3, does not include a memory hierarchy. Communication among application tasks, as well as data exchange only occurs through message passing. Silva et al. [SIL09] show that applications requiring a great number of data exchange might suffer from the increased latency present in NoCs. These types of applications may benefit from having a shared cache memory that stores global data. The shared-memory model facilitates interchange of data between processors and according to [TOT10] facilitates the programming model. Furthermore, the shared memory model may improve performance when executing applications that operate on large objects such as images.

Despite the benefits aforementioned, careful evaluation must be done to determine the impact of additional memory structures. The evaluation must consider mainly energy consumption and area, which means that the memory organization, as well the cache-coherence protocol (Chapter 5) must be designed considering both constraints.

The current chapter presents the first contribution of this dissertation. It presents the development of a memory hierarchy for the HeMPS Platform. The hierarchy is constituted by L2 shared data cache banks, which are connected to the NoC, and two memories per processor: a local memory that stores both data and instructions private to the PE and a L1 cache for global data. Some of the control mechanisms of the memory hierarchy are executed in software by the microkernel that runs on slave PEs. One of the mechanisms implemented in software is the handling of cache *misses*. This approach was adopted to simplify the hardware modules.

This Chapter is structured in three subsections: the first presents the hardware modules that compose the memory organization developed for the HeMPS Platform; the second presents a NUCA memory organization developed for the HeMPS platform and, the last subsection presents the software structures developed in the microkernel to support the use of the cache memories by application tasks.

4.1 Hardware support

The reference HeMPS MPSoC (Figure 10) has been modified to support a two-level memory hierarchy, as illustrated in Figure 17. The first memory level contains processor's memories, whose address space is divided into two separated segments: one for the L1 data cache and the other for the local memory. The local memory can be seen as a *scratchpad memory* [KAN05] because it guarantees one-cycle latency for every access. Additionally, all accesses to this memory end in this level, which means that accesses never result in a *miss*. The local memory stores the microkernel, application tasks and application tasks private variables. Accesses to the L1 data cache might result in a *miss*. In this case, a block request is sent to L2 cache, which is the upper level in the hierarchy. The L1 private data cache stores copies of blocks of the L2 shared data cache. The L2 data cache stores data shared by applications.

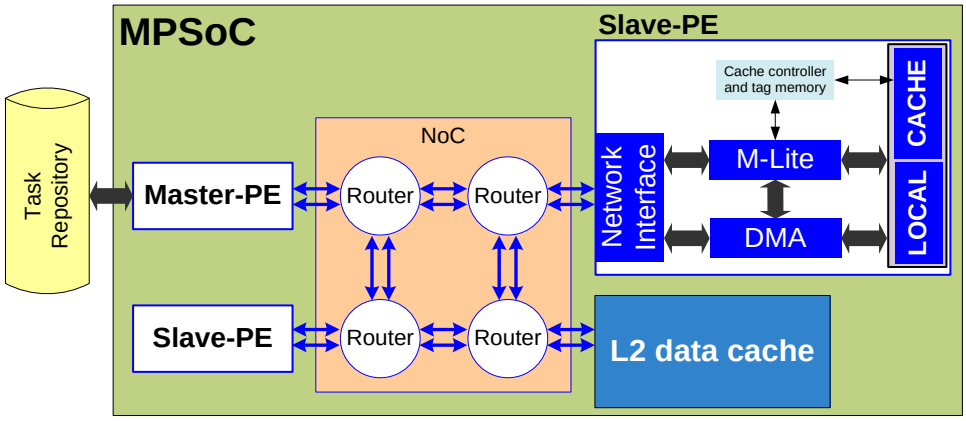


Figure 17 - MPSoC with a two level memory hierarchy – shared memory and caches (only Slave-PEs may contain caches).

The M-Lite processor may receive data from several sources. The source that provides the data depends on the address being accessed. Figure 18 shows the modules of the Plasma-IP and some interconnections between modules. The *mem_data_r* signal of the M-Lite processor is a 32-bit input signal that receives data from four sources: (*i*) local memory; (*ii*) private L1 data cache (only slave-PEs); (*iii*) task repository (only master-PE); (*iv*) memory-mapped registers. Several memory-mapped registers are used in the M-Lite processor to help the development of control routines in software (microkernel). For instance, the DMA_STATUS is a memory-mapped register, which is mapped to a given address of the memory. In this case, such mapping allows the microkernel to verify DMA status (busy, idle) before programming it to execute a given operation.

4.1.1 L1 data cache architecture

The L1 data cache adopts the direct mapping scheme, due to the minimum hardware support required to implement it. A *tag memory* is used to store control bits and tags for each block of the cache. The tag memory stores for each block: (*i*) tag, the most significant bits of the L2 cache block address stored; (*ii*) valid bit indicates if the block is valid and

shared; (*iii*) modified bit, the PE modified its exclusive copy of the block. The cache block size contains 128 32-bit words (the same size of the packet that can be stored in the communication pipe).

Cache management is done both in hardware and software. A module named *cache controller*, implemented in hardware, controls read and write operations as well as hit and miss signalizing. The control in software is performed by the microkernel which handles *misses* and updates the tag memory.

When a write access is generated to the L1 data cache, the cache controller receives as input the following signals from the M-Lite processor (Figure 18): *address,* which informs the cache where to write; *data_write* the data to write in the cache; and write_*byte_enable,* which is a four-bit signal that indicates which bytes of the *data_write* signal must be written to the cache. The cache is only activated if the *address* generated by the processor is above the address *0x*10000000. For a read access, only the *address* signal is necessary.

For each new access, the cache controller signalizes the occurrence of a *hit* or a *miss* by setting a memory-mapped register. After performing an operation on the data cache, the microkernel reads the memory-mapped register, named CACHE_HIT, to decide which operation to take next. If the first bit of CACHE_HIT is asserted, it means that the accessed address is valid in the cache. If not, a *miss* occurred and the microkernel must assemble a read request packet and send it to the L2 cache controller.

When the L2 cache controller sends the block back to the processor, the NI detects the packet arrival and interrupts the microkernel. Subsequently, the processor programs the DMA to copy the block to the cache and updates the tag memory entry of the block read.

4.1.2 L2 data cache architecture

The second and lowest level of the memory hierarchy is the L2 shared data cache (Figure 19). The L2 cache is connect to the NoC through a Network Interface as an IP. It provides support for handling memory operations, cache coherence and data migration. Cache coherence and data migration topics are presented in the next chapters of this dissertation (Chapter 5 and 6, respectively).

The L2 cache is accessible to all PEs and is composed of: (i) a Network Interface (NI) connecting the L2 cache bank to the NoC; (ii) a cache controller, responsible for interpreting incoming packets buffered at NI and executing read/write operations; (iii) a directory memory, which stores information to support the implementation of a cache-coherence protocol (The directory memory sub-module is described in Chapter 5); (iv) a memory bank, logically divided into blocks of 128 32-bit words. Note that more than one L2 bank can be instantiated in the MPSoC, resulting in a NUCA organization.

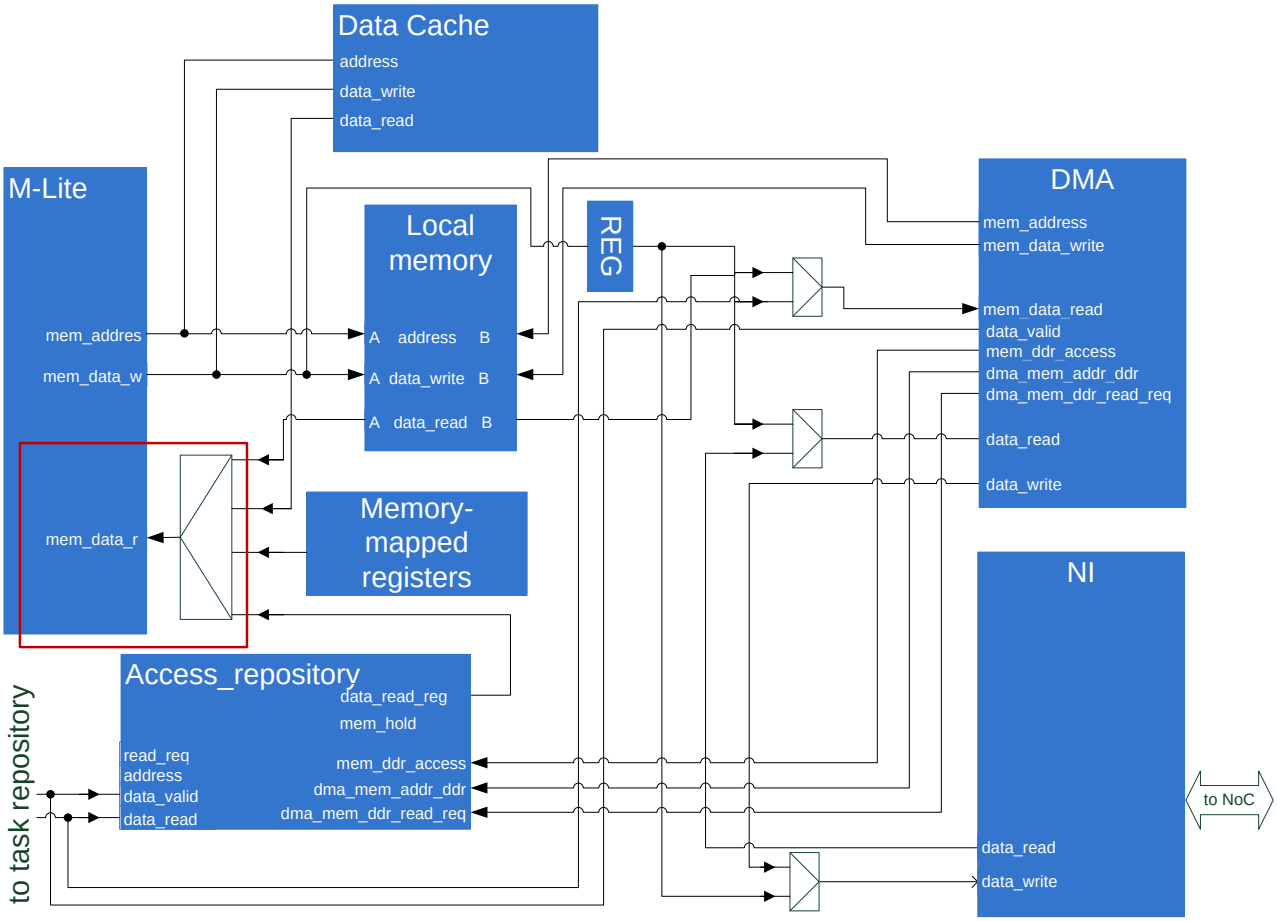


Figure 18 - Internal modules of the PLASMA processor.

All necessary instructions to execute an application 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 the L2 cache to PEs. The advantage of this memory hierarchy is the smaller traffic inside the NoC. The next subsections describe each sub-module of the L2 cache.

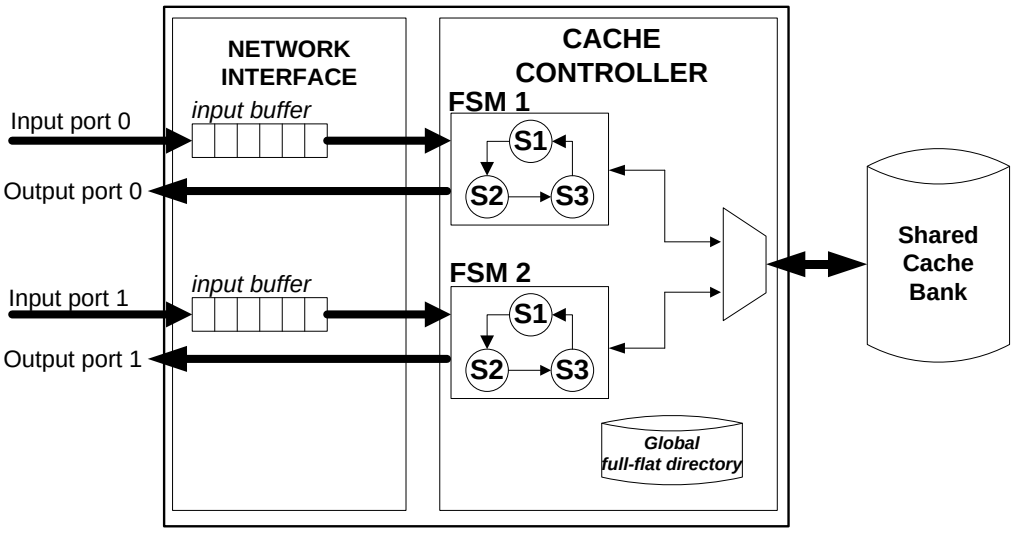


Figure 19 – Internal architecture of the L2 cache module.

4.1.2.1 Network Interface

The Network Interface sub-module is responsible for bridging the NoC router with the L2 cache bank. The NI is responsible for buffering incoming packets and also, for sending packets generated by the cache controller. The NI interfaces both channels of NoC and has two sub-modules, send and receive.

