UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GUSTAVO GIRAO BARRETO DA SILVA

Resource-Aware Clustering Design for NoC-based MPSoCs

Thesis presented in partial fulfillment of the requirements for the degree of Doctor in Computer Science

Prof. Dr. Flávio Rech Wagner
Advisor

Porto Alegre
2014

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Silva, Gustavo Girão Barreto da
Resource Aware Clustering Design for NoC-based MPSoCs [manuscrito] / Gustavo Girão Barreto da Silva. – 2014.
146 f.:il.
Orientador: Flávio Rech Wagner.
Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2014.
1.Multiprocessors. 2.Networks-on-chip. 3.Cluster. 4. Resource Management. 5.Parallel Programming. 6.Reliability I. Wagner, Flávio Rech. II. Título.

Para minha mãe.

AGRADECIMENTO

Agradeço, primeiramente, à minha mãe que sempre me apoiou em tudo que eu fiz e principalmente nos meus estudos. Ela sempre foi meu porto seguro quando precisei e nunca imaginou, por um segundo sequer, que eu não seria capaz de alcançar qualquer objetivo que eu tivesse planejado.

Agradeço a meu irmão e minha cunhada que me deram palavras de encorajamento em momentos que me senti sem ânimo. Também preciso agradecer à minha namorada Patrícia que foi presença constante nos momentos finais (porém fundamentais) do desenvolvimento desta tese .

Agradeço muito ao meu orientador Flávio Rech Wagner que cumpriu com perfeição seu papel. Em alguns momentos deste doutorado estive à deriva e não conseguia achar um direcionamento. Ele foi responsável por me guiar nestas horas e me fazer enxergar novamente meus objetivos. Sei que dei muito trabalho e mesmo assim ele nunca desistiu de mim. Por isso, sou muito grato.

Durante o doutorado é natural que muitas pessoas tenham contribuído, diretamente ou não, para sua realização. Não serei capaz de citar todas as pessoas que me ajudaram neste longo caminho mas quero que saibam que todas foram lembradas na finalização deste trabalho. Sua ajuda foi fundamental para me motivar a concluir esta jornada e para que a mesma não fosse em vão.

Não poderia ter feito este trabalho sem o apoio (científico ou não) de colegas do LSE como Monica Pereira, Ronaldo Ferreira, Daniel Barcelos, Mateus Rutzig, Tomás Moreira, Ulisses Brisolara, Thiago Santini, Leonardo Kunz, Gabriel Nazar, Márcio Oliveira, Marco Wehrmeister, entre outros. Vocês, mais do que outras pessoas, sabem o quanto este caminho é cheio de obstáculos e, por vezes, pouco reconhecido. A minha gratidão pelo companheirismo de vocês é algo ficará para sempre.

Last, but not least, I would like to express my appreciation for all members of the Dutt Research Group (Dr. G!): Abbas, Danny, Hossein, Jan, Trent, Santanu, Soo, Codrut and Prof. Dutt. You all contributed immensely for my work and also made me feel very welcomed inside and outside the campus. It was a great experience because of you. Thank you very much!

Obrigado.

Projeto de MPSoCs baseados em NoC utilizando Clusterização e Gerenciamento de Recursos

RESUMO

Atualmente, o paradigma multicore é uma tendência fortemente estabelecida também na área de sistemas embarcados. O grau de paralelismo provido por tal arquitetura tem sido a principal causa de avanços de performance na área além de economia de energia e potência. Entretanto, para obter paralelismo eficiente desta arquitetura não é uma tarefa simples. Assim, desenvolvedores propuseram diversos modelos de ambientes de programação tentando prover o máximo de transparência possível. No nível do hardware, este crescente aumento no número de componentes dentro chip cria um problema de gerenciamento a ser tratado. No contexto deste cenário complexo, esta tese propõe o uso de abordagens de gerenciamento de recursos para aumentar a eficiência, levando em consideração tanto performance quanto consumo de energia, de ambientes MPSoC em diferentes níveis. Além disso, estas abordagens tem em comum a noção de clusterização, a qual tenta agregar recursos logicamente de acordo com as demandas da aplicação. Primeiramente no nível do processador/aplicação, é proposto um hardware dinamicamente adaptável para suportar modelos de programação paralelos distintos sem nenhum sobrecusto computacional uma vez que todo o processo é completamente transparente para o programador. Ainda neste ambiente, onde aplicações distintas podem ser executadas, é proposto um mecanismo de escalonamento visando gerenciamento de recursos para aumentar a performance chamado *Processor Clustering*. São propostas quatro diferentes políticas de mapeamento de recursos que tiram vantagem de aspectos distintos da natureza paralela das aplicações e das restrições arquiteturais do sistema. Entretanto, algumas aplicações tem demandas de memória mais altas do que demandas computacionais. Logo, uma abordagem similar pode ser utilizada no nível da hierarquia de memória. Neste caso, o objetivo é redistribuir recursos de memória de acordo com as demandas da aplicação. Redistribuição de memória é explorada tanto em tempo de projeto quanto em tempo de execução. Um mecanismo de mapeamento de distribuição é proposto baseado na quantidade de requisições de acesso à memória externa. Finalmente, é proposto um mecanismo de tolerância à falhas baseado em gerenciamento de recursos para memórias distribuídas dentro do chip em NoCs. É introduzido um modelo de Reliability Clustering que tira proveito da infraestrutura da NoC. Neste caso, os roteadores tem conhecimento dos blocos com falhas e blocos redundantes. Baseado neste conhecimento, o mecanismo é capaz evitar altas latências de acesso à memória.

Palavras-chave: Multiprocessadores, Redes-em-Chip, Cluster, Gerenciamento de Recursos, Programação Paralela, Confiabilidade.

Resource-Aware Clustering Design for NoC-based MPSoCs

ABSTRACT

The multicore paradigm is a solid trend nowadays, also in the field of embedded systems. The degree of parallelism provided by such architecture has been the foundation of performance advancements in the field as well as for power and energy savings. However, to obtain efficient parallelism of such architecture is not an easy task. Therefore, developers come up with several proposals of programming environments trying to provide as much transparency as possible. On the hardware side, this increasing number of on-chip components creates a management issue to be handled. In the context of this complex scenario this thesis proposes the use of resource management approaches to improve the efficiency, regarding both performance and energy consumption, of MPSoC environments at different levels. Also, these approaches have in common the notion of clustering, which tries to logically aggregate resources according to application demands. First, at the processor/application level, we propose a dynamically adaptable hardware to support distinct parallel programming models at no computational overhead, since the entire process is completely transparent to the programmer. Also, in this environment, where distinct applications can be executed, we propose a resource-aware scheduling mechanism to improve performance named Processor Clustering. We propose four different resource mapping policies that leverage on distinct aspects of the parallel nature of the applications and on architecture constraints. However, some applications have higher memory demands than computational demands. Therefore, a similar approach can be used at the memory level. In this case, we aim at redistributing memory resources according to application demands. We explore memory redistribution at both design time and runtime and propose a distribution mapping mechanism based on the amount of off-chip memory requests. Finally, we propose a resource-aware fault-tolerance mechanism for distributed on-chip memories in NoCs. We introduce a Reliability Clustering model that leverages on the NoC infrastructure. In this case, the routers have knowledge of faulty blocks and redundancy blocks and, based on that, they are able to avoid higher memory access latency.

Keywords: Multiprocessors, Networks-on-Chip, Cluster, Resource Management, Parallel Programming, Reliability.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ACG	Application Control Graph
AMURHA	Adaptive Multi-grained Hardware Architecture
API	Application Programming Interface
ATA	Address Table
BE	Best Effort
BIST	Built-In-Self-Test
CaCoMa	Cache Communication Manager
CIC	Core i-let1 Controlers
CMP	Chip Multiprocessor
CM	Cluster Mirror
CS	Cluster Shape
DCT	Discrete Cosine Transform
DDR	Double Data Rate
DIMM	Dual Inline Memory Module
DMA	Direct Memory Access
DVFS	Dynamic Voltage and Frequency Scaling
ECC	Error Correction Codes
EDF	Earliest Deadline First
EDP	Energy Delay Product
FIFO	First In First Out
FMB	Fault Map Block
FPGA	Field-Programmable Gate Array
FT	Fault Tolerant
GMP	Global Master Processing element
GS	Guaranteed Service
HWL	Higher Workload
ILP	Instruction-Level Parallelism

IPC	Instructions per Cycle
iRTSS	Invasive Runtime Support System
ITRS	International Technology Roadmap for Semiconductors
JPEG	Joint Photographic Experts Group
LLC	Last Level Cache
LMP	Local Master Processing element
ME	Motion Estimation
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MM	Matrix Multiplication
MPEG	Moving Picture Experts Group
MPI	Message Passing Interface
MPMMU	Multiprocessor Memory Management Unit
MPSoC	Multiprocessor System-on-Chip
MS	Mergesort
NH	Number of Hops
NI	Network Interface
NN	Nearest Neighbor
NoC	Network-on-Chip
NT	Number of Tasks
NUCA	Non-Uniform Cache Access
NUMA	Non-Uniform Memory Access
OpenMP	Open Multi-Processing
OCM	Off-Chip Memory
PE	Processing Element
PREM	Predictable Execution Model
PTA	Processor Table
QCIF	Quarter Common Intermediate Format
RDB	Remapping Data Block
RISC	Reduced Instruction Set Computing
RM	Resource Manager
RTOS	Real-Time Operating System
SCC	Single-chip Cloud Computer
SDF	Synchronous Data Flow
SVF	Shared Variables First

SIMD	Single Instruction Multiple Data
SIMPLE	Simple Multiprocessor Platform Environment
SISD	Single Instruction Single Data
SMP	Symmetric Multiprocessor
SoC	System-on-Chip
SoCIN	System-on-Chip Interconnection Network
SP	Slave Processing element
SPFP	Simple Partitioned FIFO Locking Protocol
STA	Status Table
TCPA	Tightly-Coupled Processor Array
TLP	Thread-Level Parallelism
TSV	Through-Silicon Via
WPPA	Weakly Programmable Processor Arrays

TABLE OF CONTENTS

1 INTRODUCTION

Since the last decades, embedded systems have become more and more complex. One of the reasons is because these systems have shifted their audience to users that demand a higher amount of applications. Smartphone users, for instance, not only run mp3 players, video recording, browsers but also several tasks that are not visible for them such as operating system services. In addition, these applications can be ever more computationally demanding (being composed of, potentially several threads) and very diversified, creating situations in which the loading of an entirely new application into the system is a real (and frequent) possibility. In order to support such scenario, the hardware of these systems has to become more robust.

In conventional computational systems, the use of processing elements with ever higher operating frequencies presents little performance enhancement (proportionally speaking) and an elevated energy consumption. Considering that embedded systems present severe energy restrictions, this kind of solution is prohibitive. Facing this problem and thanks to the high level of integration we have nowadays, the embedded systems paradigm has shifted to explore solutions composed of multiple processors in a single chip. This solution is known as Multiprocessor System-on-Chip (MPSoCs).

MPSoCs use multiple processors, which, typically, operate at a lower nominal frequency if compared to single-core conventional general purpose processors used until the beginning of the decade of 2000. This is a design choice with the goal of increasing the energy efficiency while keeping the required computational performance. This energy consumption situation has become a critical issue, both for high-end large-scale parallel systems, as well as for portable devices. For some application domains, specialized architectures deliver more performance per Watt than general-purpose processors. Research has shown that for such application domains, multi-processor systems (especially heterogeneous MPSoCs) can deliver higher performance at a given energy budget (LEE, 2007).

The parallelism of such systems is key to achieve gains of performance, and MPSoC solutions are strongly based on parallelism between applications (either at thread or process level) that are running on the processors. Some of the multiprocessed systems still use buses as communication mechanism. However, these structures have a lower degree of scalability, in a sense that the inclusion of few elements in the system leads to a rapid degradation on communication as a whole (LEE, 2007). With the goal of having a more efficient communication mechanism a solution heavily inspired on conventional computer networks, named Networks-on-Chip (NoC), has been proposed (BENINI, 2002). An NoC is a communication structure composed by several routers

interconnected following certain topology (i.e. Mesh, Torus, Cube). Each router is associated to a resource (processors, memories or I/O modules, for instance). Besides the high degree of scalability, NoCs have as main feature the ability to provide parallel communication amongst the system components. These features raise questions about the multiprocessed systems design when based on a communication mechanism such as NoCs. Several studies were and still are performed around this topic, such as Task Migration (NOLLET, 2005; BRIÈRE, 2005; BARCELOS 2007; DUMMLER, 2008; LATIF, 2010) and Memory Hierarchy (MONCHIERO, 2006; MARESCAUX, 2007; LIU-YAN, 2009; GIRÃO, 2009; ABOUSAMARA, 2012), or aspects related to the communication mechanism itself (BANERJEE, 2007; OGRAS, 2007; CONCATTO, 2009; QIAN, 2012; KUMAR, 2013).

In this document we present the proposal of Resource-aware Cluster-based solutions for NoC-based MPSoCs. In the last decades, the increase of resources in an embedded system has been taking these environments to the point where an efficient management is mandatory. The solutions proposed here try to improve the overall performance of these systems by using mechanisms that evaluate hardware and software aspects as metrics to distribute resources, mainly processor nodes and memory nodes, among applications that run in parallel. In addition, these solutions try to aggregate these resources using a virtual cluster-based approach. These cluster solutions are considered virtual because they do not use any physical restrictions on the resources when aggregating them. In this thesis proposal, the mechanisms presented act at three different layers: the processor/application layer, memory subsystem layer, and NoC/fault-tolerance layer.