The *receive* sub-module has two buffered ports (port 0 and port 1 in Figure 19), with a buffer at each input port. Packets containing *write* operations have higher priority in the NoC because they are longer than packets containing other operations (e.g. read request). Packets containing operations such as *read* and coherence specific actions are sent using low priority. The output ports of the NI are not buffered, since the cache controller may send the data in burst.

Two signals (*first* and *last*) are used to control the write and read operations of data on the input buffer, following a circular buffer approach. Each position of the buffer stores a *word* (32 bits) plus an additional control bit, which indicates if that position of the buffer stores the last word of a packet. This allows the NI to separate flits from one packet to another. Every time a new word is written to the buffer, the *last* signal is incremented. On the other hand, every time a word is read from the buffer, the *first* signal is decremented. When both signals have the same value (point to the same position of the buffer), the buffer is considered empty. When the *last* signal is one position behind the *first* signal, the buffer is considered full.

If the *first* and *last* signals are different, it means that there is available data to be read in the buffer. The NI is responsible for signalizing to the cache controller module that there is available data to be read. In situations where the buffer of the NI is full, and the cache controller is busy, the *credit_i* signal is set to low. This signal indicates to the NoC that the L2 cache bank is fully occupied and cannot receive any additional packet. This signal is set back to high after the cache controller consumes some words of the buffer.

The *send* sub-module is controlled by the cache controller, which sets a *send_data* signal to high and puts a word in the bus. The send module needs to split a word into two flits before sending them to the target PE in the NoC. The split operation is necessary because the width of the NoC links is of 16 bits (flit size).

4.1.2.2 Cache controller

The cache controller is responsible for performing the following operations:

- read and write of packets from/into the cache banks;
- cache coherence control (Chapter 5);
- data migration (Chapter 6);
- forwarding packets to another L2 cache (block search - Chapter 6).

The architecture of the cache controller is constituted by two finite-sate machines (FSM). One FSM is responsible for handling only write packets that are buffered at port 0 (high priority packets), whilst the other is responsible for handling all other types of packets.

Each write packet contains a 128-word cache block, having a total size of 262 flits. The format of write packets is shown in Figure 20. The first flit of the packet contains the network address of the shared memory (*TargetNetAddr*) (shown as flit 0 in Figure 20); the second flit (Size) contains the size of the packet; the third flit contains the service that must be executed by the cache controller; the fourth flit (*SourceNetAddr*) contains the network address of the PE that is writing the cache block to the L2 cache; the fifth flit (*TargetBlock*) contains the address of the cache block being updated; the sixth flit (*SourceTaskId*) contains the id of the task that issued the write operation; the remaining flits are the contents (words) of the cache block (Payload).

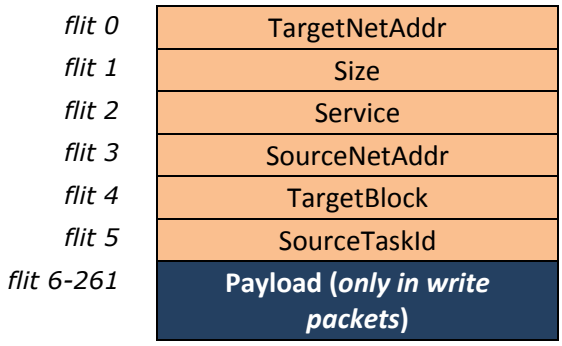


Figure 20 - Packet format for shared memory operations.

The other FSM (FSM 2 - Figure 19) handles incoming packets buffered at port 1 (low priority packets). These types of packets are short (6 flits). The format of short packets is the same as of the *write* packets, except that short packets do not contain the payload.

The priority mechanism included in the Hermes QoS NoC (Chapter 3) enables to differentiate the memory transactions. Such strategy avoids the contention of a L2 cache bank for long periods, such as in situations when a L2 cache is receiving a write packet originated at a cache miss. In this situation, even having one of the ports busy, the other one is free to receive and buffer service requests that will be handled as soon as the write operation finishes.

4.1.2.3 Directory memory

The directory memory consists of an auxiliary memory that stores the status of each block of the L2 shared cache, enabling the implementation of the directory-based cache coherence protocol (Chapter 5). For each block of the L2 shared cache, there is a correspondent entry in the directory memory that stores a set of bits as shown in Figure

21. Each field of the structure shown in Figure 21 represents 1 bit. The first bit indicates if the block is valid. The second bit, *Shared*, informs if the current block is in the *shared* state of the MSI (Modified-Shared-Invalid) protocol. The next bit, *transition,* is used in this work to optimize the protocol in certain scenarios, as explained later in section 5.4. The *index* bits store the network address of the PE when the block is in *exclusive* state (*shared* bit is low). This helps reducing time searching time when write-back messages need to be sent to the PE holding an exclusive copy of the block. The remaining bits (*P0, P1, …, PN*) inform which PEs have a non-modified cached copy of this block, when the block is in *shared* state. The number of *index* bits is equal to log_2 *nPEs*, here *nPEs* is the number of PEs in the system and the number of PN bits is equal to *nPEs*.

Figure 21 – Structure of an entry of the directory memory.

4.2 Non-Uniform Cache Access (NUCA)

The decoupled architecture of the L2 cache module allows the instantiation of several caches at a given MPSoC configuration, resulting in a NUCA architecture. In such architecture, the shared cache banks are physically distributed but logically centralized [HEN03]. In this architecture, it is necessary to determine which L2 cache bank will provide a cache block in case of *miss*.

The L2 cache bank that provides a cache block in case of a cache *miss* is chosen based on the physical address being accessed by the PE. The advantage of having more than one L2 cache bank is the increased memory throughput as there are more cache controllers to handle requests simultaneously [MAN10]. Also, it allows the implementation of mechanisms that allow moving cache blocks to L2 cache banks closer to the PEs that mostly access them, reducing energy consumption spent in communication. This is the main goal of a data migration algorithm described in Chapter 6. In a CSM architecture, which has only one L2 cache bank, the cache controller node may congest the region of the NoC where it is placed when several PEs issue requests to this cache.

In the HeMPS platform, applications obtain access to the cache through the microkernel. In the NUCA architecture, the microkernel is responsible for determining which L2 cache bank will provide a given block in case of *miss*. To implement NUCA support in the microkernel, we assumed a sequential address space, divined into *n* segments of equal size, as shown in Figure 22. Each segment is assigned to a different L2 cache bank. The network address of each bank is associated to a position of an array named *memory_banks*. The array is used by the microkernel to retrieve the network address of the cache bank that will receive the cache *miss*.

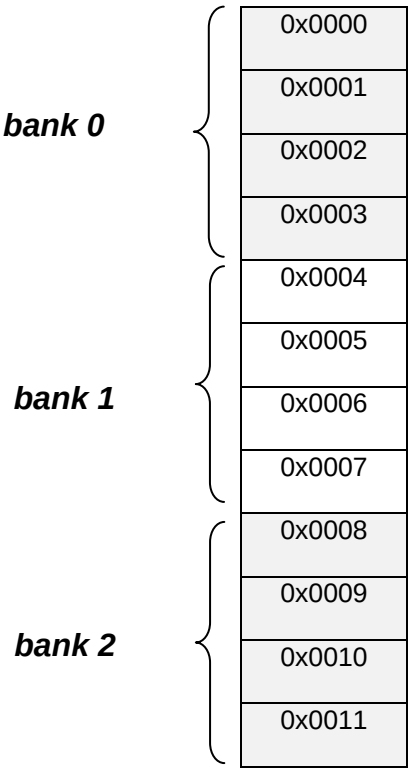


Figure 22 – Example of address space division in cache banks, assuming four blocks per L2 cache bank.

L2 cache banks have a configurable number of blocks. In a 32 blocks per bank configuration (*number_of_blocks_per_bank*), when an application task access a block whose address is in the range of 0 to 31, the microkernel sends a read request to the first cache bank in the *memory_banks* array. If the application task accesses a block in the range 32 to 63, and a cache *miss* arrives, a request will be sent to the second cache bank.

4.3 Software support for memory hierarchy

This section describes the control mechanisms of the memory hierarchy implemented in software. These mechanisms have two main goals: (*i*) minimize the amount of control structures required by the L1 and L2 cache controllers; (*ii*) provide a cache API which allows applications to manipulate the cache. The microkernel of the HeMPS platform is responsible for: (*i*) handling cache misses; (*ii*) controlling cache coherence on the PE side; and (*iii*) programming the DMA to perform the copy of an incoming cache block from the NoC to the L1 cache.

4.3.1 Cache API

The cache API provides an interface that allows applications to read and write data on the data cache memory. The API hides implementation details and complexity from the user, controlling cache coherence and handling *miss*es. The functions provided by the API act as an interface between applications tasks and the microkernel.

Currently, three functions are provided by the cache API to the applications tasks:

- ***read_block_word***(block_address, offset): Allows the task to read a 32-bit word from a given block of the cache. The function receives two parameters: *block_address*, which is the logical address of the block being accessed; *offset,* which is the address of the word being read in the cache block.
- ***write_block_word***(block_address, offset, value): Allows the user to write a 32-bit value to a word of a given cache block. The function receives three parameters. The first two parameters define the logical block address and word offset inside the block. The third parameter (*value*) corresponds to a 32-bit word that will be written in the specified location.
- ***flush_block***(block_address): This function allows the user to force an eviction of a cache block. It is useful for debug purposes and also to the implementation of the cache coherence protocol.

All functions manipulate integer values. For instance, the *read_block_word* function returns an integer as a result of the read operation. If the user needs to manipulate data from the cache as a different type, such as a char for instance, the 32-bit integer value needs to be casted to the desired type. It is out of the scope of this work to provide a full-fledged API that supports read and write operations having different value sizes and types.

4.3.2 Handling of cache operations inside the microkernel

In the HeMPS platform, the *syscall* instruction is used as an entry point for application tasks to request the execution of a privileged operation to the microkernel. A privileged operation accesses a data structure of the microkernel or communicates with a hardware element, such as the DMA, which cannot be programmed directly by an application, for instance.

When an application needs to execute a privileged operation, it requests the microkernel to execute a service, such as the *Send* service, which sends a message in the NoC. The request is done through a call to *syscall* that causes *trap*. A *trap* is a software interrupt that makes the processor start executing the microkernel syscall handler function. The *syscall* approach is treated by the cache API functions to request the microkernel to access the data cache.

Every time an application executes a call to the *read_block_word* function, for instance, a trap occurs and the microkernel is called to perform a read on a given position of the cache. The microkernel then executes the read operation as shown in Figure 23. The first operation executed by the microkernel is to place the physical address of the cache block being accessed in the address bus (step 1). In this step, the logical address is used. This address is converted inside the M-Lite processor to a physical address. The conversion is based on a logical to physical mapping table that is updated after each data migration (Chapter 6). If the mapping table does not contain a logical to physical mapping,

the logical address is assumed to be the physical address. The L1 cache controller uses this address to access *tag memory* and signalize *hit* or *miss* occurrence (step 2). In case the cache block being accessed is present in the cache, the cache controller also puts the requested word on the data bus of the processor in the next clock cycle. The microkernel verifies the CACHE_HIT register to verify *hit/miss* occurrence (step 3). If a *hit* occurred, the microkernel returns the word as a result of the *syscall* function to the application task (step 4). In case the *hit* bit is low, a packet must be sent to the L2 cache bank that has a valid copy of the cache block. Before doing this, the microkernel must verify if it needs to write back the current cache block (step 5). After this, the microkernel sends a write back packet to a L2 cache (step 6). Next, the microkernel assembles a read request packet and programs the NI to send the packet to the corresponding L2 cache (step 7) and sets the task to WAITING status (step 8). All the steps described so far are executed on the PE.

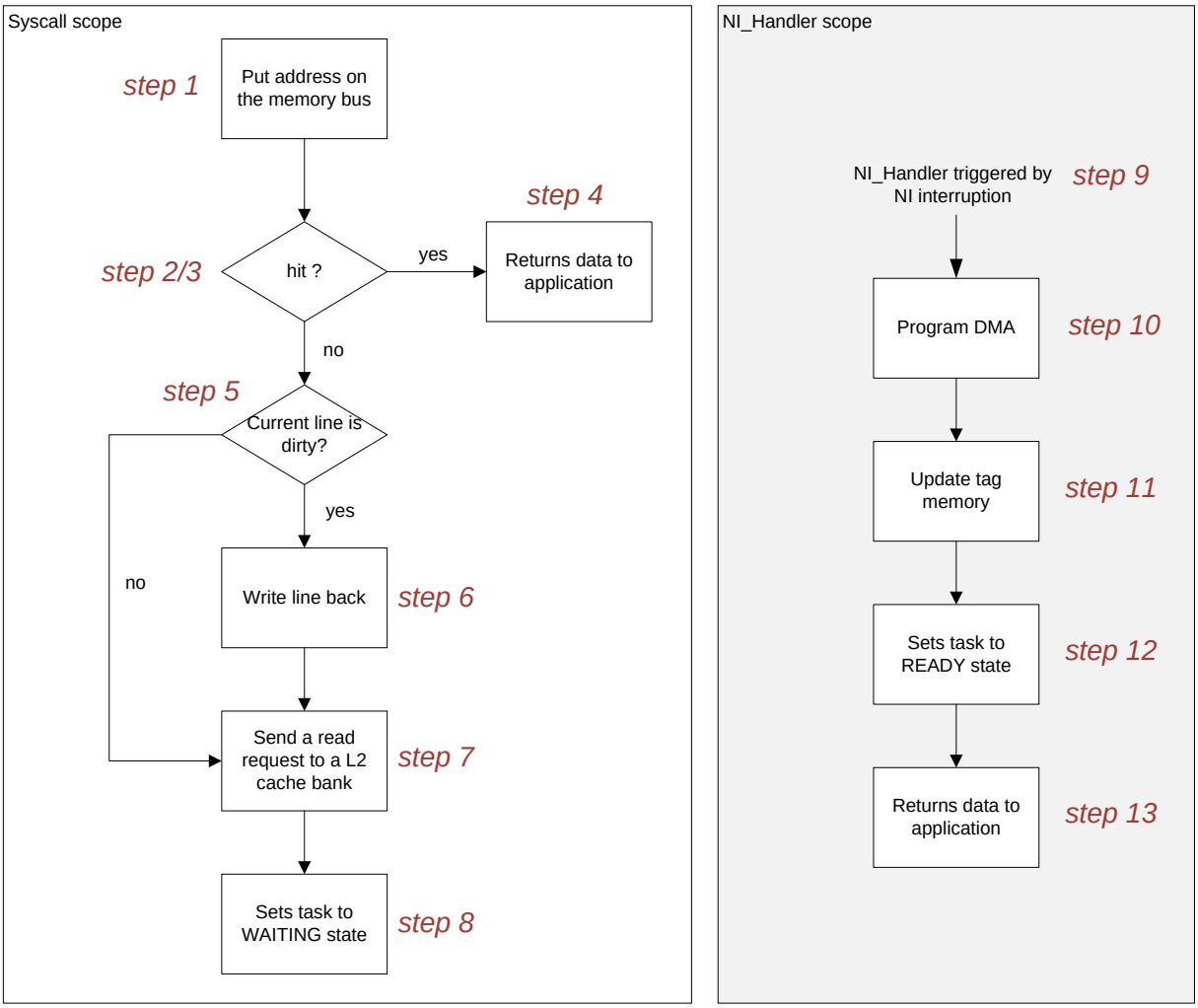


Figure 23 – Sequence diagram of all possible actions performed by the microkernel in a read operation.

Next, the L2 cache bank receives the packet and decodes it, and then reads the cache block from the cache bank. As the words of the block are being read from the cache memory bank, the L2 cache controller assembles and starts sending a message with the response block to the PE that issued the cache miss.

When the message arrives at the NI of the PE, a hardware interrupt occurs (step 9). Before treating the read response packet, the microkernel needs to save the context of the current task that is running (in the cases where there is only a task per processor, an idle task is scheduled and runs in the processor). The microkernel then programs the DMA to copy the block to the cache memory (step 10), updates the *tag* memory (step 11), sets the application to READY state so that it can be schedule again (step 12), and finally, returns the word back to the application (step 13).

For a write operation, there is an additional action necessary that is acquiring exclusivity to write in the cache block. In this operation, instead of issuing a read request such as in the read operation (step 7 - Figure 23), a read with exclusivity request is issued. The difference between the requests is that in the last the L2 cache bank responds with the data of the block being read (such as in the ordinary read request) plus the enforcement that the block is in exclusive state and can be modified.

4.4 Concluding remarks

An evaluation of the mechanisms and modules of the memory organization is presented later at Section 7.1. The experiments mainly evaluate the gains in latency when increasing the number of L2 cache banks (NUCA organization).

5 CACHE COHERENCE PROTOCOL

The presence of caches in microprocessor-based systems improves performance by reducing the processor's memory access time and by decreasing the bandwidth requirements of both the local memory module and in the NoC. Unfortunately, the local caching of data introduces the cache coherence problem. Directory-based protocols are suitable for NoC-based MPSoCs because these protocols do not require global visibility for all memory transactions.

The first description of directory-based protocols appears in Censier and Feautrier's paper [CEN78]. The directory is simply an auxiliary data structure that tracks the caching state of each cache block in the system. For each cache block in the system, the directory needs to track which caches, if any, have read-only copies of the block, or which cache has the latest copy of the block if the block is held exclusively. A directory-based cache-coherent machine works by consulting the directory on each cache miss and taking the appropriate action based on the type of request and the current state of the directory

The type and number of status that can be assigned to a cache block varies according to the protocol. In the MSI protocol, a shared block can be in 3 states: *modified* – which indicates that only one PE has a valid copy of the block, which means that the L2 cache does not contain a valid entry of that block; *shared* – one or more PEs contain a read-only 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 cache block is not consistent 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 bank that has the modified copy of the block. More details about the T state are presented in section 5.4.

This work adopts a hybrid implementation of the MSI protocol, being part of it implemented in hardware (in the L1/L2 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.

The L1 cache controller is responsible for: (*i*) detecting and signaling hit/miss when the address value changes in the cache; (*ii*) updating the tag memory; (*iii*) executing read and write operations. The microkernel is responsible for: (*i*) exchanging messages with the L2 cache bank; (*ii*) replacing blocks when necessary; (*iii*) handling write-back operations. The L2 cache controller is responsible for: (*i*) handling the directory memory; (*ii*) enforcing coherence according to the MSI protocol. To enforce coherence, it is necessary to: request *write-back* of a cache block if a read occurs on a *modified* block; invalidate L1 copies when

a PE requests permission (*exclusivity*) to modify a given cache block; forwarding read requests to the correct L1 cache controller when a cache block is in the T state.

Cache coherence protocols may adopt different approaches to maintain cache blocks consistent. For instance, some protocols update all copies of a cache block after it is modified locally in a given PE. Other protocols invalidate the local copies of all PEs but the one that did the update. Such protocols are usually classified as invalidation based protocols.

In invalidation based protocols, several unicast messages are issued whenever a cache block needs to be invalidated. The traffic generated by these messages increases significantly the energy consumption introduced by the cache and latency [JER08b]. One way to minimize this overhead is to send multicast messages to reduce the traffic induced by invalidation messages.

The main goal of the implemented protocol is to explore NoC services such as multicast and priorities, as well as some properties of the NoC, such as duplicated physical links. Exploring low-level features of the NoC may reduce the energy consumption and increase the performance of cache coherence protocols for NoC-based MPSoCs. The exploration of NoC services in the cache coherence protocol implementation is the second contribution of the current work. The next subsections present the optimizations proposed by this work to implement the MSI directory-based protocol.

5.1 Invalidating cache blocks

Whenever a PE needs to modify a given cache block, the L2 cache must invalidate all valid entries of the block to prevent cache incoherence. An invalidation message is then sent to every PE sharing this cache 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 24(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 24(b) shows a scenario where multicast is exploited. In this case, the L2 cache issues a multicast message targeting several PEs, reducing network traffic. The traffic reduction decreases the switching activity of the routers, therefore reducing energy consumption.

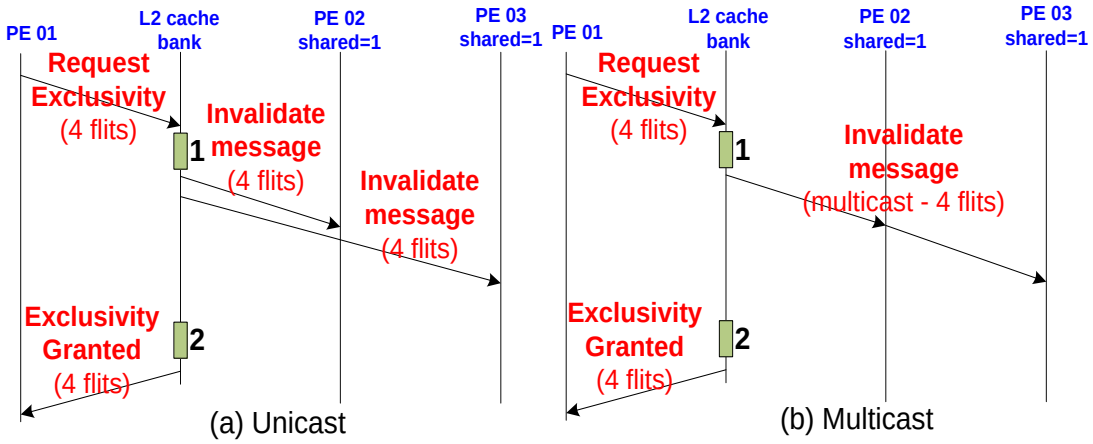


Figure 24 - Sequence diagram for a request of exclusivity on a shared cache block.

5.2 Read request optimization

The use of multicast messages might optimize a read operation on a cache block that is in modified state. The non-optimized operation occurs as shown in Figure 25(a). After receiving the modified block from PE02 (event 3), the L2 cache first writes the block into the cache 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.

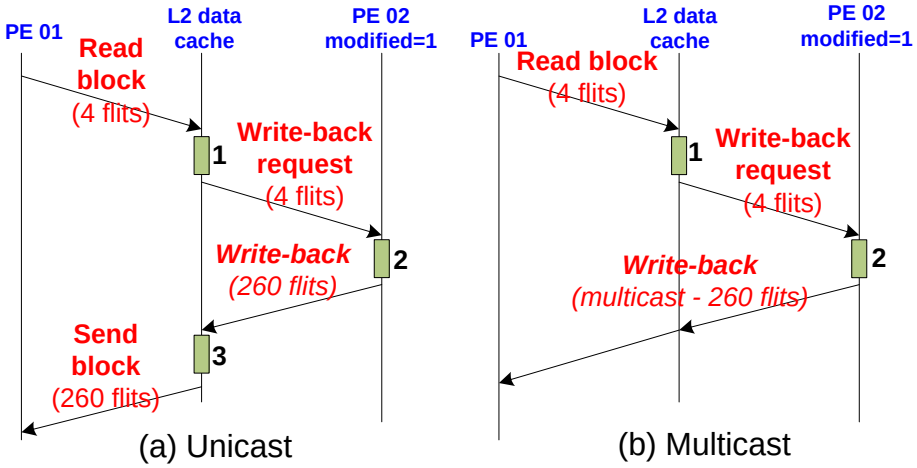


Figure 25 - Sequence diagram for a read operation of a modified cache block.

5.3 Write request optimization

To write on a cache block, the PE 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 cache block, only cached by PE02. PE01 sends a *read-with-exclusivity* request to the L2 cache, which, in the non-optimized implementation (Figure 26(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 26(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 cache 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.

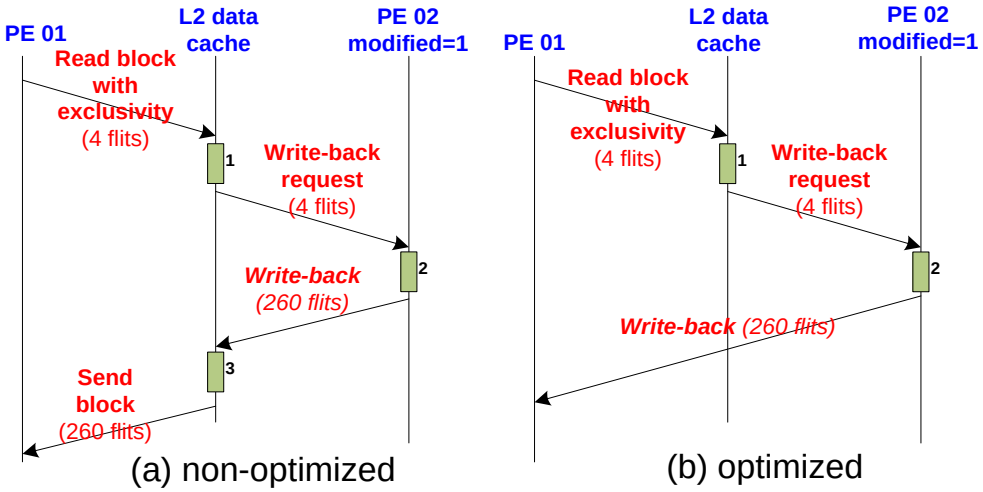


Figure 26 - Sequence diagram for a write-back operation after a write request.

5.4 The Transition State

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

Figure 27 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 cache block in modified state, and PE00 wants to read it. A read miss occurs, resulting in a read request to the L2 cache (event 1 in Figure 28). 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 it. 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 block to be on the L2 cache.