In order to present this thesis proposal, we need the knowledge of several concepts and established mechanisms, which are introduced in the following sections.

1.1 Parallel Programmming on MPSoCs

As established in Flynn's taxonomy (FLYNN, 1966) which is based on the instruction and data flow of information in a computer, there are four main computer architectures

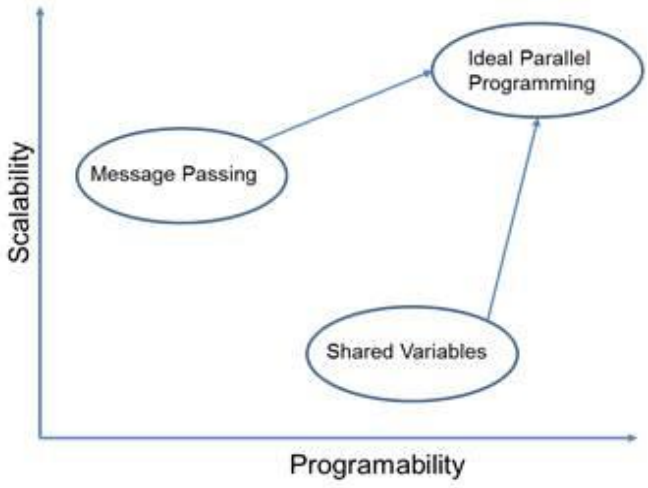
- **SISD** (single instruction, single data): Classical Von Neumann architecture in which each instruction initiates an operation on a data item taken from a single stream of data elements.
- **SIMD** (single instruction, multiple data): A single instruction being streamed to operate on multiple data. Typically this architecture is implemented in the form of Vector Processors, which are widely used in graphic cards.
- **MISD** (multiple instruction, single data): A single data stream flows through multiple processors, each one having its own instruction stream. This is a very unusual architecture, and the only implementations that relate to this model are the ones used for fault-tolerance purposes. In this case, a set of processors must execute on the same data and agree on the result.
- **MIMD** (multiple instruction, multiple data): Each processor has its own instruction stream and input data. This architecture is the foundation of multi-cores and MPSoCs. The idea is to fully exploit the parallelism of a certain task by simultaneously performing different operations over distinct data.

Based on that classification, computer scientists have tried for years to develop mechanisms for programming languages at high level that could take advantage of such models. As described previously, the rise of multiprocessor systems has increased efforts to provide parallel programming models for SIMD and MIMD machines.

Essentially, there are two types of parallel programming models: shared variables and message passing. Each of these models is more suitable for a specific memory organization. Shared variables are used in a shared memory organization, since this model relies on communication through reads and writes on a subset of variables that can be accessed by any processor. On the other hand, message passing uses explicit communication through primitives for sending and receiving data. Message passing is largely used in a distributed memory organization.

There are significant differences on the use of each of these models, and it is usually the programmer that, based on his/her expertise, decides which one to use, for instance considering which one would be more computationally efficient or more easily programmed for a particular application. Figure 1.1Source: Girão (2011, p. 63). highlights the differences between these programming models. Basically, the main differences are the programmability (i.e. how easy it is to develop an application) and scalability (i.e. how does the system performance degrades as the number of components increases) (FORSELL, 2002).

Figure 1.1 Characteristics of main parallel programming models.



Source: Girão (2011, p. 63).

Implementations of such models are widely used depending on the design choice. Message Passage Interface (MPI) (MESSAGE, 1995) is a specification for developers of libraries that implement the message passing model. Routines for sending and receiving messages (including variations that make use of different approaches) and synchronization are provided. Here, the programmer is responsible for explicitly determining which data are going to be sent, to which task and when this is going to happen. This approach creates degrees of freedom for the programmer to use different methods to parallelize the application. However, it also easily opens the possibility of deadlocks since a task may be waiting for a synchronization that never happens.

OpenMP (Open Multi-Processing) (OPENMP, 2002) is a Shared Variables-based implementation of an API that makes use of compiler directives and library routines to provide parallelization of loops in C, C++ and Fortran languages. OpenMP has different implementations for each of these languages, and, essentially, the compiler (using as reference directives provided by the programmer) tries to parallelize loops in order to execute these basic blocks as different threads. The main advantage of OpenMP is the level of abstraction that excludes the programmer from the fine grain decision making of the implementation. This decreases the chance of a deadlock problem.

The rapid increase in the number of cores in a single chip and the investment on the efficient use of Thread Level Parallelism (TLP) require techniques that make adequate use of these resources. These techniques cover aspects from topics such as parallel programming and task allocation and migration.

1.2 Resource Management

As discussed previously, MPSoCs are an undeniable trend in the semiconductor industry. For instance, nowadays we can find several examples of processors with dozens of cores, such as the Xeon Phi family (INTEL, 2013), the teraflop processor (HOSKOTE, 2007) and the Single-chip Cloud Computer (SCC) (HELD, 2010), all by Intel. There are expectations that the number of cores will continue to increase, leading to a thousand cores until the end of the decade (BORKAR, 2007).

This scenario induces several design challenges to the system, such as application mapping, fault tolerance design, hardware cost, power consumption, communication topology, and memory architecture. This growth also raises even more the issue of scalability. As mentioned previously, this scalability issue has been tackled at the level of communication mechanisms but, beyond this, the MPSoC resources must be managed to deliver the expected performance. Resource management duties may include access to I/O devices, task monitoring, mapping and migration, and DVFS (Dynamic Voltage and Frequency Scaling, in order to meet power requirements). The key idea to resource management is to maintain an efficient usage of all resources according to the system's workload. Usually this problem becomes more complex as the applications are not known beforehand and the workload may change drastically from time to time.

There are several strategies to cope with this situation. Virtually all of them rely on a resource manager entity that can either be a software application or a hardware component. At the same time, the resource management strategy can be of a centralized or distributed nature. The design space for the solution of this problem also opens the possibility for the use of a static or dynamic approach. All these aspects are further discussed in Chapter 3. In the end, critical challenges have to be addressed by system designers when developing resource management policies for their MPSoCs. Smooth switching between mappings has to be enforced, so that users do not experience any noticeable delays when a transition between any two consecutive mapping scenarios takes place. Typically, in an efficient resource management, the higher the mode switching overhead to reconfigure the system and adapt it to the new working conditions, the higher the efficiency of the system in the new execution scenario. Finally, since the number of use-cases is exponential in the number of applications in the system, finding a design-time mapping of applications on the processors for each use-case requires computational-efficient optimization engines.

Resources are not limited to processing elements. A scalable management of communication resources is also required. Approaches for the management of NoC resources (PASTRNAK, 2006) may provide better scalability and prevent single points of failures. In decentralized concepts, the resource allocation decisions are taken within the granularity of a region in a distributed and asynchronous manner. Similarly, to manage memory resources may pay dividends such as reduced memory latency. This situation is further explained in Chapter 4.

1.3 Clustering

The first ideas and implementations of computer clusters date back to the 1960's. The general intuition was to increase the execution time of applications by connecting several computers. The concept of parallel programming emerged in the same environment, when Gene Amdahl created the Amdahl's Law (AMDAHL, 1967). At the same time, the main component that made possible the creation of these early clusters was the development of faster computer networks. Fifty years later, these computers and networks are now embedded inside single chips (MPSoCs) and the notion of clustering has been brought back in an environment that already has reused many other old concepts. In the literature, several examples of cluster on MPSoCs can be found. Typically, these clusters are conceived by physically aggregating the resources by means of the memory subsystem architecture or the interconnection mechanism itself. In Chapter 2 these works are explained in more detail.

In the case of clusters based on interconnection mechanism, the intuition is that, when many tasks run in parallel on an MPSoC, one level of interconnect is generally insufficient to cope with the contention between various requests and responses issued by such large number of parallel tasks. A hierarchical architecture regroups processors and coprocessors around local interconnects. In this case, such group is labeled a cluster. Normally, these hierarchical architectures present themselves in the format of heterogeneous interconnections creating islands with distinct communication delays.

Other clusters are formed by simply connecting groups of cores to different physical memory modules, and these groups in turn are connected amongst themselves. In this kind of clustered architecture, memory access times differ significantly, depending on whether a processor accesses a memory bank local to its own cluster, or situated on another cluster. This phenomenon, called Non Uniform Memory Access (NUMA), makes it practically impossible to predict the order of arrival or latency of tasks executing in any cluster.

1.4 Task Mapping and Scheduling

Task mapping is defined as finding a suitable distribution of application tasks among processor elements in such way that a set of pre-established requirements are fulfilled. These requirements may be faster execution, energy saving, or congestion reduction, just to name a few. Mapping decisions can considerably affect the overall system performance. Regarding the moment in which it is defined, task mapping can be classified as static or dynamic. In the first one, the placement of tasks is decided at design time. Multiple works propose static mapping techniques (HU, 2005; MANOLACHE, 2005; ORSILA, 2007; MURALI, 2006). However, these approaches are not adequate for environments where the workloads are dynamic, due to their

complexity and therefore longer execution time. Dynamic task mapping, on the other hand, is capable of defining each task placement at runtime.

Generally, the problem of task mapping and scheduling can be difficult since several use cases have to be considered simultaneously. For each case, a combination of applications must be considered to ensure that deadlines, energy budgets or performance guarantees are met. In addition, these applications may be using different models of computation (MARWEDEL, 2003), which can be an important factor when considering task migration costs. For instance, Asynchronous Message Passage models need to guarantee that messages sent before the migration are delivered even when tasks move to another processor. Also, distinct models may be more suitable for different memory architectures, which makes data migration an important factor as well.

Generated mappings have to include solutions to the scheduling, resource allocation and resource assignment problems. The type and number of resources (processors, buses, etc.) may either be already specified in the initial input by the designer or they may be generated during the design steps. If this information is not provided by the designer, it has to be generated by solving the resource allocation problem. Resource assignment is supposed to map computations and communication to hardware resources. Solutions to the scheduling problem provide a mapping from operations to the times during which these operations are performed. Multiprocessor scheduling approaches will sometimes also decide which hardware resources to be used, i.e. the resource assignment problem is included in the scheduling problem.

1.5 Thesis Goals

Given the ever increasing number of components in a NoC-based System-on-Chip, it is imperative to manage efficiently its resources. This thesis has the goal of proposing ways to deal with this situation at processor level and memory level. In addition, another study on resource-aware fault-tolerance mechanism is presented as a consequence of using cluster-based approaches.

At the processor level, the idea is to aggregate processing elements and give more resources to applications that need them most. As shown in (XU, 2000), applications change their degree of parallelism and, therefore, the number of processing elements needed to execute their tasks can increase or decrease at runtime. Hence, the need for a dynamic approach is more suitable not only because new applications can arrive at the system. A dynamic approach can also improve the performance of a system by re-distributing processing resources that are idle or currently have a smaller workload compared to the rest of the resources.

Based on that, we introduce the Processor Clustering mechanism. In this mechanism, we consider an MPSoC environment that is able to support distinct parallel programming models. In order to do that, a hardware mechanism that enables/disables the cache controller connection in local memories is presented. That way, a management entity (a scheduler, for instance) can decide when to change this memory subsystem in order to execute applications modeled as Message Passing (which are more suitable for distributed memories) or Shared Variables (which needs shared memories).

In this scenario, the Processor Clustering mechanism aims at managing the different applications in the system by efficiently migrating the tasks as more processor resources (cores) become available. The idea is to use a work-stealing approach and devise a set

of metrics to use as rules for migration. A set of experiments are performed considering the execution of several tasks and combinations of instances of these applications implemented with distinct parallel programming models. Results show that the dynamic redistribution of these resources can bring improvements on both performance and energy aspects of the system. Also, aspects such as task placement and distance and parallel programming model can heavily impact these improvements.

Regarding the memory subsystem, the memory resources are even more abundant than processing elements. As a classical and perennial bottleneck of computer systems, the memory resources must be dealt with the outmost efficiency in order to provide decent performance. In this work we propose the use of an analogous approach used at the processor level by means of resource awareness and a cluster-based method. In this case, the idea is to aggregate the memory modules in clusters similar to the ones at the processor level. However, there is no guarantee that the aggregation (clusterization) used at the processor level would be the most fair distribution of resources at the memory level. It is well known that some applications, at times, may have more memory needs than computational needs. Thus, the most efficient clustering, at the two levels, may be quite different. For this Memory Clustering proposal, we used the information of external memory accesses in order to find out which application needs more memory. The key idea is that external memory accesses are very expensive and, therefore, can be diminished by giving more cache memory modules to certain applications. Evidently, the cache memory modules given to this application were not idle since, differently from processor elements, all cache memory modules available in the system can be used by any application at any time.