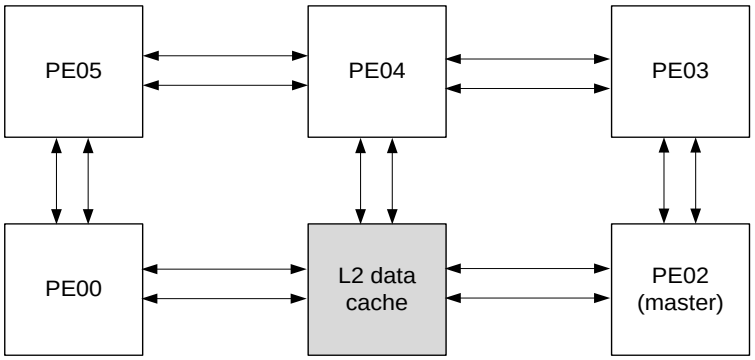


Figure 27 - MPSoC configuration: 3x2 NoC, 5 PEs (1 master, 4 slaves) and 1 L2 cache.

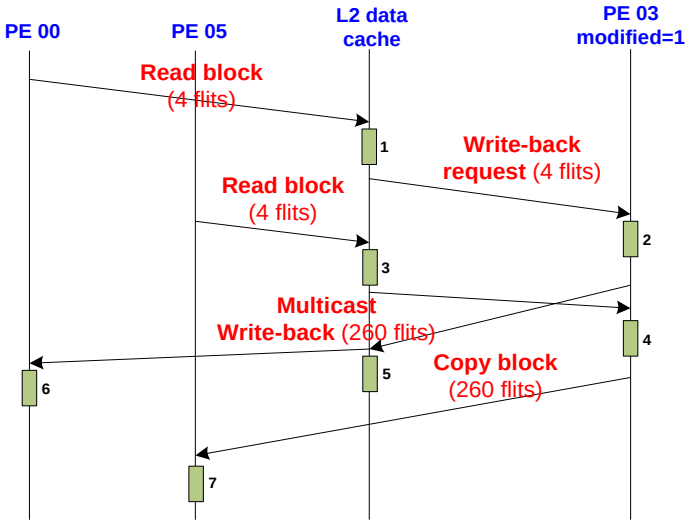


Figure 28 - Sequence diagram for the T state.

5.5 Concluding remarks

The proposed optimizations are evaluated in section 7.1.5. The results obtained from this study were published as a full-paper in two conferences, ReCoSoC [CHA11a] and SBCCI [CHA11b] . The paper published in the SBCCI conference has been chosen to be published in a special edition of the JICS journal.

6 DATA MIGRATION PROTOCOL

Task mapping is a challenging task in MPSoC designs [EVE10]. Given the amount of available resources in the MPSoC, it is not always possible to map application tasks near to all resources that will be accessed by them. Additionally, some mapping approaches do not consider the frequency of memory accesses and the distance of an application task to the memory bank that it mostly accesses. Thus, it is possible to have scenarios where tasks are mapped in PEs distant to the memory bank that they access. Moreover, dynamic mechanisms implemented in the MPSoC, such as task migration, or the occurrence of an unexpected problem, such as a faulty path on the NoC, may degrade performance significantly. Such problems are not predicted by task mapping algorithms.

In NUCA architectures, blocks of data can be migrated between cache banks to reduce memory accesses communication costs [BEC04]. Such migrations are triggered by a data migration mechanism. The implementation of such mechanism requires additional structures in the MPSoC. Some of the advantages of performing data migration are: reduction in the energy consumption of memory accesses; access latency reduction; and reduction of hot spots in a given region of the NoC, as the distance traversed by a memory access request and response are usually shortened after migration.

A data migration mechanism needs to monitor accesses done to memory blocks at runtime. The monitoring process takes into consideration: the block being accessed, which cache bank that the block is currently stored and the PE that is accessing it most frequently. Based on these parameters, the mechanism can evaluate and decide whether a migration will optimize the system. The proposal and development of the data migration protocol is the third contribution of the current work.

The approach adopted by this work to evaluate if a block needs to be migrated is defined by a function that calculates the cost of a single L2 cache access. In this work, we define as parameters for *f_cost* the communication energy and the access latency (measured in clock cycles). A block is only migrated if the new cost is smaller than the current one, considering the function *f_cost*.

In the scope of this work, the main goal of the data migration mechanism when migrating a block is not to decrease *f_cost* for a single PE, but for a *region* of PEs. A *region* is herein defined as a set of PEs and a L2 cache bank, named *home bank*. The number of regions in a given MPSoC configuration is equal to the number of L2 cache banks instantiated in the platform. The smallest *f_cost* is achieved when a given PE accesses a block that is located at the *home bank.* Data migration only occurs between L2 cache banks of different regions, based on the frequency of accesses done to a block by a *region*.

The advantage of adopting the *region* approach is the smaller amount of control required by the data migration mechanism, once accounting accesses in a given block considers regions and not individual PEs. The disadvantage of this approach is that it requires the definition of regular regions having the minimum mean distance of PEs to the L2 cache bank. Additionally, it is desired that regions have a similar number of PEs so that the bandwidth required by the cache controllers is nearly the same. This is not possible to achieve in some situations depending on the positioning of L2 cache banks.

Table 4 shows the distance in number of hops between each PE and L2 cache bank for a 4x4 2D NoC configuration. The minimum distance (marked in gray) of a PE to a given bank is used for defining the region that the PE belongs to. For instance, the closest bank to PE whose network address is 1 is the Bank 0. Consequently, PE 1 will be part of the same region of bank 0. The same approach is applied to all PEs to compose regions. Figure 29 shows a 2D representation of the MPSoC presenting the resulting regions delimited by the rectangles.

Table 4 – Distances in number of hops of all PE to all L2 cache banks (placed at address 0, 3, C, F).

PE Address	Bank 0	Bank 3	Bank C	Bank F
1	1	2	5	4
2	2	1	4	5
4	4	1	2	5
5	3	2	3	4
6	2	3	4	3
7	1	4	5	2
8	2	5	4	1
9	3	4	3	2
A	4	3	2	3
B	5	2	1	4
D	5	4	1	2
E	4	5	2	1

Considering the scenario illustrated at Figure 29 where task T2 is mapped at PE Slave 7. This task accesses a block α located at the L2 cache bank attached to C router, therefore, assuming the NoC uses deterministic Hamiltonian routing, for each access of task T2 to that block, the packet traverses 5 routers. If block α is mostly accessed by the PEs of the same region of PE where task T2 is mapped, it could be migrated to the L2 cache bank attached to router 0, decreasing the path traversed from 5 to 1 router. This migration would reduce the energy consumption in the NoC for data accesses of task T2.

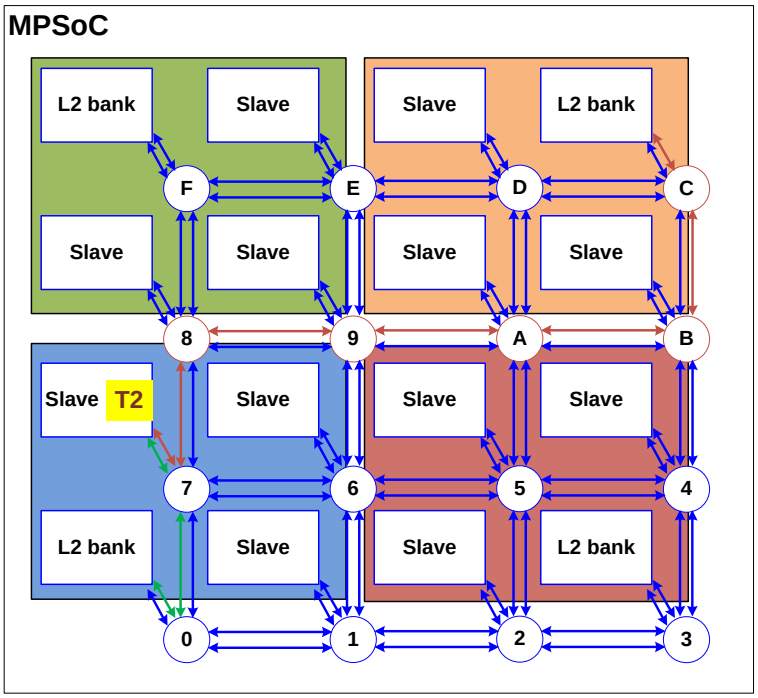


Figure 29 – Regions (rectangles in the figure) defined according to the minimum distance to the L2 cache bank.

The following subsections detail the implementation of the four mechanisms that compose the data migration heuristic: access accounting; free space verification; block migration; and block search.

6.1 Block Accesses Accounting

To account accesses done to a given block by PEs, each block of the directory memory was extended to include *N* 4-bit counters, where *N* is the number of regions in the MPSoC. As each PE belongs to a region, for every access performed to a given block of the L2 cache, the region counter is incremented. Each counter maintains the number of accesses done by PEs of the same region to that block. When one of the counters reaches a pre-defined *threshold*, it indicates that the block should be migrated. The *threshold* corresponds to the minimum difference between counters of the same block to trigger block migration.

Figure 30 shows an algorithmic description of the *counter checker* module, which is implemented in hardware as a finite state machine. The algorithm receives as input a vector of counters, whose length is equal to the number of regions in the MPSoC. This vector stores the number of accesses performed by each region to the lastly accessed block of the L2 cache bank. An auxiliary index $region_i$ is used to point to a position of the vector, which will be compared to all region counters. The first while loop, starting at line 6, moves the $region_i$ pointer through the vector array. The inner loop (lines 12-19), compares the region's counter pointed by $region_i$ to the counter of the other regions, whose location in the array is pointed by $region_j$. The idea of the inner loop is to compare the counters of

all regions to the region pointed by *region*$_i$. If the difference of the counter pointed by *region*$_i$ to the one pointed by *region*$_j$ is higher than a given *threshold* for all cases, the *migrate_counter* will be incremented *N-1* times, where *N* is the number of regions of the MPSoC. It is not equal to *N* because during the comparison *region*$_i$ and *region*$_j$ point to the same region once in the iteration. When *migrate_counter* variable is equal to *N-1* the migration is triggered (line 23).

```
Input: Region counters (counters) for a given block and a threshold.
Output: The decision to migrate or not a given block.
1.    // auxiliary variable that stores the index of the region being
2.      // compared with others
3.    region_i ← 1
4.    // The number of regions in the MPSoC
5.    total_regions ← the number of regions in the MPSoC
6.    WHILE region_i < total_regions DO
7.       // pointer to the region to be compared to the one pointed
8.       // by region_i
9.       region_j ← 1
10.      // auxiliary counter that will help deciding to migrate
11.       migrate_counter ← 0
12.       WHILE region_j < total_regions DO
13.          // The number of regions in the MPSoC
14.          diff ← counters[region_i] – counters[region_j]
15.          //  threshold is a constant defined by the user
16.          IF diff ≥ threshold THEN
17.             migrate_counter ← migrate_counter + 1
18.          END IF
19.       END WHILE
20.       // checks if the block should migrate
21.       IF migrate_counter = (total_regions-1) THEN
22.          // initiates the migration process
23.          migrate_block()
24.       END IF
25.   END WHILE
```

Figure 30 – Block access accounting algorithm.

6.2 Free space verification

The migration process is only performed if the L2 cache bank that will receive the migrating block has a free block slot to store it. Differently from other data migration approaches that swap two blocks of two different banks, our approach only migrates a block to a bank that has a free block slot.

To verify if the migration can be performed, after detecting that a given block can be migrated, the source L2 cache bank (where the block to be migrated is currently stored) sends a message to the target L2 cache bank requesting the address of a free block slot. If a free block slot is available, the target L2 bank reserves the position for receiving the migrating block and sends back the address of the free block to the source L2 bank.

A submodule of the L2 cache bank, named *free block searcher* (FBS) is responsible for finding the address of the first free block in the bank. The FBS module uses the strategy of a circular search as shown in Figure 31. Each entry of directory memory has a *valid* bit that indicates if the block is valid or not. If a given block is not *valid,* it can be used to store a migrating block.

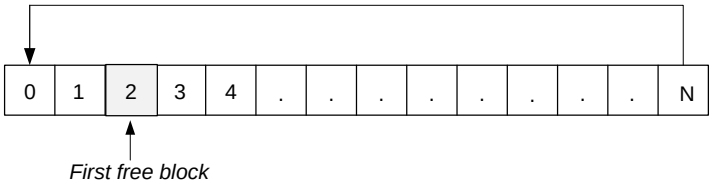


Figure 31 – Circular mechanism for verifying free blocks in the L2 cache bank.

The search algorithm starts checking if the first block is free (*invalid*) at position 0. If the block is not free, then, in the next *clock cycle*, the block 1 is checked. If a free block search has already been executed in the bank, the first block checked is the subsequent of the previous free block found. The same procedure is repeated for all blocks until a free block is found. If no block is found, the search is stopped at the position that it started.