In this work, we explore the aspects of this memory resource management approach. First, we introduce a mechanism with which we are able to assign each L2 memory bank in the system to be used exclusively by a given application (and its respective tasks). This mechanism is based on dynamically changeable association tables present in every L1 cache, which indicate the L2 cache banks, their memory address range, and their NoC address. Second, we must establish how to redistribute the L2 cache banks among the applications. As mentioned earlier, we use as metric the amount of external memory requests. Applications with higher external memory requests must be awarded with a higher amount of L2 banks. We compare two approaches in this case: a pre-established approach in which the amount of L2 banks per applications is known before execution based on application profiling; and a runtime approach that uses statistical analysis (at different checkpoints during execution) in order to come to the conclusion of how many L2 memory banks each application needs. Finally, we present an approach to establish which L2 memory banks should be taken/given by the applications. The main idea of this approach is to keep memory banks assigned to a given application close to each other and avoid fragmentation. Experimental results are presented considering a certain initial allocation of tasks (and a redistribution of memory resources, at some point) and no task migration. These results show that the overall system improvement can be reached at the expense of the performance of some applications. The key observation is that some applications have more impact on the overall execution than others, and, consequently, reducing the memory latency of these applications can lead to performance gains.

At this point, the contributions of this work at processor level and at the memory level provide means to manage resources in the system in order to improve performance and save energy. These goals are achieved by relocating resources to applications that

need them the most at the expense of smaller applications (which need fewer resources). In this scenario other features can be provided in order to improve the system.

- At the programmer level, one can provide APIs to improve thread communication considering this resource management (explicitly indicating certain priorities to certain applications).
- At the interconnection level, a cluster-based router can be used in order to isolate traffic in the system in an attempt to minimize the interference of one application on another, and bring a notion of justice in terms of communication latency.
- Virtualization of applications can also be supported by the hardware due to the isolation of clusters. This can minimize the overhead of a hypervisor by taken from it the responsibility of management. Simpler hypervisors can be more efficient and easier to maintain.
- Data security has become a frequent field of investigation on systems-on-chip. Over the last decades some concerns regarding data exposure, user intrusion, and system integrity have been investigated by researchers and developers in the field of Operating Systems. The use of cluster-based system can offer support at the hardware level to deal with such concerns. Resource isolation, data protection and traffic control are some features offered by systems implementing resource-aware clustering.

This is only to name a few aspects in the system that can take advantage of a cluster-based resource-aware approach. However, a different aspect is explored as part of this thesis' contributions: fault-tolerance and reliability.

In an MPSoC, it is also important to efficiently reallocate resources in a case of failure. Nowadays it is not uncommon to have processors, memories and even interconnection mechanisms falling ill to a series of faulty sources. In this work, we try to use the concepts of resource-awareness and clustering organizations to improve the yield of a SoC memory subsystem by using the network-on-chip as resource manager. The intuition here is that some memory faults can be detected at design time and therefore a fault map can be generated with the information of which memory sections are not working. These fault tolerant methods based on fault maps are very suitable for simple resource-aware approaches. Given the knowledge of where the healthy memory areas might be, some resource manager entity can easily reduce the number of accesses to faulty resources. By means of a bookkeeping-based scheme, we reuse partially faulty cache blocks to compose healthy ones. This mechanism is composed of three steps: issuing memory request for the selected cache banks; receiving these cache blocks; and matching them to come up with a healthier one. We propose different policies that explore distinct points in the system to perform each one of these steps. Results show that these policies present different results depending on the application demands. Also we show that the use of a fault-map based mechanism does not generate a large overhead.

The main idea to implement the Reliability Clustering approach introduced in this work is to insert this aforementioned knowledge into the network-on-chip routers. Therefore, memory requests can be redirected in order to increase the number of healthy memory accesses, thus avoiding the external memory access. We perform a design space exploration considering different redundant memories in the system and placing them using several cluster-based patterns. Also, we consider a certain range of fault rates and memory redundancy to explore. The results presented here point out to the fact

that there is no clear winner regarding memory redundancy placement. However, aspects like the average distance between these memories in the system and the amount of redundancy per node are decisive in the overall performance. The specifics of this approach are given in Chapter 5.

Overall, these three mechanisms in three different layers have in common the goal of managing efficiently resources in the system by means of cluster-based methods. To the extent of our knowledge, these are original approaches at each own level.

1.6 Text Organization

The remaining of this proposal is organized as follows. In Chapter 2 the most relevant works that are related to this thesis proposal are presented. Chapter 3 presents the concept of Processor Clustering used in this work. The architecture used to explore the concepts of our approach is also presented, as well as some results based on different policies and use-case scenarios. Chapter 4 presents the Memory Clustering approach, explaining the shared memory architecture used to develop the ideas involved in this proposal. Some results are presented considering design-time and runtime approaches. Chapter 5 describes the Reliability Clustering with its own architecture and memory subsystem. Also a fault tolerance method for last level caches used in the Reliability Clustering is presented. Some results are presented considering the different clustering formats of reliable memories. Finally, in Chapter 6 conclusions are drawn and further studies to complement the ones presented in this document are proposed. These studies include deeper experiments on Memory Clustering as well as a solution combining Processor and Memory Clustering policies.

2 RELATED WORK

As this work deals with cluster-based resource-aware solutions for MPSoCs, the papers discussed in this chapter are divided in three categories: Cluster-based organizations, Resource Management and Task Mapping. Typically, the literature presents several papers that deal with these three aspects. However, some papers have a more intense focus in some of these three key points. This chapter presents these papers giving detailed descriptions of the ones considered more important.

2.1 Cluster-based Multi-cores

Some works present key contributions regarding the physical or logical distribution or resources in cluster structures.

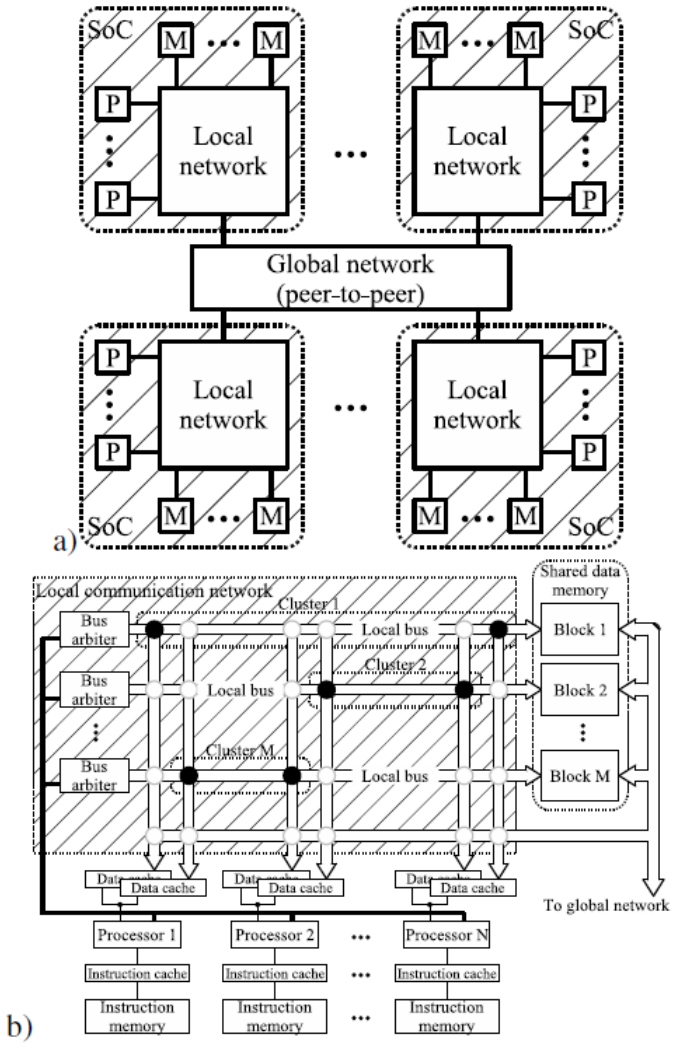
Masko (2008) presents an algorithm for scheduling parallel tasks in a parallel architecture featuring multiple dynamic SMP (Symmetric Multiprocessor) clusters. Each SoC module is a cluster formed by processors that are connected to the same memory.

Figure 2.1a presents this architecture of SoC modules interconnected by a global network. Each one of these SoCs work as a physical cluster. As shown in Figure 2.1Source: Masko (2008, p. 100).

b, the SoCs contain processors and memory modules connected through a local data networks. The shared data memory is subdivided in blocks, and bus arbiters connect processors to them. Logically, each one of these blocks and the respective processors connected to them can be considered a cluster itself. Since this shared data memory is multi-port, one processor can be connected to more than one cluster, which, as the authors state, can improve data communication efficiency.

In this work, the communication control inside clusters is performed by means of data pre-fetch, reads on-the fly, and processor switching between clusters. Reads on-the-fly are performed by capturing data in a processor data cache while the data are being transmitted through the local network (in order to be written in the memory). Also, the processors can switch between clusters as an arbiter decision. This can help data communication since the processor switching between clusters will bring useful data in its cache.

Figure 2.1. Overall System Architecture and structure of a single SoC.



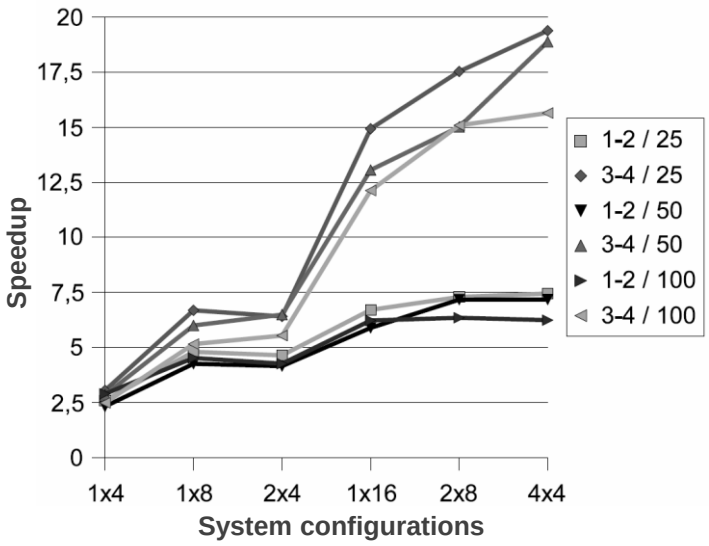
Source: Masko (2008, p. 100).

The algorithm schedules an initial macro dataflow program graph for such architecture with a given number of SoC modules, considering a fixed size for a module. First, it distributes program graph nodes among processors, assuming no distribution between SoC modules. Then it transforms and schedules computations and communications using a new communication paradigm that is based on the fact that each processor has multi-port data caches that are also connected to arbiters from different clusters. Finally, using a genetic algorithm, it divides the whole set of processors into subsets of a given size, which are then mapped to separate SoC modules.

As results, the authors present speedup and parallel efficiency of the algorithm as shown in Figure 2.2. These results are from experiments with program graphs in 6 configurations, where the number of clusters may vary as well as the number of cores inside each one of these clusters. The X-axis shows these configurations as "*number_of_clusters* X *number_of_cores_per_cluster*". Each line in the graphs represent a type of program graph by varying the depth of graph nodes (1-2 or 3-4 subgraphs) and the weight of communication edges (25, 50 or 100).

The speedup shown in Figure 2.2 suggests that more dense graphs with heavier communication demands present higher speedups than sparse graphs. Authors state that the reason is the use of reads-on-the-fly (more communication benefits this approach) and the behavior of data caches.

Figure 2.2. Speedup gains.



Source: Masko (2008, p. 105).

In these experiments, the program graphs were executed in a homogeneous system. This means that every processor used the same data cache size. For configurations with a total number of 4 or 8 processors, some processors execute more than one thread. More threads being executed in the same processor require more data caches. Therefore, the shared memory access in this case would increase and, consequently, the performance takes a hit, as seen in Figure 2.2.

In the end, the paper is a proposal of a dynamic clustering based on communication through shared memory. However, it does not present cycle accurate performance results and does not discuss the impact of a segmented network formed by global and local networks. In addition, no energy results are presented and there is no evidence regarding the impact of such multi-port memory architecture.

The work presented in (MANDELLI, 2012) tackles the resource management problem by means of a distributed resource manager in a clustered NoC-based MPSoC. This cluster-based architecture is divided in N clusters of same size, which are defined at design time. This means that physical constraints (such as the interconnection) aggregate these processor elements in regions. At runtime it may be decided that if an application requires more resources than the ones available in that cluster there will be a request sent to another cluster to request additional resources. The authors emphasize that this is a reasonable approach since:

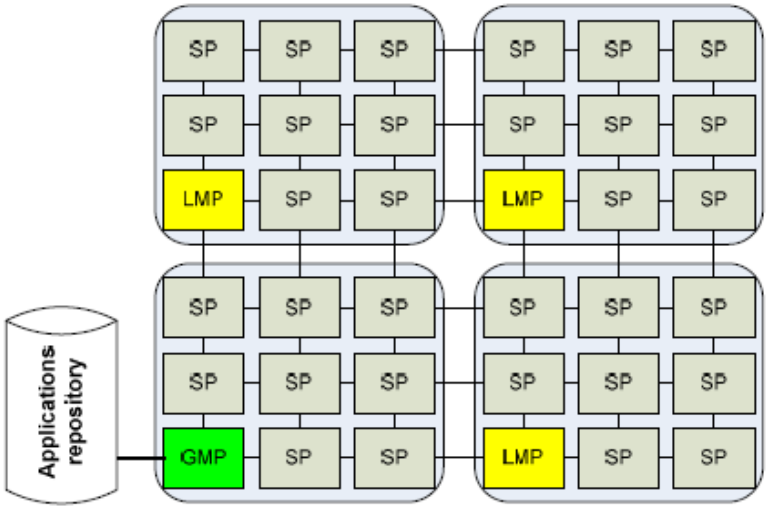
1. The number of PE's dedicated to manage the resources is limited to the number of clusters (more processors per cluster means less clusters in the system). To use one manager per application would create an

uncertain scenario in terms of scalability since the number of applications may vary.