The FBS module takes *L* clock cycles to finish, where *L* is the number of blocks of the L2 cache to detect that it is full. The advantage of using such a serial search is the minimum area overhead. A parallel approach could be used instead. Several parallel searches could be triggered. Although resulting in a best performance, such strategy requires additional control structures and area.

6.3 Migration process

The migration process, shown in Figure 32, occurs in four steps: (*i*) request free block address; (*ii*) receive free block address; (*iii*) block transfer; and (*iv*) update address mapping table in PEs having the block. The first step (Figure 32 – 1) and the second step (Figure 32 – 2) were covered in Section 6.2. After receiving the free block address, the source L2 cache bank sends a copy of the block to the target bank (Figure 32 – step 3) and invalidates its local copy. In the last step (Figure 32 – step 4), the source L2 cache bank checks the directory entry of the block and sends an update address message to all PEs that have a valid copy of the block. The update address message informs to the PE that the bank storing the block has changed, causing an update to the address table.

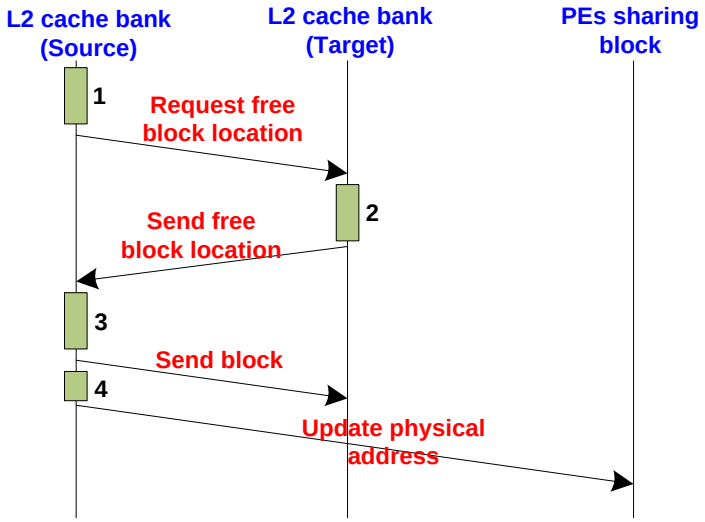


Figure 32 – Four steps of the data migration process.

Figure 33 shows the waveform of a logical simulation of block migration in an 8x8 MPsoC instance. In the figure it is possible to see all steps of the process. In the scenario shown in Figure 33 the entire data migration operation takes 740 clock cycles. The *timeslice* of a task is of 16384 clock cycles. Thus, the time spent in data migration represents, in this case, only 4.5% of the *timeslice*.

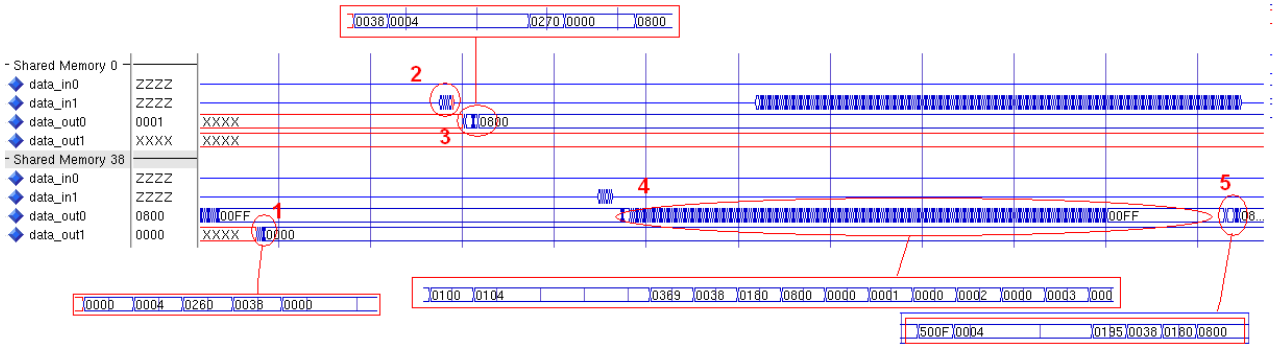


Figure 33 – Logical simulation of a data migration.

(1) In label 1 of Figure 33, L2 cache bank whose network address is 38 sends a *free block* request to the bank whose network address is 0. The message has 6 flits, as shown in Figure 34, being the two first used by the NoC to route the packet (target network address and size, respectively). The third flit (0x0260) indicates the service, the next flit represents the source network address of the message and the last flit has no valid information (padding).

flit 0	Target Address (0x0000)
flit 1	Size (0x004)
flit 2	Service (0x0260)
flit 3	Source Address (0x0038)
flit 4	Padding (No use)
flit 5	Padding (No use)

Figure 34 – Free block slot request format.

(2) In label 2, L2 cache bank 0 receives the message shown in (1).

(3) In 3, L2 cache bank whose address is 0 sends the location of a free block back to cache bank 38 in a message as shown in Figure 35. In this example, the first free block found is in the address 0x0800 (5th flit in the message).

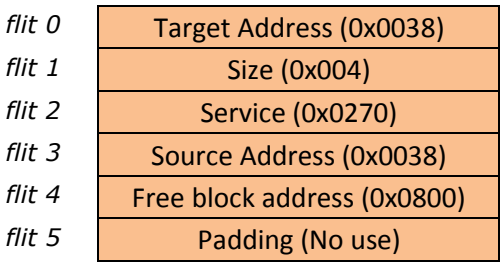


Figure 35 – Free block slot response format.

(4) In 4 L2 cache bank whose network address is 38 sends a copy of the block, which represents the migration of the block. The format of this message is shown in Figure 36. This operation takes 337 cycles to complete.

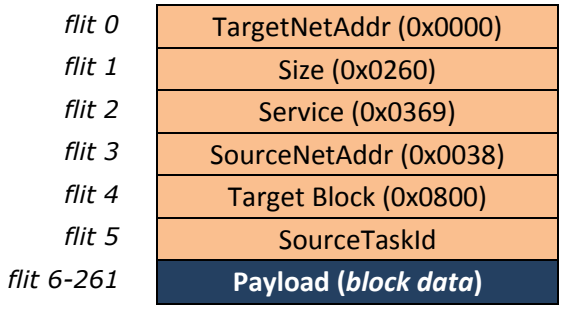


Figure 36 – Block migration message format.

(5) In 5, L2 cache bank whose network address is 38 updates all PEs with the new physical location of the block with a multicast message. The format of this message is shown in Figure 36. If in the future, any request arrives at bank 38 for the migrated block, it forwards the message to L2 cache bank 0.

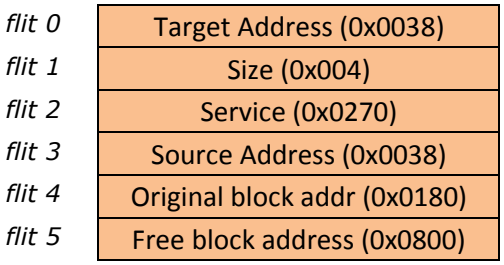


Figure 37 – Update address mapping table message format.

6.4 Migrated Block Search

In the data migration algorithm proposed in this work, at most two messages are required to find a block in the system. A search occurs every time a PE accesses a word from a block that is not in its local data cache (*cache miss*). In this situation, the following actions are performed:

(1) The microkernel of PE where the cache miss occurred sends a read request message to the L2 cache bank that stored that block for the first time. The target bank is determined based on the physical address of the block accessed;

(2) If the block has not been migrated, the cache bank provides the block to the PE. Otherwise, the first L2 cache bank forwards the read request to the bank that currently stores the block, which will respond to the PE.

A L2 cache bank stores, for each block, the network address of the L2 bank that stores the blocks that migrated from it. This approach requires the update of the network address of a given block after its migration.

6.5 Concluding remarks

The mechanisms of the proposed data migration protocol are evaluated in section 7.3. The results obtained from this study were submitted to the Networks-on-Chip (NoCs) 2012 conference.

7 RESULTS

This Chapter presents the experiments performed to validate and evaluate the solutions proposed in chapters 4, 5, and 6. All solutions were implemented in a RTL version of HeMPS platform and validated through functional simulation (using the ModelSim simulator).

To evaluate the consumed energy per memory transaction, the present work adopts the volume-based energy model proposed by Hu et al. [HU03]. 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.

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 (Synopsys) tool generates the power and energy values used in Equation 1.

This chapter is organized as follows: section 7.1 presents the experiments performed to evaluate the L2 cache bank and the NUCA organization (Chapter 4); section 7.1.5 presents the experiments performed to evaluate the proposed optimizations to the MSI cache coherence protocol (Chapter 5); section 7.3 presents the experiments proposed to evaluate the region-based data migration algorithm (Chapter 6).

7.1 Memory Organization experiments

According to [VER07], the way that applications access the memory can be classified in: *mostly-read*, *mostly-write* and a *mixed* of both. 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 cache block it must be read, we assume that mostly-write and mixed sets can be evaluated through the use of the same set of applications.

To evaluate the benefits of NUCA architecture proposed by this work in a NoC-based MPSoC, three sets of synthetic application were developed. Each set access the memory according to one of the three patterns described previously. For each pattern, 4 different implementations of the synthetic application are evaluated. Each implementation differs from the others in the number of tasks, which can be of 4, 6, 8 and 10 tasks. Each task

accesses four cache blocks of the L2 cache. In the mostly-read set, each task performs four read operations. For each block read, 40% of it is locally processed by the application before reading the next block. In the mostly-write set, each task executes four write operations. After each write, 40% of each block is modified by the task before starting writing in the next block. This simulates the *processing* time of the task.

Each set of the synthetic applications was simulated in a 4x4 instance of the HeMPS Platform. Each task is mapped in a separated PE (up to 10 PEs are used). The number of L2 cache banks is varied: 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 communication energy consumption.

Also, two applications were adopted as benchmarks to evaluate the benefits of NUCA over CSM architecture: Matrix Multiplication and Equation Solver.

7.1.1 Mostly-read set

The results for mostly-read applications set are shown in Figure 38. 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 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.

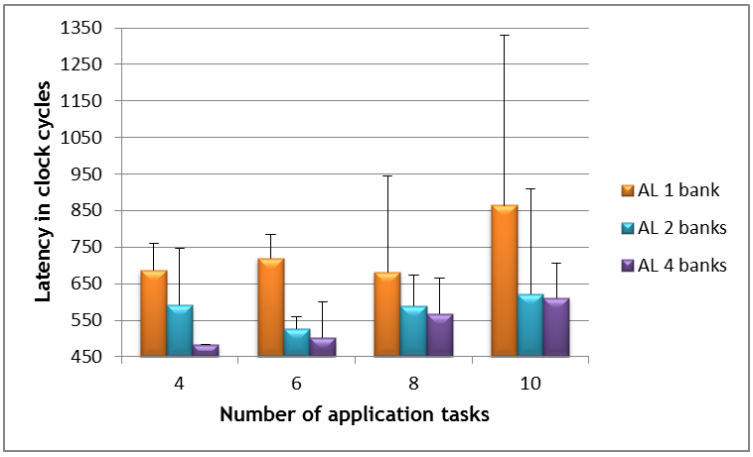


Figure 38 – Cache miss average latency (AL), expressed in number of clock cycles, for a varying number of applications tasks and L2 cache banks for the mostly-read set.

As shown in Figure 38, 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 blocks) is high for 10 tasks, which means that some of the tasks experience high miss latency in this configuration.

7.1.2 Mostly-write set

The results for the mostly-write application set are shown in Figure 39. 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 NUCA architecture can help reducing latency by distributing cache access messages across the NoC and also, by increasing access bandwidth.

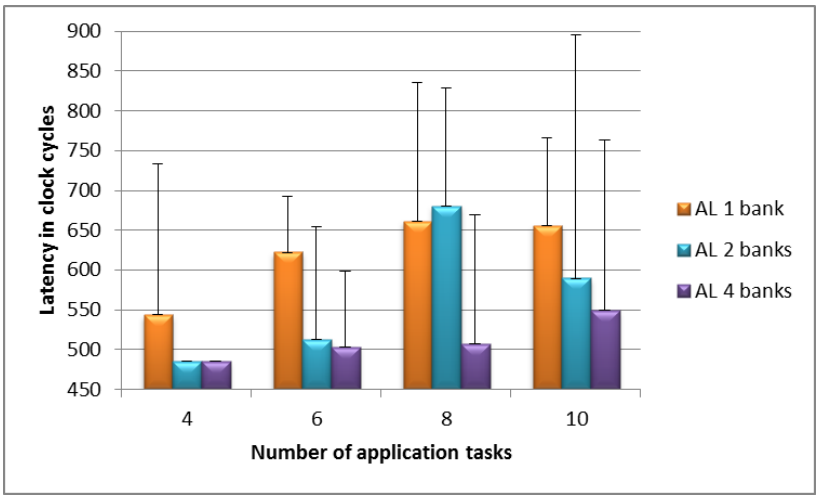


Figure 39 - Average write latency (AL), shown in number of clock cycles, for a given number of L2 cache banks. Also, worst write request latency (WL) for a given configuration of cache banks.

7.1.3 Matrix multiplication (MM)

The implementation of the *MM* application follows the traditional quadratic algorithm, with no optimizations. The core of the implementation of this application for the HeMPS platform is shown in Figure 40. The outmost loop iterates over the blocks of matrix, while the subsequent loop iterates over the columns of the matrix. The value of a cell of the resulting matrix is the result of the sum of all columns of the first matrix multiplied by all blocks of the second matrix. In the code, the *tmp* variable accumulates this value, and is then used to write the value to the resulting matrix (in the block where the function *write_block_word* is called).

The outmost loop can be parallelized. Each task can iterate over part of the blocks 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.

```
for(i = 0; i < NUMBER_OF_LINES; i++) {
    for( j = 0; j < NUMBER_OF_COLUMNS; j++) {
        tmp = 0;
        for(k = 0; k < MULT; k++) {
            tmp +=  read_block_word(i, k) * read_block_word(k+10, j);
        }

        write_block_word(0, i + result_offset, j, tmp);
    }
}
```

Figure 40 – Core part of the implementation of the matrix multiplication for the HeMPS platform.

- **Scenario 1**

Figure 41 presents the task graphs 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 between applications is that in application a *(a)*, both tasks access the same L2 cache bank, while in application b *(b)*, each task accesses a different L2 cache bank.

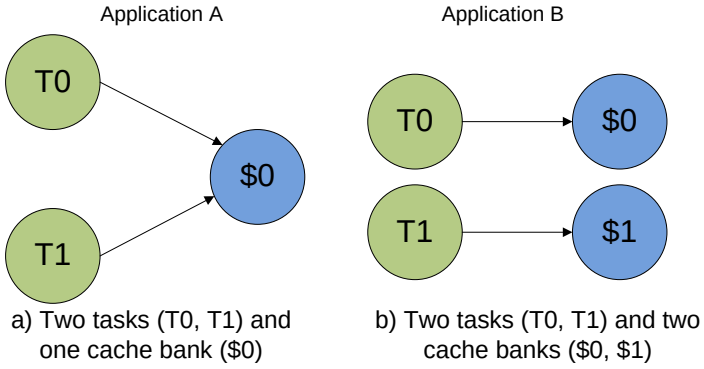


Figure 41 – Two different implementations of a MM application using 2 tasks.

Table 5 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 5 – 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

- **Scenario 2**

Figure 42 presents two task graphs for a MM application, which uses 3 tasks. The situation is similar to the one presented in Figure 41. The difference is that each task is responsible for generating 1/3 of the resulting matrix.

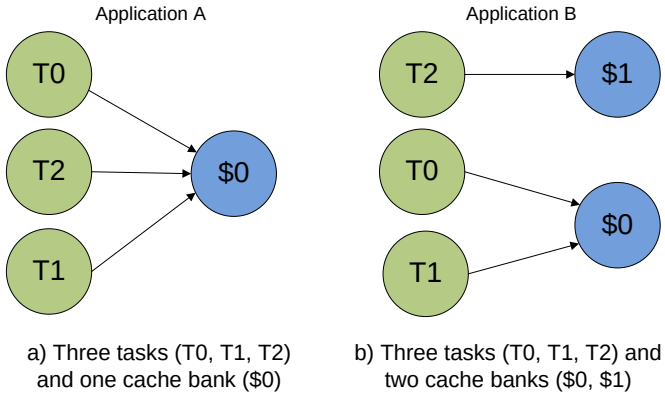


Figure 42 – Two different implementation of a MM application using 3 tasks.

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.

Table 6 – 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

7.1.4 Equation Solver

The equation solver kernel [CUL99] solves a simple partial differential equation on a grid, using what is referred to as finite differencing method. The kernel sweeps over the grid, excluding the border blocks, 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. The expression used for updating each point is described in Equation 2:

$$A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] + A[i+1,j]) \quad (2)$$

In Equation 2, A is the grid of values of the original expression.

Figure 43 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 blocks 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 (*synchronization*). The difference between task graphs *a*, *b* and *c* is the number of L2 cache banks used.

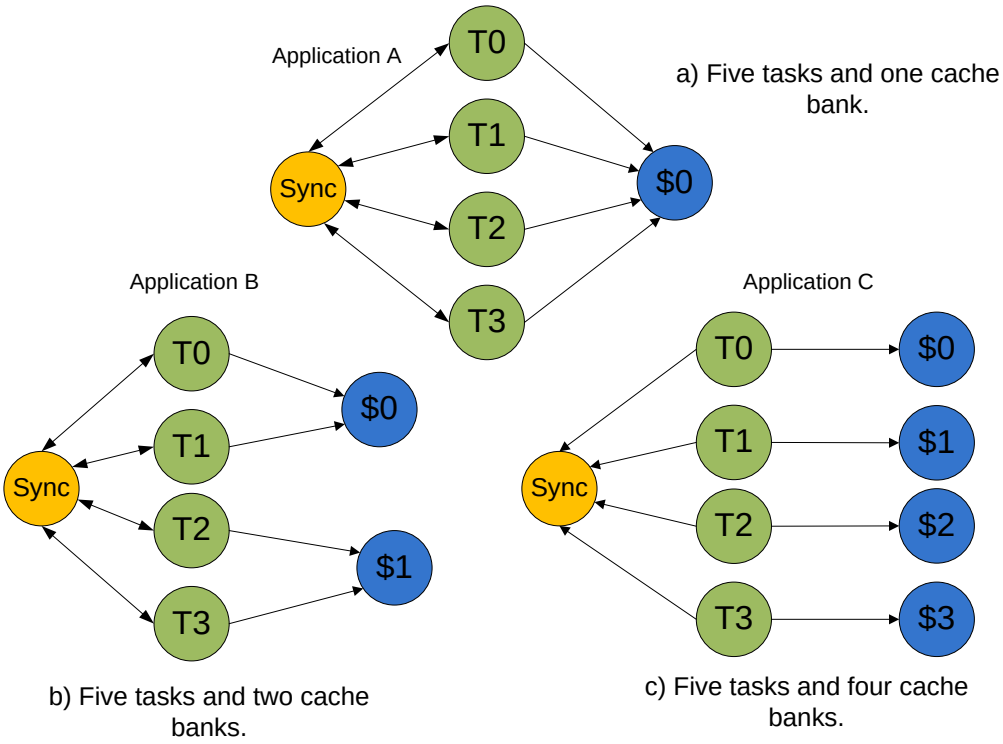


Figure 43 – Tasks graphs for equation solver application.

Table 7 shows the results for the three applications. In this case, the only gain using more cache banks is the reduction of the average cache miss, 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.

Table 7 – 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

7.1.5 Concluding remarks

The results for the MM applications presented a reduction in both latency and communication energy consumption as the number of L2 cache banks increases. The reduction in latency is justified by the increased bandwidth of the higher number of banks, whilst the reduction in energy consumption is justified by the fact that application tasks can be statically mapped closer to the L2 banks, decreasing the distance travelled by the packets of the memory transactions. The Equation Solver applications do not show significant reductions on latency because the load of simulated applications is small and therefore, the communication overhead overlooks the gain provided by using more L2 cache banks.

7.2 Cache Coherence Protocol experiments

This section presents experiments performed to compare the optimized cache-coherence protocol presented in Chapter 5, against a non-optimized MSI directory-based protocol.

7.2.1 Experimental Setup

The HeMPS 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 Chapter 5. 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 synthetic application tasks. Figure 44 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 44); *cache_write_block_word*, which writes to a given word of a shared block (indicated by number 2 in Figure 44). In the current example, the value 0x44 is written to the first word of block 1.