2. The cluster-based approach can reduce the NoC distance among tasks from the same application. This would alleviate the NoC traffic.

Figure 2.3 presents the 6x6 MPSoC architecture proposed by the authors. This architecture has four 3x3 clusters. This environment has three types of processing elements: the Global Master PE (GMP); Local Master PEs (LMP); Slave PEs (SP). Master PEs are solely responsible for resource management functions. The Local Master, in particular, has the function of controlling its cluster and performing task mapping, migration and monitoring, as well as deadline verification and communication with other masters. The Global Master has the same functions as the LMP (since the GMP is also located in a cluster that needs management of its own). But, in addition, the GMP is responsible for the overall system management. Some functions are similar to the ones associated to the LMP but at a higher instance, like the application mapping among the clusters (not the tasks inside each cluster), control of available resources in each cluster, control of the external memory access, and management of the admission control (admittance of new application into the system). The Slave PEs are only responsible for actual execution of the application.

Figure 2.3. Clustered NoC-based MPSoC.



Source: Mandelli (2012, p. 545).

At system boot, the GMP must also initialize the clusters by letting each LMP know which region it must control. In turn, the LMPs broadcast the information to all SPs in that region informing them that it will be the Local Master. This initialization provides reasonable adaptability for the system, thus creating the possibility of re-clustering.

The admission of new applications into the system is performed at two levels. First, the GMP must select which cluster should execute the applications. This decision is based on the number of processors available in the cluster and the number of tasks a given PE can execute simultaneously. This simple calculation gives the total number of tasks that a cluster can execute. If the application fits in a cluster (using a best-fit heuristic), the GMP reads the application description from the repository (shown in Figure 2.3) and sends it to the LMP from the selected cluster. This description contains

the application ID and a list of information about every task that belongs to that application:

- Whether or not it is the initial task;
- The initial program memory address;
- The size of the object code; and
- A list of the other tasks that communicate with this task and the communication volume;

Upon receiving the application description, the LMP starts the task mapping onto the SPs. The SP with the highest number of available resources is chosen to execute the initial task. This has the goal of keeping the tasks as closer as possible in the system in order to reduce communication latency and energy consumption. As for the other tasks, the LMP sends a packet to the GMP with allocation requests giving the information of the PE location in the NoC, the application ID, the initial memory position and the size of the task. The GMP programs the DMA module to send this data from the repository to the selected PE. This is performed for all PEs involved in this cluster, which means that this allocation using the GMP is sequential for every PE in the cluster and during this process no other cluster can allocate tasks.

The results presented in this paper were obtained from a cycle-accurate NoC-based MPSoC named HeMPS (CARARA, 2009). In this architecture, there are 9 clusters (in a 3x3 grid) and each cluster has 8 SPs which can execute 2 simultaneous tasks. That leaves each cluster with the possibility of executing up to 16 tasks. Several experiments scenarios were used combining different MPSoC sizes (from 6x6 cores up to 12x12), three applications (MPEG decoder, *multispec* image analysis (TAN, 2008) and a synthetic application), and configurations of both a centralized approach (using only a GMP core and no LMP cores) and a distributed approach, as shown in Figure 2.3. Several instances of the same applications were used in the same hardware scenario (which makes the number of running tasks vary from 48 to 224 depending on the MPSoC size). Table 2.1 presents these scenarios, and the last two columns represent the system usage for the distributed and centralized approaches.

Table 2.1. Evaluated scenarios.

	MPSoC Size	Benchmark – # of tasks	App_{CL}	Total number of tasks	SU_{dist}	SU_{centr}
A	6x6 – 4 Clusters – 32SPs (distributed) – 35 SPs (centralized)	Synthetic – 6	2	48	75%	69%
B		MPEG – 5	3	60	94%	86%
C		Multispec – 14	1	56	88%	80%
D	9x9 – 9 Clusters – 72SPs (distributed) – 80 SPs (centralized)	Synthetic – 6	2	108	75%	68%
E		MPEG – 5	3	135	94%	84%
F		Multispec – 14	1	126	88%	79%
G	12x12 – 16 Clusters – 128 SPs (distributed) – 143 SPs (centralized)	Synthetic – 6	2	192	75%	67%
H		MPEG – 5	3	240	94%	84%
I		Multispec – 14	1	224	88%	78%

Source: Mandelli (2012, p. 547).

Table 2.2 presents the execution time reduction of the distributed approach for each scenario using the centralized management as baseline. The distributed approach presents better results as the MPSoC size increases, which was expected since larger numbers of SPs create larger NoC distances between cores and the GMP. Furthermore,

the use of LMP tends to minimize the execution time of task allocation algorithms, since the number of SPs to deal with is reduced.

Table 2.2. Distributed vs. Centralized mapping comparison.

	MPSoC Size	Benchmark	Execution time reduction (compared to centralized mapping)
A	6x6	Synthetic	-15% (increase of time)
B		MPEG	28%
C		Multispec	34%
D	9x9	Synthetic	50%
E		MPEG	63%
F		Multispec	54%
G	12x12	Synthetic	79%
H		MPEG	86%
I		Multispec	85%

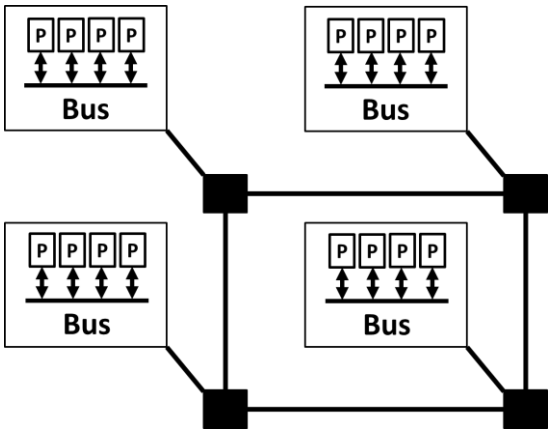
Source: Mandelli (2008, p. 547).

This paper presents an interesting solution for resource management and a comparison between centralized and distributed mapping. However, there are no energy results comparing the two approaches even though energy consumption seems to be one of the motivations. In addition, the overhead of using a GMP as a centralized component needed in the decision making of local task mapping seems to limit the distributed method gains. Finally, the use of dedicated cores to manage the resources is only briefly discussed. The authors claim that, in a 16x16 MPSoC using 4x4 clusters, this overhead becomes 6.25%. This is true, but the size of the clusters depends on the number of applications (more applications lead to more clusters) and, as for each cluster, there must be a LMP core. With that, the overhead calculation should be based on the number of applications (more applications, more LMP cores, thus more overhead).

Several other papers also present a cluster-based approach, as follows. Al Faruque et al. (2008) present an environment where clusters are controlled by an management component with the goal of mapping tasks in these clusters. The system is controlled by distributed global agents that keep track of allocation information on all clusters. These agents also decide eventual re-allocation at runtime with the possibility of rearranging clusters. Another paper (CUI, 2012), based on the works of Al Faruque, proposes a cluster-based approach with the goal of reducing communication traffic between global managers and cluster agents. The approach tries to minimize the task mapping information on local agents and proposes a new clustering arrangement.

Zhang et al (2012) present a communication performance study on a configurable Cluster-based MPSoC architecture. The architecture is composed of an NoC infrastructure that interconnects shared-bus based clusters, as presented in Figure 2.4. Experimental results compare the use of this hybrid interconnection versus the use of a homogeneous mesh-based NoC interconnection. The authors show through the experiments that assuming that the number of processing cores per cluster is reasonable for the set of applications to be executed in the system, the hybrid infrastructure presents better communication performance. However, these results depend heavily on this cluster sizing meeting the requirements of applications and the memory locality favoring local data communications only.

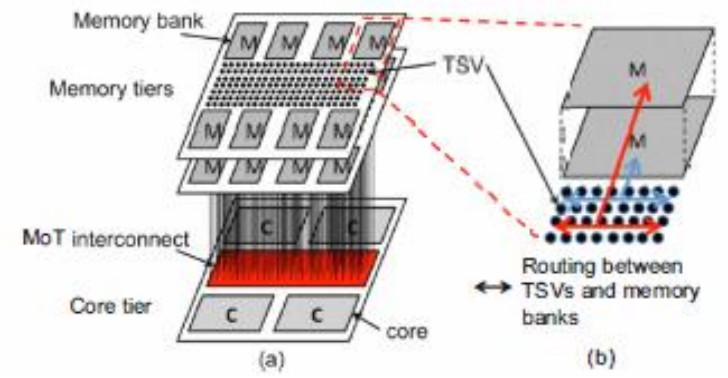
Figure 2.4. Cluster-based MPSoC with hybrid interconnection.



Source: Zhang (2012, p. 2).

Other cluster-based architectures can come from the advances on physical design. Kang et al (2012) present a Mesh-of-Trees network by means of 3-D integration. The goals are to achieve high-throughput and low-latency using TSV (Through-Silicon Via) connections. These TSV channels connect the cores to a 3D stacked L2 memory structure using dedicated sequential routers. This infrastructure is presented in Figure 2.5**Error! Reference source not found.**. Results show significant improvement in the overall system performance due to faster memory access. However, no energy results are presented.

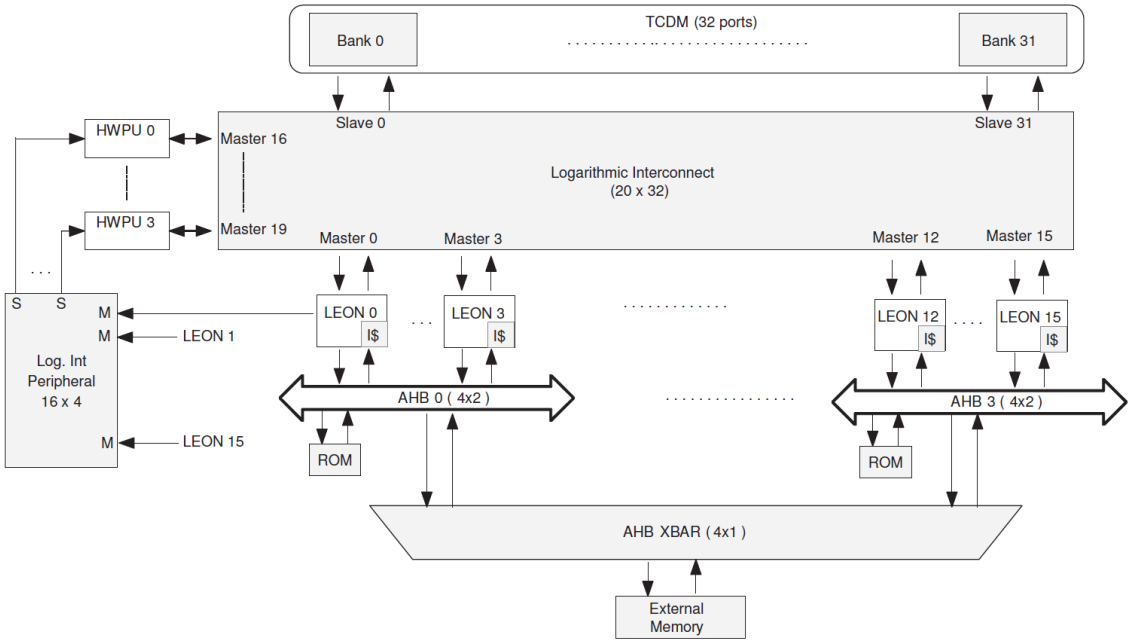
Figure 2.5. 3D cluster using a Mesh-of-Trees architecture and TSV channels.



Source: Kang (2012, p. 8).

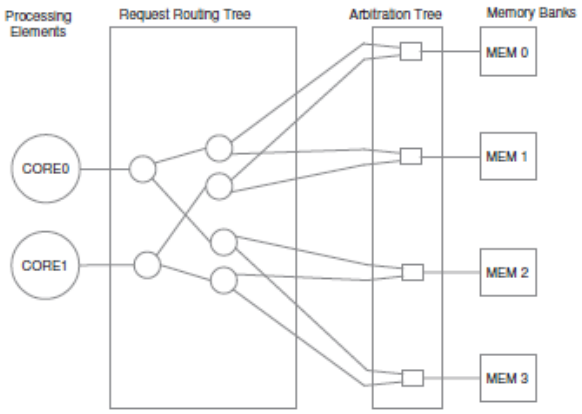
In (DEHYADEGARI, 2012) a tightly-coupled multi-core cluster architecture named Mega-leon is proposed. This architecture presents a set of homogeneous parallel cores (Leon cores (GAISLER, 2001)) coupled with hardware accelerators, as shown in Figure 2.6. The communication between these hardware processing units and CPUs is performed through the shared memory using a zero-copy mechanism. The interconnection between cores, hardware unit and the memory subsystem is established through a Mesh-of-Trees in order to improve memory latency, as presented in Figure 2.7. In this architecture, the data inconsistencies do not exist since local copies of data are not allowed. Also, the memory distribution is very important since this architecture uses physical clustering by means of dedicated interconnections. The authors present several experiments on different data mapping approaches.

Figure 2.6. Mega-leon Architecture.



Source: Dehyadegari (2012, p. 97).