7.2.2 Invalidating cache blocks

In situations where more than one cache is sharing the same block of the L2 cache, the cache 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 8 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, 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%.

```
#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 44 - Example of an application reading and writing to the shared memory.

Table 8 – 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%

7.2.3 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 45 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.

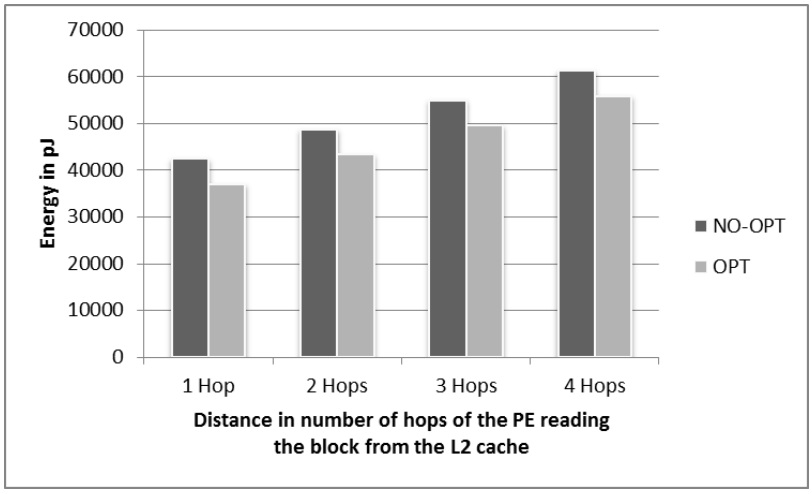


Figure 45 - Energy consumption of the read operation on a modified block as the number of hops increases.

7.2.4 Write request optimization

To evaluate the write optimization, a task after a write 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 48(a). The PE holding the modified copy of the block is fixed at PE00. The evaluated scenarios varied the position of the PE writing to a block of the cache.

Figure 46 shows that there is an average reduction of 17% in the number of clock cycles required to finish the write operation. Also, Figure 47 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 L2 cache can be bypassed because its copy of the block would be altered right after.

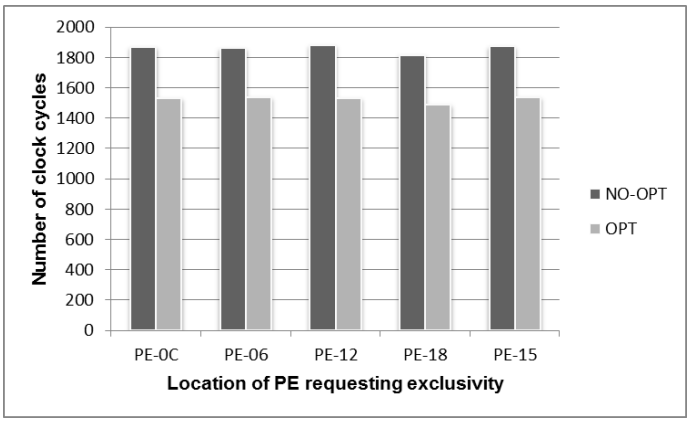


Figure 46 - Number of cycles required to execute a read operation on a modified block varying the location of the modified block.

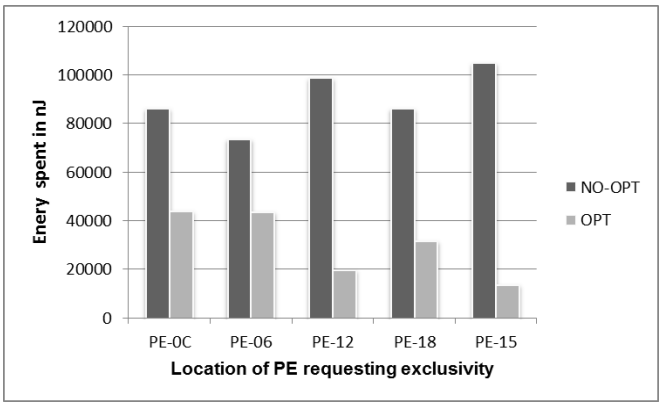


Figure 47 - Energy consumed to execute a read operation on a modified block varying the location of the modified block.

7.2.5 The Transition state

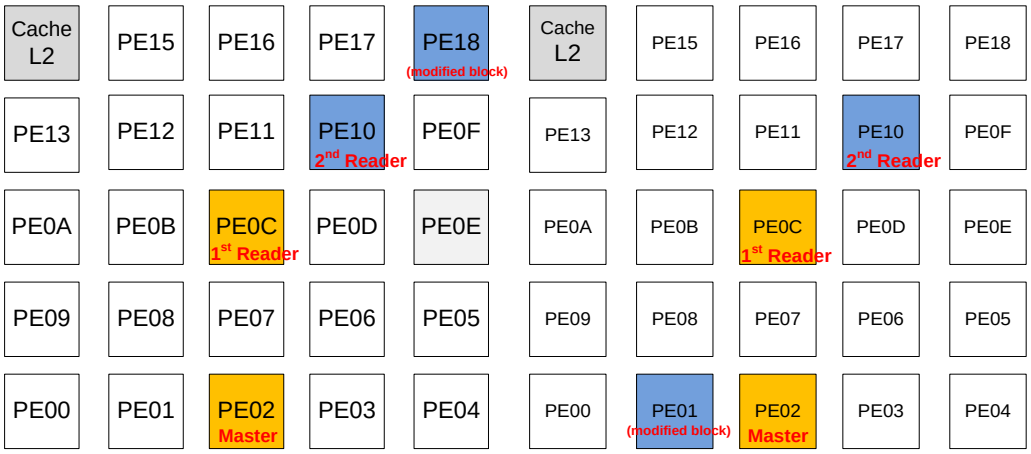
To evaluate the addition of the transition state, a scenario where 2 PEs issue subsequent reads to a modified block of the L2 cache is analyzed. The first PE which issues a read request will benefit from the Read request optimization, whilst the second PE will benefit from the addition of the T state. The NoC feature enabling this optimization is the duplicated physical channels, because while the L2 cache controller monitors 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 48(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 pJ 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 clock cycles, and the consumed energy. In Figure 48(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 pJ 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.



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

Figure 48 –Task mappings for the T state optimization.

7.2.6 Concluding remarks

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. 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.

7.3 Data Migration Protocol experiments

This section presents the experiments performed with the data migration algorithm in the HeMPS platform. The experiments presented in this section were performed in a HeMPS platform configured as follows: 6x6 and 8x8 2D mesh; four regions and one L2 bank per region. The *threshold* for data migration is *four* accesses.

Experiments were performed with a synthetic and a real applications. The synthetic application simulates the execution of 15 cache misses. In this way, 15 accesses are issued to a L2 cache bank. Between each read operations, a loop simulates the execution of the application. The real application used to evaluate data migration consists of a 16x16 matrix multiplication algorithm (section 7.1.3), implemented with four tasks. Two scenarios are proposed to evaluate energy consumption. In scenario 1, illustrated in Figure 49, two PEs (PE 11 and PE 14 – region 0) access a block located at L2 cache bank that is located at network address 30 (L2 Bank 30 – region 2). In this scenario, the energy consumed by data migration is important, as the data must be transmitted from a L2 cache bank that is 10 hops away from the original location of the block. The distance before data migration of the PE 11 and PE 14 to the accessed L2 cache bank is of 9 and 6 hops, respectively. After migration, the distance is reduced for 1 hop for PE 11 and 4 hops for PE 14.

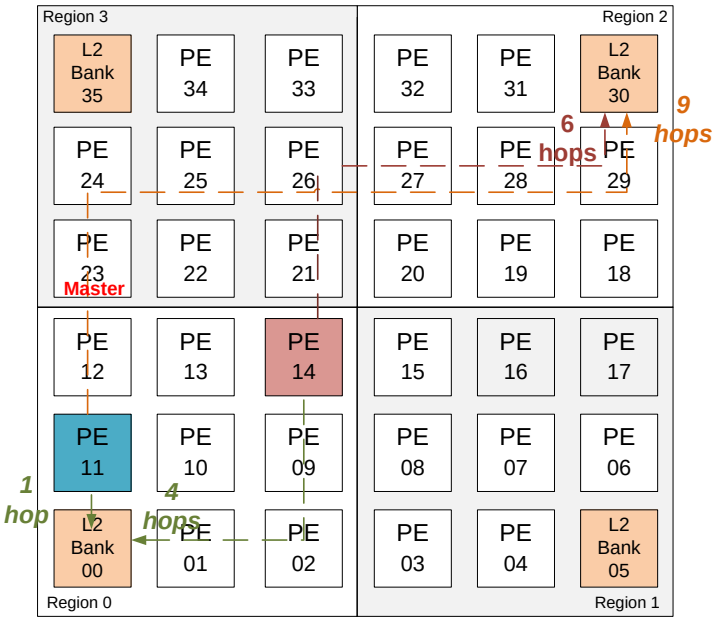


Figure 49 – Example scenario 1 for data migration, evaluates migration of a block from a distant region.

In scenario 2, illustrated in Figure 50, the same two PEs of the previous scenario access a block that is located at a neighbor region (region 3). The distances of the PEs to the accessed L2 bank, which is originally L2 bank 35, for PE 11 and PE 14 is 4 and 5 hops, respectively. After migration, distance is reduced to 1 and 4 hops, respectively.

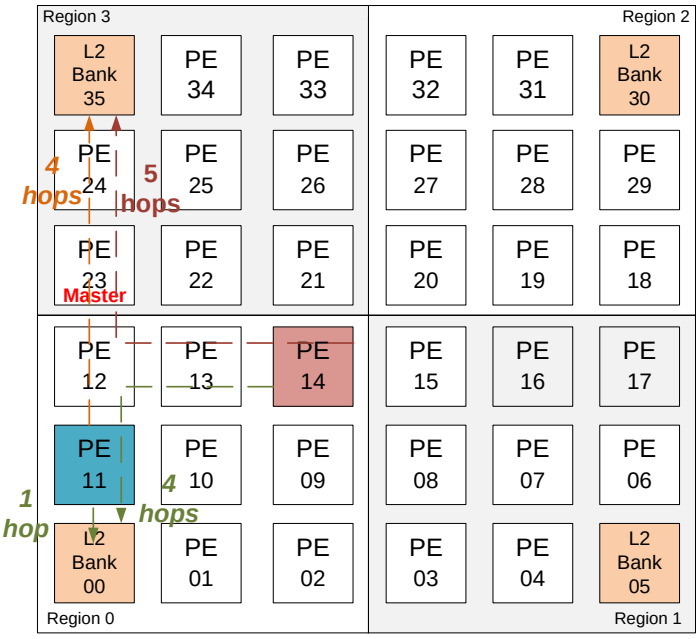


Figure 50 – Example scenario 2 for data migration, evaluates migration of a block to a neighbor region.

Both scenarios are used to evaluate data migration in an 8x8 NoC configuration. For all scenarios energy consumption and latency of cache accesses are measured before and after migration.

All tasks of the matrix application were mapped in Region 0, as shown in Figure 51 (PEs 01, 09, 10, 11). The data of the matrices were statically initialized at L2 Bank 35. This situation simulates a scenario where a task mapping heuristic mapped application tasks far from the location where the data they accesses are stored.

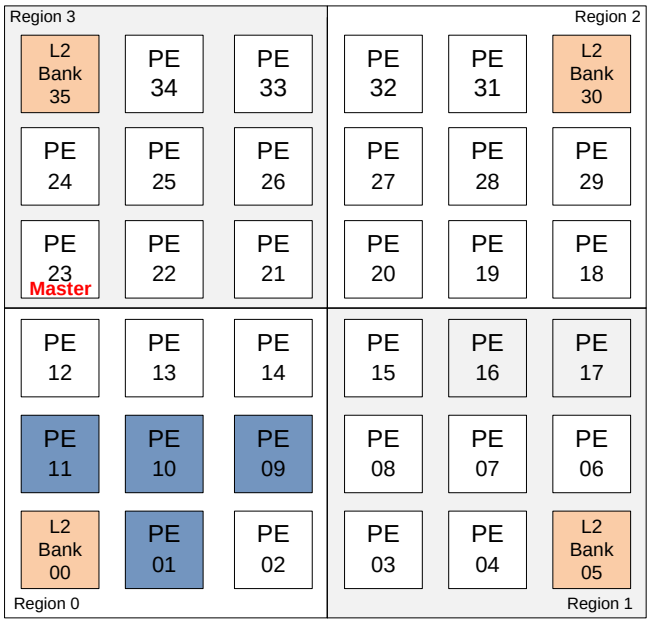


Figure 51 – Matrix multiplication task mapping.

7.3.1 Energy evaluation

Table 9 shows energy values for both scenarios and NoC configurations. It is possible to notice that a 52.4% reduction in energy consumption in communication was obtaining when migrating the block accessed by PE 11 from the L2 cache bank of region 2 to region 0 in the 6x6 NoC configurations. The significant reduction is obtained because after migration, the accesses messages must traverse only 1 hop instead of 9. The energy cost of migration is easily hidden in this situation. A smaller gain of 11% is noticed for PE 14. This is justified by the fact that for this PE the distance to the accessed L2 cache bank decreased only 2 hops. As the size of the NoC increases, from 6x6 to 8x8, it is possible to see that migrating the block close to the PE brings an even higher decrease of 54.7% for PE 11, although the decrease in energy consumption after migration for PE 1C, positioned in the border of two regions, is smaller than in the previous scenario. This happens because the decrease in energy consumption is small and the migration cost is high. Therefore, to amortize the migration cost, a significant number of operations must be performed in the migrated block after migration.

Table 9 – Energy values obtained for scenario 1 and 2 considering 6x6 and 8x8 NoC MPSoCs.

	6x6 MPSoC				8x8 MPSoC			
		Energy	PE 0x11	PE 0x14		Energy	PE 0xF	PE 0x1C
Scenario 1	Without data migration	Single access	878.3 nJ	590.3 nJ	Without data migration	Single access	1274.04 nJ	779.4 nJ
		Overall	13175.07 nJ	8723.4 nJ		Overall	19110.6 nJ	11691.1 nJ
	With data migration	Single access	86.9 nJ	383.7 nJ	With data migration	Single access	86.9 nJ	581.5 nJ
		Overall	6267.8 nJ	7751.7 nJ		Overall	8654.08 nJ	11127.2 nJ
		Overall reduction	52.4%	11.1%		Overall reduction	54.7%	4.8%
Scenario 2	Without data migration	Single access	383.7 nJ	482.6 nJ	Without data migration	Single access	581.5 nJ	680.4 nJ
		Overall	5755.6 nJ	7239.5 nJ		Overall	8723.4 nJ	10207 nJ
	With data migration	Single access	86.9 nJ	383.7 nJ	With data migration	Single access	86929.92 nJ	581.5 nJ
		Overall	3285.09 nJ	6747.5 nJ		Overall	4478 nJ	9919.1 nJ
		Overall reduction	42.3%	6.8%		Overall reduction	48.7%	2.82%

For the second scenario, the migration cost is smaller, as the source and target L2 cache banks for the migrated block are closer. The gains for PE 0x11 and PE 0xF that are

one hop from the target L2 cache bank is significant because PEs are closer to the L2 cache bank 0. Although, for the other PEs, the distance from the source L2 cache bank to the target one decreases by only one hop. Even in this case, the gains reach 4.8%.

The matrix multiplication was executed and evaluated with and without migration. Application tasks were mapped in a neighbor region of the L2 cache bank were the matrices were stored. A 7.2% reduction in energy consumption was obtained. The smaller reduction observed in this application comes from the fact that some blocks are constantly evicted from the L2 cache bank due to a conflict of blocks in direct mapping. The eviction of the blocks triggered the migration of one of these blocks, resulting in a smaller decrease of energy consumption.

7.3.2 Latency evaluation

The latency results obtained for both scenarios and matrix multiplication were similar. The evaluation of the first scenario (Figure 49) showed that without using data migration, the average miss latency was of *766 clock cycles*, and the worst case was of *1197 clock cycles*. When using data migration, the average miss latency increased slightly to *817 clock cycles*, and the worst case was the same of *1197 clock cycles.* The reason for the increase of 6% in number of clock cycles was that during the process of data migration, cache misses take longer to be treated by the L2 cache bank. Although NoC bandwidth is high, and data migration takes only about 535 clock cycles to finish in a 6x6 NoC configuration, misses that are issued during this period are penalized.

The latency results for the performed experiments showed that even in situations that a PE accesses data from a closer L2 bank, the number of clock cycles required to finish the operation does not decrease. This happens because, even for the cases where the L2 bank is farther, *network latency is significantly smaller than PEs latency*. Figure 52 shows the latency for cache access latency before and after migration considering an 8x8 configuration for the NoC. The accessed block was originally in L2 cache bank located at network address 56, and migrated to a L2 bank located at network address 00. The PE accessing the block has the network address 15. When the PE accesses the block in the distant memory, the read request takes 76 clock cycles to be received by L2 bank 56. This bank takes 6 clock cycles to start sending the requested block back to the PE. After 68 clock cycles the first flit arrives at PE 15. The packet issued in response operation takes 534 clock cycles to be consumed by the PE. When the response packet arrives at PE 15, it has not finishing processing the request packet that it had sent previously. This happens because after sending a read request packet, the PE runs the scheduler function of the microkernel, which looks if there is any task available to execute while waiting for the response packet.

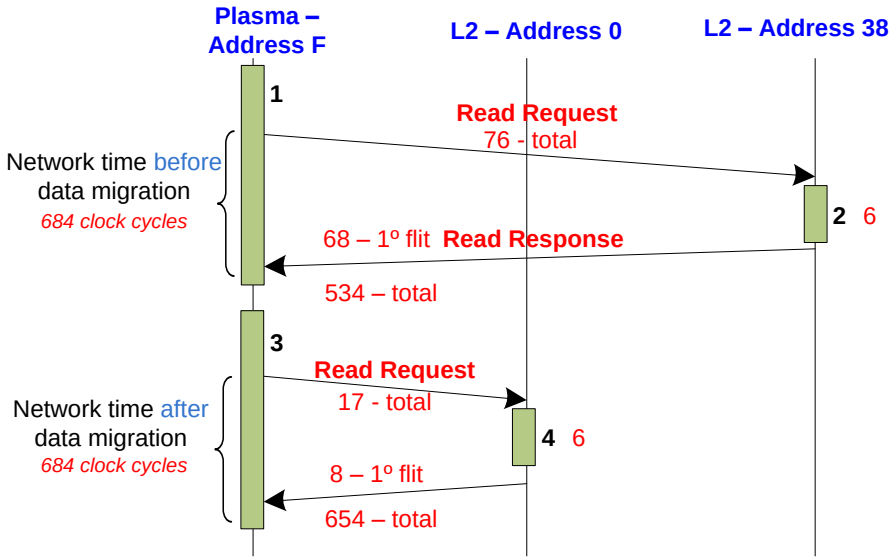


Figure 52 – Evaluation of number of cycles spent in each step of a read request operation before and after migration.