Figure 2.7. Mesh-of-Trees 2x4 example.

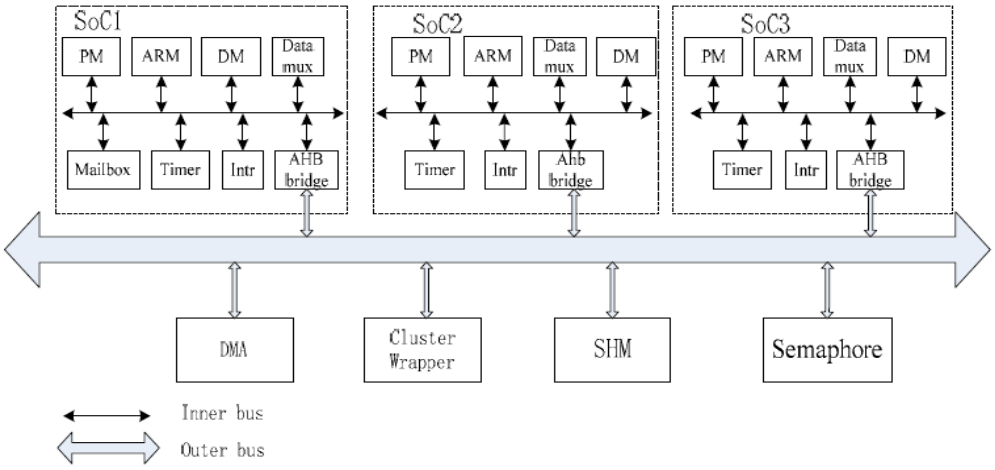


Source: Dehyadegari (2012, p. 97).

The authors in (ZHANG, 2007) propose a cluster-based hybrid reconfigurable and programmable architecture. In this platform, each cluster is a heterogeneous structure composed by reconfigurable processors and RISC-based cores running an RTOS to control the sequence of reconfigurations as well as multiple memory modules. The cluster intercommunication is achieved by using a serial or semi-parallel link. The reconfiguration can be performed at two levels. First, at global level, the reconfiguration occurs at the communication between clusters and elements of the cluster. At a local reconfiguration level, the architecture provides the choice of executing tasks in software or hardware. These hardware versions are mapped onto a reconfigurable processor. Different types of reconfiguration are allowed depending on the partial use, full use of a reconfigurable processor, and use of multiple processors to execute a task.

In (JIN, 2010), the authors present a cluster-based MPSoC using a hierarchical bus architecture implemented on FPGA. In order to meet real-time constraints, the authors propose a synchronization improvement based on separated control path and data path. The architecture, as presented in Figure 2.8, is another example of physically arranged clustered MPSoC. However, in this case, the hybrid interconnection mechanism uses an NoC inside a cluster and a bus to connect the clusters. Here, the SoCs act as the clusters connected to a bus, and the slave peripherals include a DMA controller, a Shared Memory (SHM), a semaphore used for synchronization, and a Cluster-Wrapper that maps the external bus protocol to the NoC protocol (internal to the SoC).

Figure 2.8. Clustered MPSoC.



Source: Jin (2010, p. 72).

Even though a number of cluster-based architectures are presented here, some of them present clusters based on physical restrictions. This can create a communication latency problem if the number of tasks in an application does not fit the physical distribution of resources. As presented in the following chapters, this thesis proposal presents virtual cluster-based approaches. This means that there are no physical restrictions to aggregate the resources into clusters. All clusters are logically established based on specific metrics in order to take advantage of hardware and software system features.

2.2 Resource Management

The increase of on-chip resources according to Moore's law has led to a management problem since these resources should be used efficiently. Some recent works have been dealing with this problem by using physical or logical components identified as resource managers. This section presents some of these papers.

Recent works propose a paradigm of multicore programming called Invasive Computing (TEICH, 2011). The proposal is to take advantage of a large many-core processor in the best possible way. The idea leverages on malleable applications where the degree of parallelism can change on the fly. At the application level, the programmer uses functions that trigger the invasion process. When an application sends an invade request, a distributed resource manager evaluates the request based on the amount of resources required and the estimated speed-up per core. This section explains in more details the essence of this proposal.

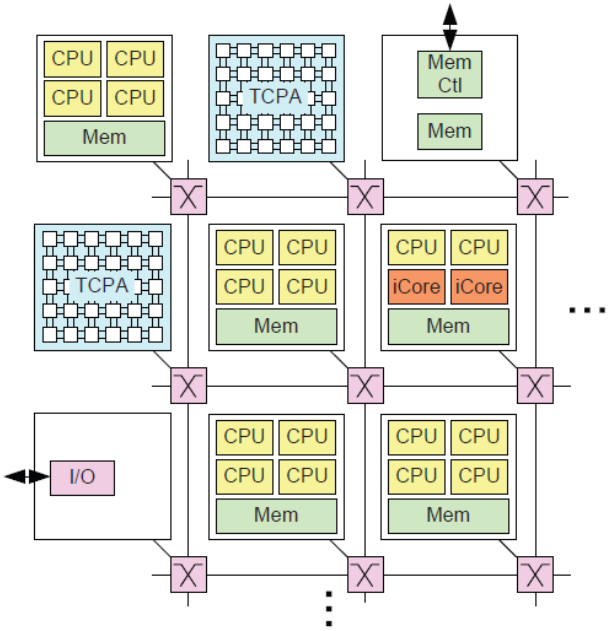
Invasive Programming is defined in (UREÑA, 2012) as the capability of a parallel application currently executing to request and temporarily claim processor, communication and memory resources in the vicinity of its environment inside the chip, to execute in parallel the given application using these claimed resources, and to be capable of eventually releasing these resources.

Figure 2.9 presents the invasive multi-processor architecture presented in the paper, which includes loosely-coupled processors and tightly-coupled co-processor arrays (TCPA). These processors are aggregated in physical nodes, and local memories are present in them.

Figure 2.10 shows an example of how the concept of invasion works at the level of loop programs in a tightly-coupled co-processor array (TCPA) in the environment of the heterogeneous architecture presented in Figure 2.9. Here, A1 and A2 are programs executing in parallel and a third program (labeled A3) is beginning to run on a single processor.

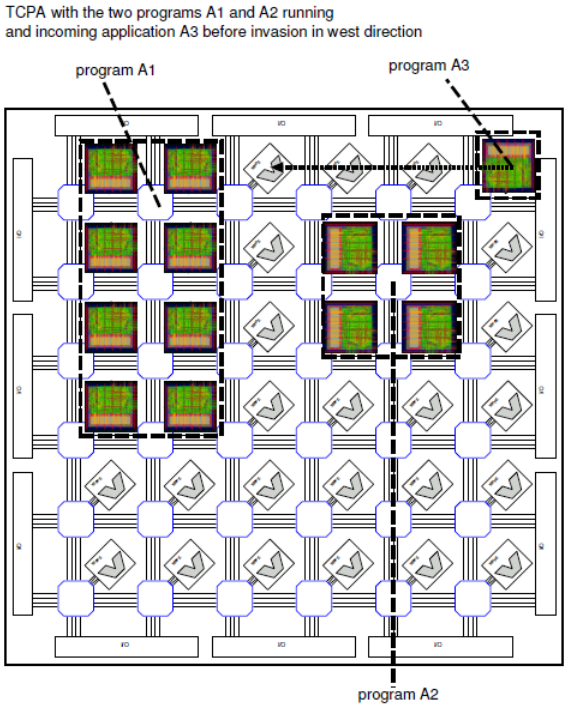
In the invasion phase, application A3 requests all of its neighbor processors on the west to join their resources (in this case, memory and processing elements) in order to execute A3 in parallel. This resource request goes on until the point where the neighbor resources are already allocated to some other application or the number of resources matches the limit of the application's degree of parallelism. After that reservation, the invasive program starts copying itself to the new claimed resources and eventually starts executing on all of them in parallel, as shown in Figure 2.11.

Figure 2.9. Invasive multi-processor architecture.



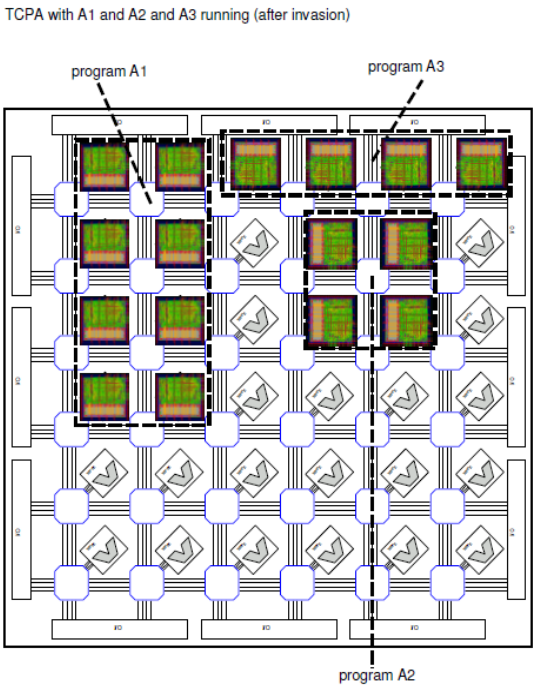
Source: Teich (2011, p. 247).

Figure 2.10. TCPA scenario before invasion.



Source: Teich (2011, p. 248).

Figure 2.11. TCPA scenario after invasion.

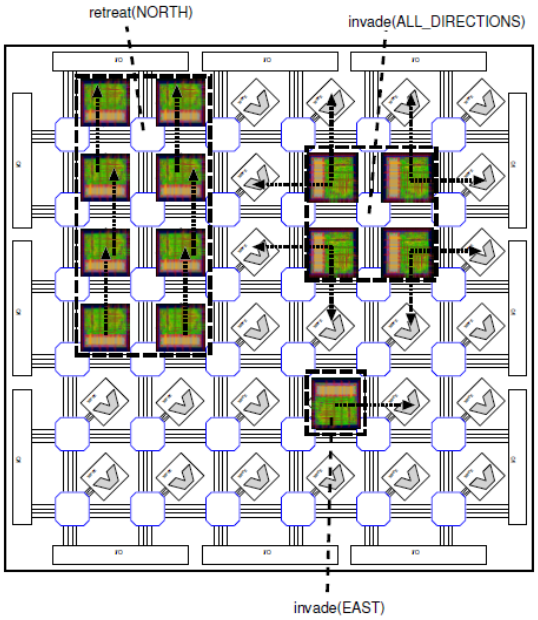


Source: Teich (2011, p. 249).

If the program finishes running or does not need the resources anymore (since the degree of parallelism may change throughout its execution), the program can initiate a

retreat operation in order to release these resources. Figure 2.12 illustrates an example of the retreat operation.

Figure 2.12. Uni- and Multi-directional invasions and Retreat operation.



Source: Teich (2011, p. 250).

Three basic operations are described by the authors as being essential to support invasive programming: Invade, Infect and Retreat. The authors also state that these operations can be implemented with little overhead on reconfigurable MPSoC architectures like WPPA (Weakly Programmable Processor Arrays) (KISSLER, 2006) or AMURHA (Adaptive Multi-grained Hardware Architecture) (HANNIG, 2006), since these architectures provide support for interconnection reconfiguration in subdomains. In (HANNIG, 2006) the authors state that a program of size L can be copied in *O*(L) clock cycles. However, there is no discussion about the overhead on regular MPSoCs. This phase where the program is copied to resources is called Infect.

The concept of Invasion can be implemented using an agent-based approach that has the job of distributing the program threads over processor resources. At this point, the idea of invasive computing can be applied using dynamic load-balancing techniques such as diffusion-based load balancing (RABANI, 1998) or even centralized algorithms based on global prioritization using distributed priority queues (SANDERS, 1998).

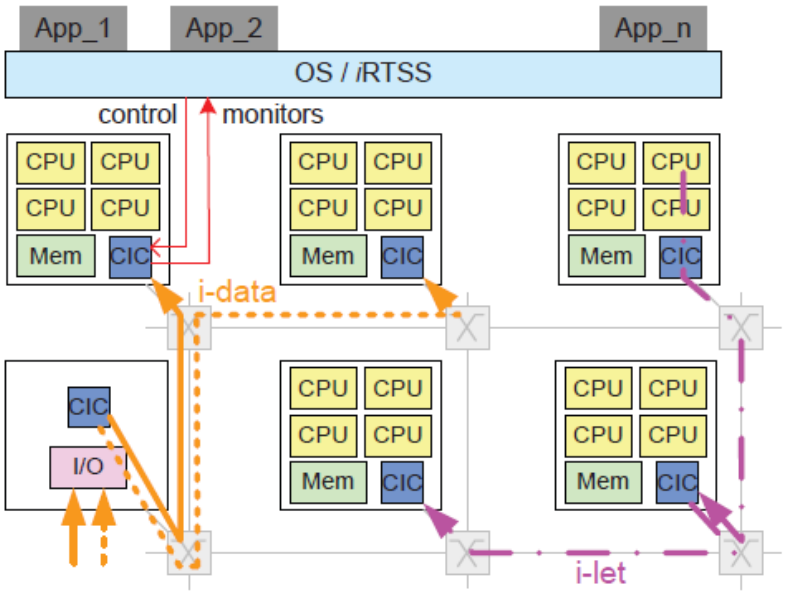
The examples shown so far presented a tile-based architecture (as the one presented in Figure 2.5). However, invasive computing can also be used in loosely-coupled multicore architectures. The cores and local or distributed memory modules can be clustered in tiles connected by an NoC. Typically, an operating system runs a distributed or multi-instance way on several cores.