As the response packet tends to arrive back very fast, reducing network latency between the read request and response will not make the overall execution time of the read operation decrease. To illustrate this point, another read access is performed to the new L2 cache bank 00, which is one hop far from PE 15. Network time for the read request is significantly smaller, 17 clock cycles. Also, the number of clock cycles required for the first flit (of the response packet) arrives at the PE decreases from 68 to 8. Although, when the first flits arrive at PE 15, it is not ready for treating incoming packets from the NI as it is running the scheduler algorithm. Therefore, the overall number of clock cycles taken in both scenarios, before and after migration, is identical (684 clock cycles).

7.3.3 Concluding remarks

A significant reduction in energy consumption was obtained in both NoC configurations. For the 6x6 NoC configuration, a decrease of 52% was obtained considering the best scenario for data migration and 54.7% for the 8x8 configuration. The reduction average of all evaluated scenarios was of 29%. The matrix multiplication showed a decrease of 7.2% in energy consumption due to data migration of constantly evicted blocks.

Despite the fact that a decrease in network latency was obtained, the overall latency of L2 read operations performed in the L2 cache banks was not observed because the PE is not ready to treat the response packet at the moment it arrives back, even for the slower scenarios. This fact clearly demonstrates the large bandwidth offered by NoCs, serving PEs with small latencies.

8 CONCLUSIONS AND FUTURE WORKS

The increasing number of PEs in MPSoCs requires the study of memory solutions that present high bandwidth, low latency and low energy consumption. Allied to this, the exploration of the NoCs by memory mechanisms is required as this interconnection architecture becomes predominant.

This Dissertation contributed in three different topics related to memory organization in NoC-based MPSoCs:

- NUCA organization;
- cache coherence protocols for NoC architectures;
- data migration protocol.

The adoption of NUCA organizations helps eliminating the problem of latency in cache access because there is no need to consider the slowest latency as in uniform organizations. Distributing a single cache bank into several banks decreases wire delay, which is a serious problem in state-of-the-art chips [BEC04]. Also, the adoption of NUCA allows the development of data migration protocols such as the one proposed in this work.

The development of cache coherence protocols for NoC-based MPSoCs must consider the interconnection mean. This exploration enables the optimization of the protocol resulting in both smaller energy consumption and latency. The use of physical multicast messages in the implementation of directory-based protocols is mandatory as the number of PEs increases to decrease the number of issued packets in the network. The MSI directory-based protocol proposed in this work showed that it is possible to reduce the energy consumed by the memory operations up to 86.8% (average reduction: 39%).

The development of data migration protocols is essential for NUCA architectures because data are *attracted* to cache banks nearby the PEs that mostly access it. The advantages of accessing closer banks are the smaller energy consumption, smaller network latency that helps preventing the occurrence of congestion in some areas of the NoC. The data migration protocol proposed by this work showed that it is possible to decrease energy consumption up to 54.7% (average of 29%) and network latency. Although due to the latency of the *software layer*, for the presented experiments the overall memory operations time has not decreased even when having a smaller latency in the NoC.

8.1 Future works

The current work is an initial effort performed by the Author to explore NUCA organizations and study adaptive memory mechanisms for NoC-based MPSoCs having

several PEs. Future works include a deeper study and evaluation of the mechanisms proposed by this work and software mechanisms that are required to provide memory consistency. Future works include:

- Synthetize the L2 cache bank controller and evaluate area and static energy consumption;
- Compare the area of a synthetized L2 cache bank with the area consumed by the PLASMA PE;
- Evaluate the impact of NoC topology on the cache coherence protocol;
- Use physical network priority to evaluate transactions of the cache coherence protocol;
- Perform a deeper evaluation of the proposed data migration protocol considering the influence of the cache coherence protocol;
- Study and evaluate new heuristics to trigger data migration;
- Study and evaluate new heuristics to determining better data placement;
- Study and evaluation of a memory consistency model at the software level.

REFERENCES

[ADV91] Adve S. V.; Adve, V. S.; Hill M. D.; Vernon M. K. “Comparison of hardware and software cache coherence schemes”. Computer Architecture News, vol. 19-3, 1991, pp. 298-308.

[BEC04] Beckmann, B.; Wood D. “Managing Wire Delay in Large Chip-Multiprocessor Caches”. In: 37th MICRO, 2004, pp. 319-330.

[BER01] Bergamaschi, R. A.; Bhattacharya, S.; Wagner, R.; Fellenz, C.; Muhlada, M.; Lee, W. R.; White, F.; Daveau, J. "Automating the Design of SOCs Using Cores". IEEE Design and Test, vol. 18-5, 2001, pp. 32-45.

[CAR09] Carara, E. A.; Oliveira, R. P.; Calazans, N. L. V.; Moraes, F. G. “HeMPS - a framework for NoC-based MPSoC Generation”. In: ISCAS, 2009, pp. 1345-1348.

[CAR08] Carara, E. A. "Estratégias para Otimização de Desempenho em Redes Intra-Chip - Implementação e Avaliação sobre a Rede Hermes". Master Thesis. PPGCC, PUCRS, 2008, 89p.

[CAR11] Carara, E. A. "Serviços de Comunicação Diferenciados em Sistemas Multiprocessados em Chip Baseados em Redes Intra-Chip ". Doctoral Thesis. PPGCC-PUCRS, 2011, 107p.

[CEN78] Censier M.; Feautier P. “A New Solution to Coherence Problems in Multicache Systems”. IEEE Transactions on Computers, vol. C-27(12), 1978, pp. 1112–1118.

[CHA11a] Chaves, T. M.; Moraes, F. G.; Carara, E. A. "Exploiting Multicast Messages in Cache-Coherence Protocols for NoC-based MPSoCs" In: ReCoSoC, 2011, pp. 1-6.

[CHA11b] Chaves, T. M.; Carara, E. A.; Moraes, F. G. “Energy-efficient Cache Coherence Protocol for NoC-based MPSoCs”. In: SBCCI, 2011, pp. 215-220.

[CHT09] Chtioui, H.; Atitallah, R. B.; Niar, S.; Dekeyser, J.; Abid, M. "A Dynamic Hybrid Cache Coherency Protocol for Shared-Memory MPSoC". In: DSD, 2009, pp. 27-29.

[CUL99] Culler, D.; Singh, J.; Gupta, A. “Parallel Computer Architecture: A Hardware/Software Approach”. Morgan Kaufmann, 1st edition, 1999, 1056p.

[EIS08] Eisley N.; Peh L.; Shang L.; “Leveraging on-chip networks for data cache migration in chip multiprocessors”. In: PACT '08, 2008, pp. 197-207.

[EVE10] Carvalho E.; Calazans N.; Moraes F. “Dynamic Task Mapping for MPSoCs”. IEEE Design and Test of Computers, vol. 27(5), 2010, pp. 26-35.

[GUP97] Gupta, R.; Zorian, Y. "Introducing Core-Based System Design". IEEE Design & Test of Computers, v. 14(4), 1997, pp. 15-25.

[HAR72] Harary, F. “Graph Theory”. Addison-Wesley, 1972, 274p.

[HEN03] Hennessy, J.; Patterson, D. "Computer architecture: A quantitative approach". Morgan Kaufmann, 2003, 704p.

[HU03] Hu J.; Marculescu, R. “Energy-aware mapping for tile-based NoC architectures under performance constraints”. In: ASP-DAC, 2003, pp. 233-239.

[ITR11] International Tecnology Roadmap for Semiconductors. “ITRS update 2010”. Accessed in: http://www.itrs.net/reports.html. November 2011.

[JER05] Jerraya, A. A.; Wolf, W. “Multiprocessor Systems-on-Chips”. Morgan Kaufmann, 2005, 602p.

[JER08a] Jerger, E.; Peh, L.; Lipasti, M. H. "Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence". In: 41st MICRO, 2008, pp. 35-46.

[JER08b] Jerger, N. E.; Peh L. S.; Lipasti M. “Virtual Circuit Tree Multicasting: A Case for On-Chip Hardware Multicast Support”. Computer Architecture News, vol. 36 (3), 2008, pp 229-240.

[JHA11] Jhalani, D.; Palsetia, D. "Adaptive Cache Coherence Protocol Using Migratory Shared Data". Accessed in: http://www.seas.upenn.edu/~palsetia/. January 2011.

[KAN05] Kandemir, M.; Dutt, N. “Memory Systems and Compiler Support for MPSoC Architectures”. In: Multiprocessor Systems-on-Chips, Kluwer Academic Publishers, 2005, pp. 251–281.

[KIM06] Kim, D.; Kim, M.; Sobelman, G.E. "DCOS: cache embedded switch architecture for distributed shared memory multiprocessor SoCs". In: ISCAS, 2006, pp. 4-8.

[KUM02] Kumar, S.; et al. “A Network on Chip Architecture and Design Methodology”. In: ISVLSI’02, 2002, pp. 105-112.

[LIM09] Limberg T.; et. al. “A Heterogeneous MPSoC with Hardware Supported Dynamic Task Scheduling for Software Defined Radio”. In: DAC University Booth, 2009, 6p.

[MAN10] Man C.; et al. "Distributed Memory Management Units Architecture for NoC-based CMPs". In: 10th CIT, 2010, pp. 54-61.

[MIL04] Millberg, M.; Nilsson, E.; Thid, R.; Kumar, S.; Jantsch, A. “The Nostrum backbone – a communication protocol stack for networks on chip”. In: VLSI Design Conference, 2004, pp. 693-696.

[MON06] Monchiero, M.; Palermo, G.; Silvano, C.; Villa, O. "Exploration of Distributed Shared Memory Architectures for NoC-based Multiprocessors". In: IC-SAMOS, 2006, pp.144-151.

[MON08] Monchiero, M.; Palermo G.; Silvano, C.; Villa O. “A Modular Approach to Model Heterogeneous MPSoC at Cycle Level”. In: EUROMICRO, 2008, pp. 158-164.

[MOR04] Moraes, F.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. "Hermes: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip". Integration, the VLSI Journal, Vol.38(1), Outubro, 2004, pp.69-93.

[NAF10] Nafziger, J.; Avakian, A.; Vemuri, R. "A Prediction-based Data Migration Algorithm for Hybrid Architecture NoC Systems". In: SOCC, 2010, pp. 435-440.

[OPE11] OpenCores. “Plasma open core”. Accessed in: http://www.opencores.org/. January 2011.

[OPH09] Ophelders, F., Bekooij, M. J., Corporaal, H. "A tuneable software cache coherence protocol for heterogeneous MPSoCs". In: CODES+ISSS'09, 2009, pp. 383-392.

[PET06] Petrot, F., Greiner, A., Gomez, P. "On Cache Coherency and Memory Consistency Issues in NoC Based Shared Memory Multiprocessor SoC Architectures". In: DSD, 2006, pp. 53-60.

[SAI07] Saint-Jean, N.; Sassatelli, G.; Benoit, P.; Torres, L.; Robert, M. “HS-Scale: a Hardware-Software Scalable MP-SOC Architecture for Embedded Systems”. In: ISVLSI, 2007, pp. 21-28

[SIL09] Silva, G. G. B.; Barcelos D.; Wagner, F. R. "Performance and Energy Evolution of Memory Hierarchies in NoC-based MPSoCs under Latency". In: VLSI-SoC’09, 2009.

[TOT10] Tota, S.V.; Casu, M.R.; Roch, M.R.; Rostagno, L.; Zamboni, M. "MEDEA: a hybrid shared-memory/message-passing multiprocessor NoC-based architecture". In: DATE, 2010, pp.45-50.

[VER07] Verma, M.; Marwedel, P. “Advanced Memory Optimization Techniques for Low-Power Embedded Processors”. Springer Publishing Company, 1st Edition, 2007, 161p.

[YUA08] Yuang, Z.; Li, L.; Shengguang, Y.; et al. "A scalable distributed memory architecture for Network on Chip". In: APCCAS 2008, 2008, pp.1260-1263.