In order to demonstrate a hardware-based environment suitable for the concepts of invasive computing, (BECKER, 2012) presents an infrastructure support named Dynamic Many-Core i-let1 Controlers (CIC). The main goal of this infrastructure is to reduce the overheads associated to the invasion/infection phases. The invasion process

requires a runtime support service of constant monitoring information on the status of the hardware platform via the CICs. These infrastructures are configured to forward the processing requests that correspond to the infection of invaded cores. Another function of CICs is to aid in the decision making in order to optimize the overall system performance by taking into account communication resources, reliability, or the temperature of the die.

CICs keep track of processing requests to cores under the control of the runtime environment (iRTSS – Invasive Runtime Support System). These requests may be generated when an application wants to spawn additional threads (creating the potential need for more processing and memory resources), as presented on the right side of Figure 2.13. Another situation that creates these requests is when data arrives from external interfaces (as shown on the left side of Figure 2.13).

Figure 2.13. Invasive computing on loosely-coupled MPSoCs.



Source: Teich (2011, p. 251).

In the case of an application that needs to create more threads, an i-let is created and sent to a target tile. An i-let is an invasion structure that requests resources in order to start the infection phase. A CIC at the target tile will negotiate its resources based on the rules given by the iRTSS considering the workload on the tile. In addition, this CIC will distribute the i-let to other CICs of other tiles in the neighborhood, as presented in Figure 2.13.

In the case of external data arriving at the system from the I/O subsystem (e.g.: more data to be processed by an application currently being executed or another application coming to be executed), the CIC itself (located at the I/O node) can start an invasion on further CPU clusters. CIC rules, established to maintain some quality requirement, would be updated and, in consequence, excess requests (invasive data or i-data) would be distributed to the newly invaded resources in order to deal with the new input workload of the application.

Shabbir (2011) proposes two versions of resource managers, stating that they are scalable regarding the number of applications and processors. The paper discusses the strategies around the development of resource managers by comparing centralized and distributed approaches. Two distributed resource managers are proposed and compared based on their throughput capabilities when dealing with multiple applications that can be added at runtime. The authors present a scenario where real-time applications are one of the focuses of the study on resource managers. In this case, it is very important that all tasks meet their deadlines.

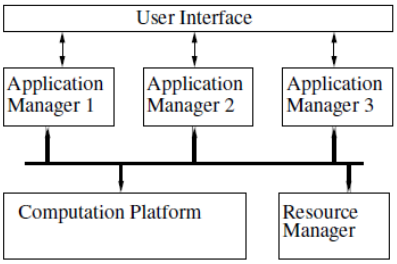
In order to provide a resource manager (RM) that deals with the addition of multiple applications at runtime, some admission control strategy must be devised. A central admission controller is presented in the system. This controller calculates and assigns credits for a new application and distributes them to the processors. The arbiters at the processors locally enforce these credits in a way that the throughput constraints of the applications are satisfied. The authors establish a monitoring period for the resource managers. This monitoring period is defined as the period of time in which the central resource manager must receive the execution information from each application and act upon it. Based on this information, the RM can define the current period (of execution) of the applications and compare to a desired period in order to meet a certain execution requirement. Based on that, the RM can enable or disable the application. All these actions, for all applications, must occur in this monitoring period

Credits are calculated as follows. Processors in the system have a certain interval of time in which all applications must be executed (replenishment interval). Considering that each task has its own period, the central admission controller must find the load created by a hypothetical new application on the system. The load can be calculated based on the repetition (periodicity) of the application, its execution time and the desired throughput (since the throughput is the main feature aimed by the authors). In order to be accepted in the system, this new application must not create a total processing load that is higher than the replenishment interval of the processor (potentially, this test is necessary for all processors in the system). The credits are a representation of this processing load.

One of the proposals in this paper is a credit-based resource manager. After the admission controller sends the corresponding credits to each processor (depending on the mapping), each processor kernel loads these credits into counters. Each task is repeated as many times as specified in its counter in one replenishment interval. At the end of this interval the counters are reloaded with the original values and the process restarts. If the task is not ready to be executed, the processor is given another task to be executed in order to use the resources more efficiently.

The other distributed RM proposed in the paper is called a Rate-based resource manager. The motivation of such RM comes from the fact that credit-based RMs are very dependent on the size of the replenishment interval, which is not true for the Rate-based one. The admission controller used is the same and calculates the credits for the Rate-based RM. A local arbiter receives these credits and executes the task in such a fashion that the task having the smallest achieved-to-desired execution ratio is the one with highest priority. The arbiter contains information on the desired rates (based on credits), the achieved rates as well as the achieved-to-desired rate for each task. In the end, the key idea here is to give more time slots for tasks that are underachieving their execution times, so they can have a chance to keep up with the execution of the application.

Figure 2.14. System setup.



Source: Shabbir (2011, p. 136).

Figure 2.14 presents the experimental setup used in this paper. The user interface is responsible for simulating inputs and consuming outputs from the applications and the application managers. The resource manager executes the RM's algorithms and the computation platform consisting of six processors. Two applications are used as benchmarks: a JPEG decoder and an H.263 decoder.

Table 2.3 presents the processor utilization for the three RMs considering applications running a 2 QCIF frames/sec for the JPEG decoder and a 40 frames/sec for the H.263 decoder. Here, the authors consider the processor utilization as the ratio of time spent on the execution of applications and the total processor running time. It is possible to see that the rate-based RM has the highest processor utilization while the centralized and credit-based have similar results. Authors argument that this is due to the fact that the Rate-based RM lets applications execute continuously and try to use the computing resources to the maximum.

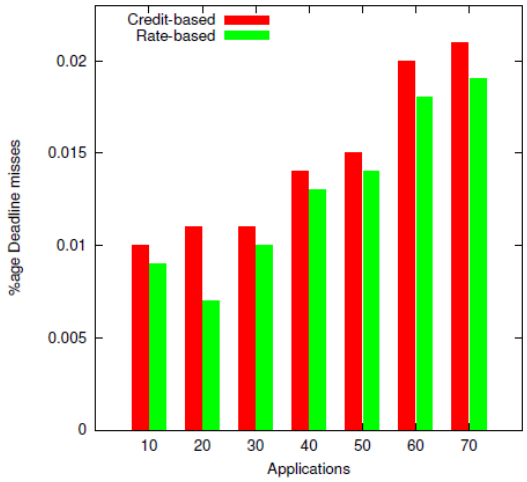
Table 2.3. Processor utilization resulting from different resource managers.

Centralized	Credit-based RM	Rate-based RM
0.1672	0.1625	0.8074

Source: Shabbir (2011, p. 139).

The paper also presents a scalability comparison between the two decentralized resource managers, by comparing their executions with an increasing number of applications and processors (a 10-processor platform was used to simulate up to 30 applications, whereas, for 40-70 applications, the authors used a 20-processor platform). Figure 2.15 shows that both resource managers scale well as the number of applications increase if taking into account the deadline misses for each one of them. In addition, the Rate-based RM presents smaller number of deadline misses in all situations.

Figure 2.15. Scalability of resource managers.



Source: Shabbir (2011, p. 139).

This work shows an interesting investigation towards the scalability of resource managers by illustrating how efficient a decentralized approach may be if compared to a centralized one, even if considering only 6 processors (in some cases up to 20 processors). However, this hardware scenario is still simplified since there is no mention to the memory subsystem or to the interconnection mechanism and how they would play a role in these experiments. In addition, the authors use as benchmarks Synchronous Data Flow Graphs and the resource managers are described in POOSL language for reactive systems. As consequence, these applications are represented as state machines and even though it does not invalidate the results at all, this makes clear that the proposal does not consider some specifics of the application characteristics, such as the implementation of communication and memory accesses. The way how these features are implemented can have influence on the result of the overall system and on the resource management decisions.

Several other papers propose distributed resource managers. In (ANAGNOSTOPOULOS, 2012), the authors propose a method to perform distributed resource mapping using a divide-and-conquer approach. During admission of a new application, the resource managers divide the network in regions based on the application size and assign it in a cluster using a best-fit method. Each region works as a cluster, and local resource managers map the tasks inside a region. The authors in (WEICHSLGARTNER, 2011) propose another decentralized method to map applications with the goal of reducing congestion in a NoC interconnection. This approach has a resource management mechanism that only foresees a limited amount of neighbors around the processor running the initial task. Authors present this work only for tree-based NoCs. The effects and usage of this method in mesh topologies is unknown.

Some papers tackle the management of interconnection resources. In the case of NoC-based systems, some ideas are to allocate the routers exclusively to regions or change router specifications at runtime depending on the applications currently running.

Following this idea, a decentralized approach is detailed in (HEISSWOLF, 2012), where the authors propose a hardware mechanism that allocates the network-on-chip resources to favor regions of processors executing the same application. These regions

are defined at runtime depending on the task mapping. This work has the goal of isolating application with different QoS requirements. Depending on the application requirements, the router resources (like Virtual Channels and Buffers) can work with a Guaranteed Service (GS) or a Best Effort (BE) approach. The main idea is to avoid allocating all resources to a GS connection, which would block an eventual BE flow This resource allocation policy for either a Guaranteed Service traffic or a Best Effort traffic is triggered by the distributed OS, and three mechanisms are implemented to support this reconfiguration: Local Reconfiguration, in which the processor connected to the router can access its internal registers to modify the QoS policy; Direct Remote Reconfiguration, where each tile can configure all routers in the NoC; Indirect Remote Reconfiguration, which is a local reconfiguration triggered by a remote tile. With these mechanisms, the distributed OS running in any tile in the system can reconfigure any region.

The authors in (BRUSCHI, 2011) propose a centralized on-chip NoC resource manager. In this approach the resource allocation policies are configured based on a communication monitoring. A centralized QoS management approach through virtual channel allocation is proposed in (WINTER, 2011). In this paper, the resource management is performed by means of a NoC manager implemented as a dedicated hardware unit. The problem with centralized resources lies on the bottleneck created by the necessity of making decisions on every resource available. It is important to keep in mind that resource managers were born with the intent to be used in large multi-core systems. Therefore, it is reasonable to assume that fully centralized approaches will never work as well as distributed ones in these systems. Centralized resource management strategies lack from worse scalability. Hence they are not appropriate to be adopted for future manycore systems.

Some of these works may seem similar to the one presented in this proposal. In the particular case of the invasive computing approach, the mechanism works at different levels of abstraction, and, therefore, the programmer must have some notion of the methodology. In the case of the Processor Clustering proposed in the Chapter 3 of this thesis, all mechanisms are transparent to the programmer, both at scheduling level and at hardware level. With that, Processor Clustering offers support to legacy parallel programmed codes. Also, this proposed cluster-based mechanism does not rely on the middleware neither on the OS to run applications programmed using distinct parallel programmed applications. To the best of our knowledge, this is the only work that offers this feature.

In addition, none of the presented works deal with memory resources re-distributing them according to applications needs. This is a very important feature, since the memory subsystem has always been considered one of the main bottlenecks of computer systems. Another aspect is the fact that tasks may have distinct needs for processing and memory resources.

Another aspect of the resource management is the nature of the entity responsible for the distribution of the resources. Most works in the literature point to the fact that dynamic resource managers are required. This is due to the dynamic nature of current systems, with new applications loaded at any given point during execution. The solutions presented in this thesis proposal consider dynamic management of resources trying to adopt simple metrics and implementations in order to diminish the overhead of this kind of approach.

2.3 Task Mapping and Migration

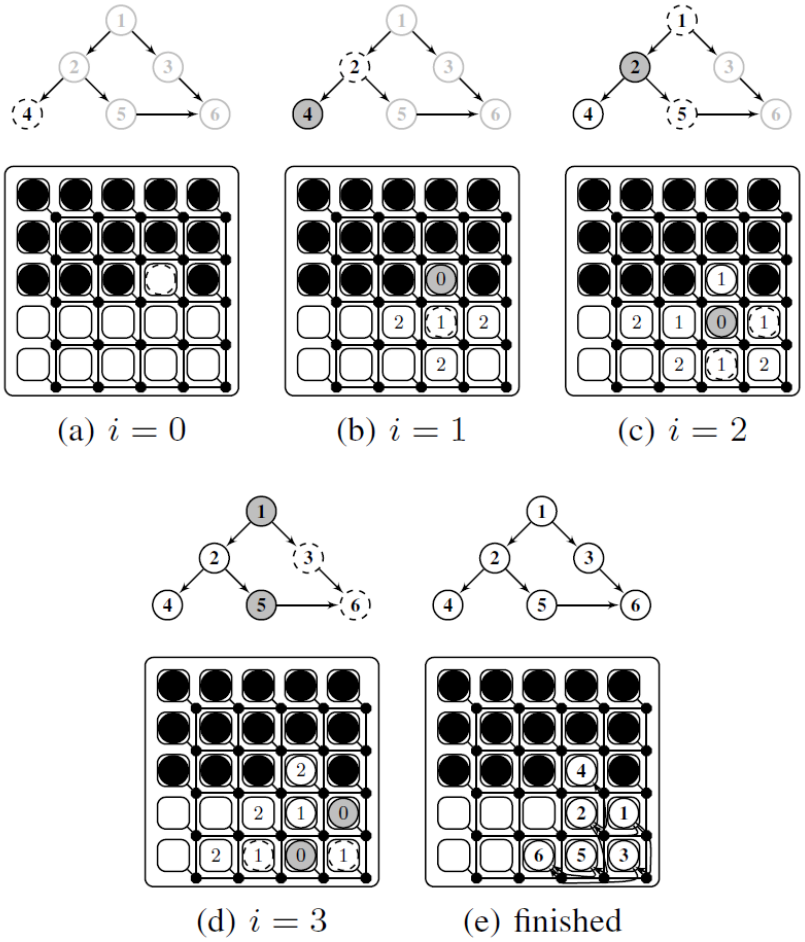
Since the advent of MPSoC era, chip designers see embedded systems as an aggregation of resources, and scheduling is the main responsible for the association between application and these resources. Some papers in the literature focus on efficient task mapping algorithms in order to improve some aspects of the system, such as performance, energy savings or communication throughput. In this section we briefly present the key ideas of these papers.

Ter Braak et al. (2011) propose a generic task mapping algorithm that can be used in several kinds of platforms using any cost function defined for the platform in question. In order to do that, the authors assume an indication of topology, but no assumptions are made for the routing algorithm. Another goal of this work is to provide an approach whose calculation time for the resource mapping could be performed at runtime in a feasible way in order to be used in a scenario of real-time applications.

Based on applications given in the form of SDF (Synchronous Data Flow) graphs, the mapping algorithm is proposed with the intention of fulfilling the resource requirements of tasks considering their communication demands. The authors propose an incremental algorithm where the task graph is traversed trying to match the topological structure of the platform. This mapping heuristic is a divide-and-conquer algorithm that tries to break the mapping problem in smaller problems of variable size. Essentially, the algorithm has three steps. First, a t_0 task is assigned to a resource and based on that location the other tasks are organized in sets of equal distance to t_0. After that, the problem is divided in smaller problems in which the algorithm traverses the task graph and searches for available resources physically close to the previously mapped resource and then maps the task into them.

One of the main concerns in this work is to avoid scenarios where the processing resources are isolated due to the lack of communication resources. To turn the efforts in this direction, the authors establish a resource fragmentation metric. This metric is defined by the percentage of pairs of adjacent processing elements in which only one element is used, over all pairs in the system. To reduce this fragmentation, at the end of the iteration on the graph the task with lowest degree of connection (sum of all edges) is mapped to processing elements that are likely to become isolated if not used at this point. Figure 2.16 presents a step-by-step example of the algorithm. The index *i* in each step represents the iteration of the algorithm. In Figure 2.16a, the algorithm starts with the task with lowest degree (task 4) and maps it to a core likely to become isolated (i.e. surrounded by cores running other tasks). Traversing the graph, each task is allocated to the most adjacent cores in terms of number of hops (steps in Figure 2.16b, c and d). Note that in Figure 2.16e, *task_3* is allocated to the core on the corner because this core would probably become isolated if not used at this point.

Figure 2.16. Mapping state after each iteration of the task graph.



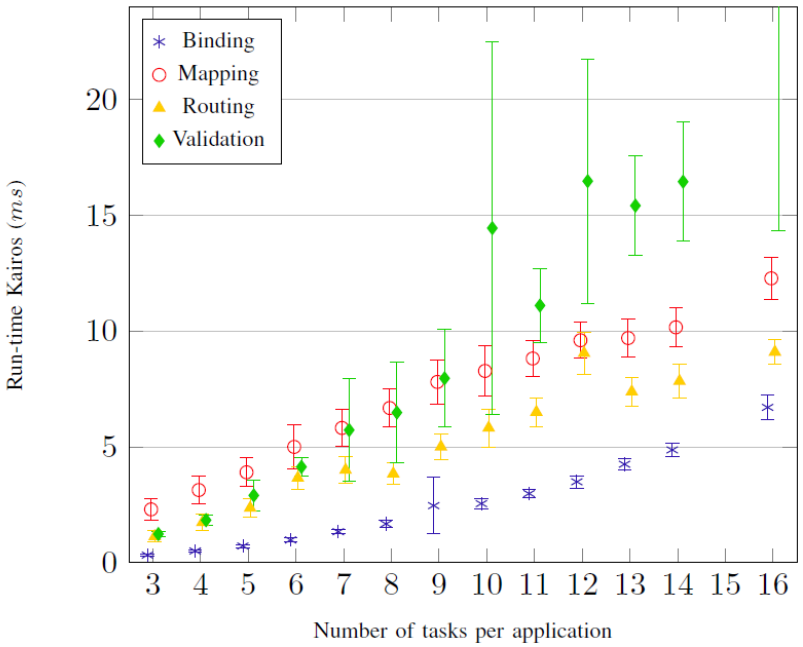
Source: Ter Braak (2011, p. 359).

During the iteration steps of the mapping algorithm, there are situations where the resources available match exactly the computation requirements of the task. However, the algorithm does not stop at the first solution found, because this would only favor the communication distance metric and neglect the resource fragmentation. To evaluate the cost of mapping, the algorithm must take into account the total communication distance involved in the candidate resource. After that, a sparse distance matrix is composed while searching the platform for resources.

The authors present results based on benchmarks composed of communication-oriented or computational-intensive applications. The first type of applications use between 10% and 70% of element resources, whereas computational-intensive applications use 70% to 100%. With that the authors claim that a scenario of communication bottleneck can be simulated.

Figure 2.17 presents the average execution time for successful resource allocation considering the different phases of the resource manager (binding, mapping, routing and validation). The algorithm scales well in most phases. The exception is the validation phase in which the requirements of the task mapping are evaluated. The authors state that the length of the simulation depends only partially on the size of the application.

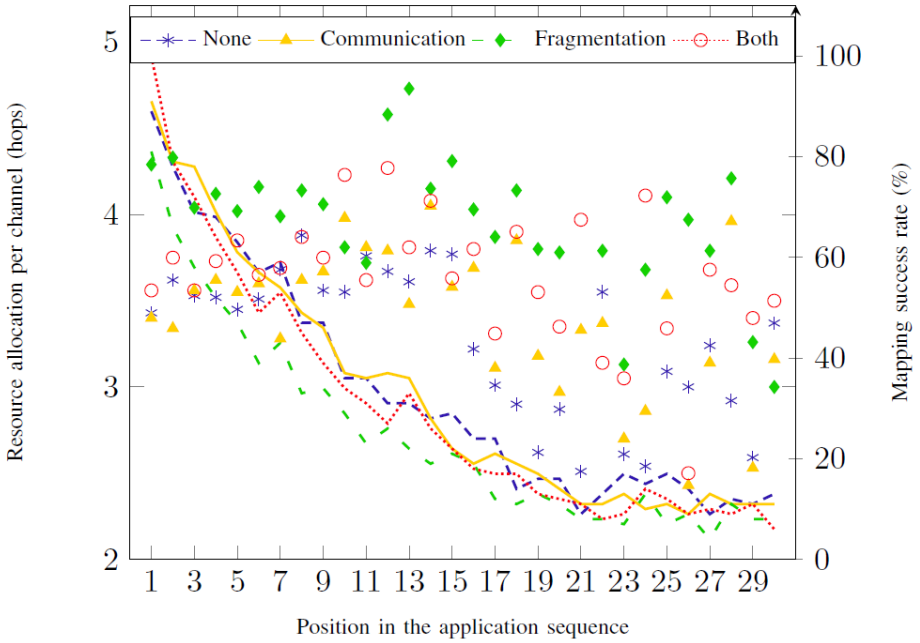
Figure 2.17. Resource allocation execution time.



Source: Ter Braak (2011, p. 361).

Figure 2.18 presents the average number of hops between two communicating tasks (Y-axis on the left hand side). The Y-axis on the right hand side is the mapping success rate (lines in the graph), and the X-axis represents the amount of tasks allocated. The authors add a certain number of applications to the system, one at a time. The proposed algorithm handles these new incoming applications and the allocation of all of them into the cores is considered a successful mapping. However, at some point, the system is already full and all its resources are in use. When this happens, it starts rejecting new applications causing a mapping failure. The figure shows that, with more than 15 applications allocated, the mapping success rate drops below 20%. The authors state that, when a system is almost saturated, new applications are only admitted if there is a set of adjacent resources available. This means that even with available resources some applications can still be rejected if the fragmentation is too high.

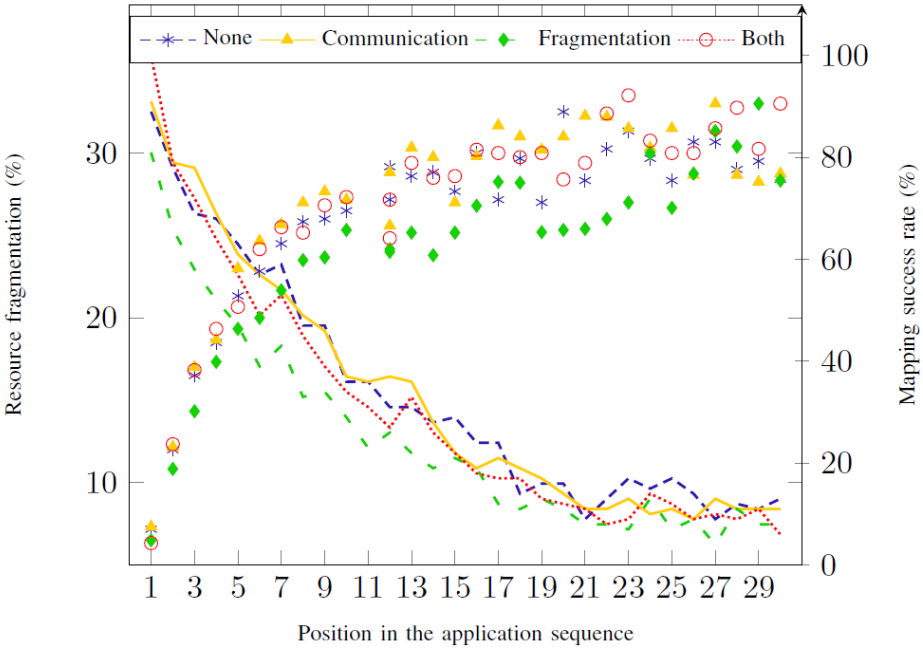
Figure 2.18. Average communication resources allocated.



Source: Ter Braak (2011, p. 362).

Figure 2.19 presents the external resource fragmentation (X-axis on the left hand side) when the application mapping progresses. According to the results, the fragmentation converges to 30% and the mapping success rate converges to 10%. This gives an idea about the required resource overhead in the platform. Even though this is a desirable feature, the excessive concern on resource fragmentation may lead to increasing communication distance, which lowers the mapping success rate.

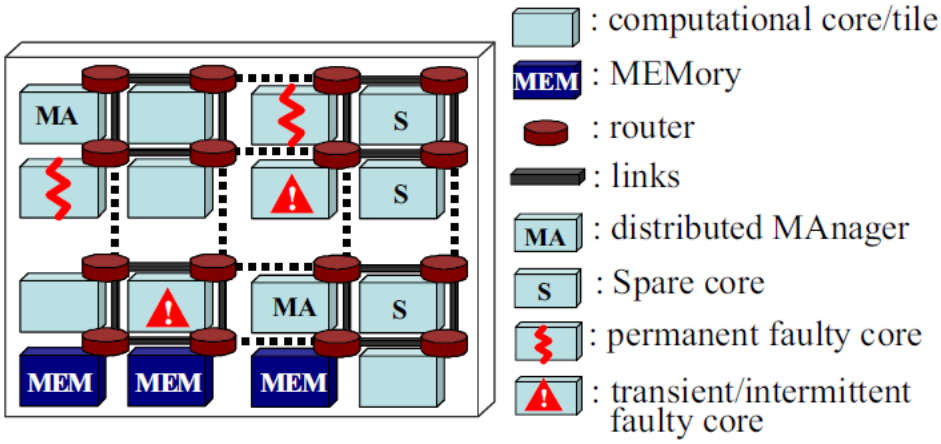
Figure 2.19. Average resource fragmentation.



Source: Ter Braak (2011, p. 362).

The authors in (CHOU, 2011) propose and evaluate a fault-aware resource management algorithm. The main goals are to minimize communication energy consumption and maximize the overall system performance. These goals are to be achieved by dealing with a faulty-prone scenario. Figure 2.20 presents the NoC platform used as the target of this resource management mechanism. This is a 2-D mesh tile-based architecture, and the resources to be handled here are manager cores, computer cores and memory tiles. The role of the manager cores (indicated in the figure as "distributed manager") is to decide which resource must be used by each task and to control the migration process.

Figure 2.20. 2-D mesh platform with spare cores.



Source: Chou (2011, p. 1).

In this platform, the Spare Core is used to replace faulty cores (the fault being either intermittent or permanent) or cores that become unreachable in the case of faults in the system interconnection.

Regularity in the task mapping is one of the main concerns of the authors in this paper. If the tasks from the same application are scattered throughout the system, this will lead to higher network contention and degrade the overall system performance. The faults that could happen in the platform create dangerous irregularities that could lead to non-contiguous task mapping. This kind of degradation must be handled at runtime.

The authors describe an energy model to quantify the task mapping solutions. This model takes into account the Manhattan Distance between tiles and the energy consumption of routers (including crossbar switches and buffers) and links.

The goal is to find a mapping function to allocate incoming application tasks to a reachable, available and fault-free set of processing cores in a way that:

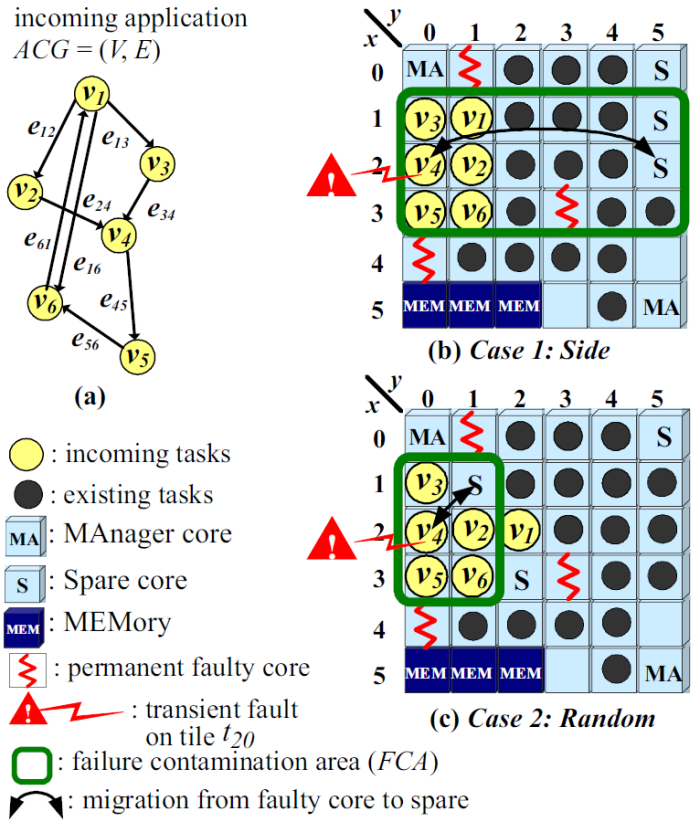
1. Communication energy is minimized;
2. Network contention is minimized; and
3. The entire system performance is maximized.

The use of spare cores in the system impacts the fault-tolerance in two aspects. First, if a processing core fails, it is unlikely that the entire system halts. Second, there is no need to shut down the system so the faulty core can be replaced. Instead, the faulty-core can be replaced by a spare core using task migration at runtime.

As part of the resource management, the authors present three different schemes to replace a faulty core. Figure 2.21a presents the Application Control Graph (ACG) of an

application that needs to be mapped in a 6x6 MPSoC. In the first scheme, the spare cores to be assigned are the ones to the right side of the system (as presented in Figure 2.21b). A second scheme uses a random approach to distribute the spare cores in the system. The final scheme distributes the spare cores evenly in the system in order to guarantee a reasonable Manhattan Distance between them and the possible faulty-cores.

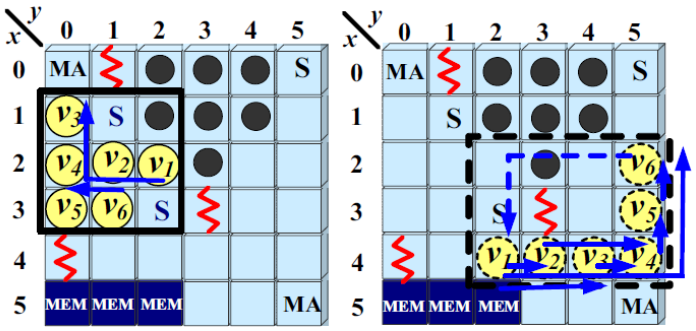
Figure 2.21. Spare cores distribution schemes.



Source: Chou (2011, p. 2).

In order to find the best mapping function, the authors establish three metrics to help achieve their goals. The *Weighted Manhattan Distance* takes into account the physical distance between cores, which correlates with the communication energy consumption. The *Link Contention Count* is characterized when two flows dispute the same link somewhere in the system. From the ACG used as input, it is possible to see edges that represent these flows. Finally, the mechanism should consider the *System Fragmentation Factor*, which defines how the non-contiguity of applications tasks may affect other regions. This fragmentation factor is a function of the area of the minimal rectangle that covers the mapping solution and the number of spare and faulty cores in that rectangle. Figure 2.22 presents two examples of mapping using a random distribution scheme of spare cores. Comparing the two mappings, the one on the left presents a lower *Weighted Manhattan Distance* and a lower *System Fragmentation Factor*. The one on the right presents a better *Link Contention Count* since it is more spread in the system, giving the opportunity to parallelize the communication flow. In order to evaluate the three metrics, the authors analyze the area of a Kiviat graph (MORRIS, 1974).

Figure 2.22. Mapping results using a random placement of spare cores.



Source: Chou (2011, p. 3).

Using an NoC monitoring scheme, the distributed managers keep tabs on the cores status and do reactive or even proactive migration when due. The method presented supports multiple applications entering and leaving the platform at runtime. The authors present a migration-based mechanism for newly admitted applications and for the case of a failure in the system. For new applications, the proposed mechanism allocates tasks based on the Kiviat graph that uses the three metrics explained earlier. To deal with failures in the system, the authors first introduce the concept of failure contamination area. This concept represents the new area in the system as the result of migration of a task due to failure. As seen in Figure 2.21a, tasks v_2, v_3 and v_5 communicate with task v_4, whose core failed. Task v_4 must migrate to a spare core, and the distance between this core and the others represents the failure contamination area. As seen, the example in Figure 2.21b shows a smaller failure contamination area than the one shown in Figure 2.21c. When a failure occurs, the task migration proceeds as follows:

1. Control messages are sent through the control network in order to notify the distributed managers so they can react and perform migrations proactively;
2. The corresponding distributed manager finds the closest available spare core that results in a smaller failure contamination area in order to suppress the propagation of failure;
3. Triggers the code migration or related data transmission using the data network.

An evaluation of the Fault-tolerant (FT) mapping is presented using communication intensive applications with patterns all-to-all (broadcast-like) or one-to-all (memory-centric). The communications rates for all ACG are randomly generated as well as the sequence of incoming applications. As for the spare cores, the authors used two schemes: the side placement, where the spares are assigned to the side, and the random placement, where the spares are randomly distributed. As for the faults, the experiments assume that 10% of the computational cores are permanently faulty and randomly distributed in the system.

Table 2.4 presents a comparison by means of throughput and communication energy consumption between the FT mapping method and a Nearest Neighbor (NN) heuristic. For all-to-all communication, the technique proposed in this paper reaches higher throughput and lower energy consumption. As for one-to-all communication, the proposed method still presents better results, although they are subtle due to the bottleneck-nature of the communication pattern. In this case, the authors do not discuss

how the Nearest Neighbor approach can take advantage of the spare cores and how it behaves when mapping an application to a faulty core.

Table 2.4. Performance and energy consumption results.

	spare core placement-Side		*spare core placement-Random*	
specific *ACGs in different NoC size*	***throughput improvement (FT vs. NN)***	***comm. energy consumption savings (FT vs. NN)***	***throughput improvement (FT vs. NN)***	***comm. energy consumption savings (FT vs. NN)***
5 × 5 *all-to-all*	23.2%	12.5%	23.5%	15.8%
10 × 10 *all-to-all*	98.1%	32.1%	102.1%	36.2%
5 × 5 *one-to-all*	3.4%	13.8%	4.1%	17.8%
10 × 10 *one-to-all*	5.7%	17.5%	6.9%	23.6%

Source: Chou (2011, p. 6).

A task mapping using distributed agents is proposed by Kobbe (2011). These agents are assigned at runtime to a random processor when a new application is admitted into the system. The agents search for processors in the neighborhood in order to assign other tasks. However, in this approach, the communication between agents and processors allocating tasks can hurt the overall performance since the random assignment of these agents can put them far away from the applications managed by them.

The work presented in (CARVALHO, 2009) evaluates different approaches for static and dynamic mapping of tasks on NoC-based MPSoC. The authors run experiments on a SystemC cycle-accurate platform using well-known task mapping algorithms like Simulated Annealing and Taboo Search as static heuristics and Path Load and Best Neighbor representing dynamic approaches. Two scenarios are considered. In the first one, a 5x4 homogeneous MPSoC is used to compare the mapping of a single application. In this case, the results show that dynamic algorithms are 4% worse than static ones and the energy consumption is 38% higher. The second scenario is for the mapping of multiple applications, and, for that, a larger environment is used (9x9 homogeneous MPSoC). Results show that the dynamic approaches present overheads of 8.5% latency and 15.5% communication energy consumption in average if compared to static solutions in the most complex scenario.

In some papers, the main memory is essentially considered a resource shared among all cores. Therefore, to execute a memory-bound portion of the code, the task must claim the processor and the shared memory resource. The Predictable Execution Model (PREM) is an approach where tasks explicitly specify when the main memory is going to be accessed. With that, the PREM scheduling problem is an instance of resource sharing in multiprocessors. This problem is more complex since both resources (memory and processor) must be acquired in order to execute the task.

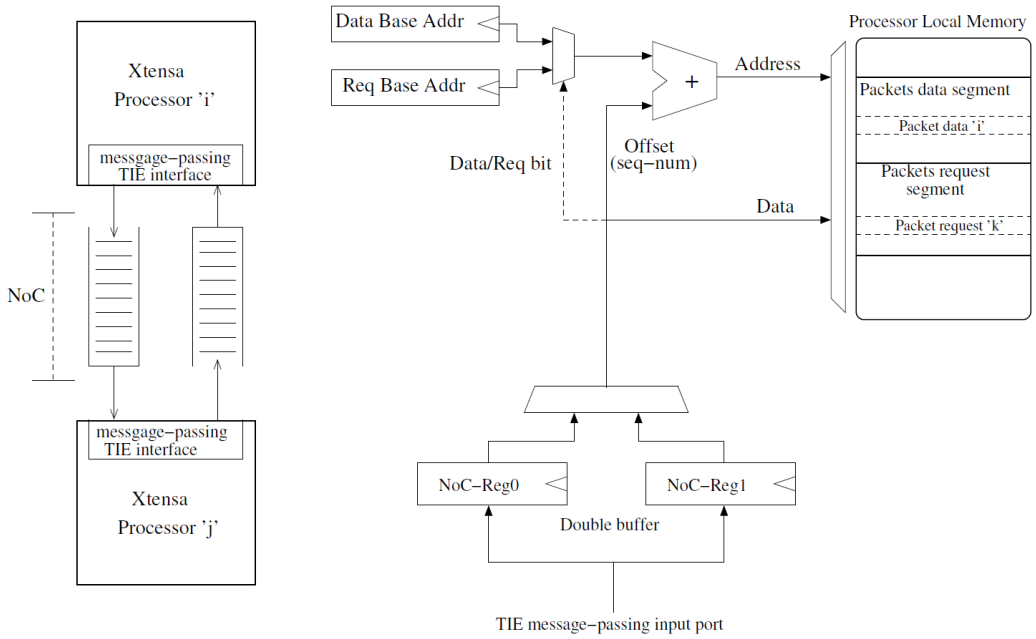
Brandenburg et al. (2010) defines the notion of blocking time in multicore scheduling and presents optimal results for resource sharing protocols. The paper deals with an interesting situation, where a task with high priority is waiting for a resource, and proposes two protocols, one that allows lower-priority tasks to execute (named s-aware blocking), and another one where this is not possible (s-oblivious blocking).

Kwon (2008) presents a retargetable parallel programming framework for MPSoCs. The authors developed a novel parallel programming model in which the functional parallelism and data parallelism of tasks can be specified independently from the target architecture. This programming model generates intermediate code that can be translated to MPI or OpenMP, depending on the target architecture. This kind of approach provides some automatic support for both major programming models. However, it loses compatibility of software when considering legacy code: all applications must be re-written in this new programming model to become retargetable.

In order to determine suitable schedulers to deal with PREM-compliant tasks, (BAK, 2012) uses a simulation-based approach. In this work, the authors performed the comparisons using schedulers based on Simple Partitioned FIFO Locking Protocol (SPFP), which uses a single global FIFO for all shared resources. According to the authors, the EDF-based scheduler, known as M-LAX, is based on the laxity of tasks, which is defined by the difference between the deadline and the amount of time remaining for a task to complete its execution. This represents the slack time, which is the amount of time that a task can spare and still meet its deadline. Results show that the M-LAX scheduler works better than EDF-based schedulers. The authors state that this occurs because tasks can have long deadlines but the computational time required can be just as long. This means that this task cannot be delayed excessively, otherwise the deadline will be missed. The EDF schedulers may realize that this task must have high priority when it is too late, whereas the laxity-based algorithm can give high priority to this task earlier.

Tota (2010) presented a NoC-based framework in order to support a hybrid shared memory/message passing approach. The authors propose a NoC-based MPSoC called MEDEA using Tensilica Xtensa LX processors. In this architecture the memory is composed by private segments for each processor (no processor is capable of addressing the segment of another processor). In addition, an extra segment is shared and can be accessed by any processor in the system. An extension of a message passing instruction is supported by the ISA (and the compiler). A hardware architecture (presented in Figure 2.23) is placed as the network interface of each processor and, as messages arrive, the data are placed in the respective memory location (in the private memory of the receiving processor) working as a DMA module. The goal is to provide support for the compiler in order to facilitate the development of ad-hoc scalable programming.

Figure 2.23. Hardware support for message passing communication and receiving interface details.

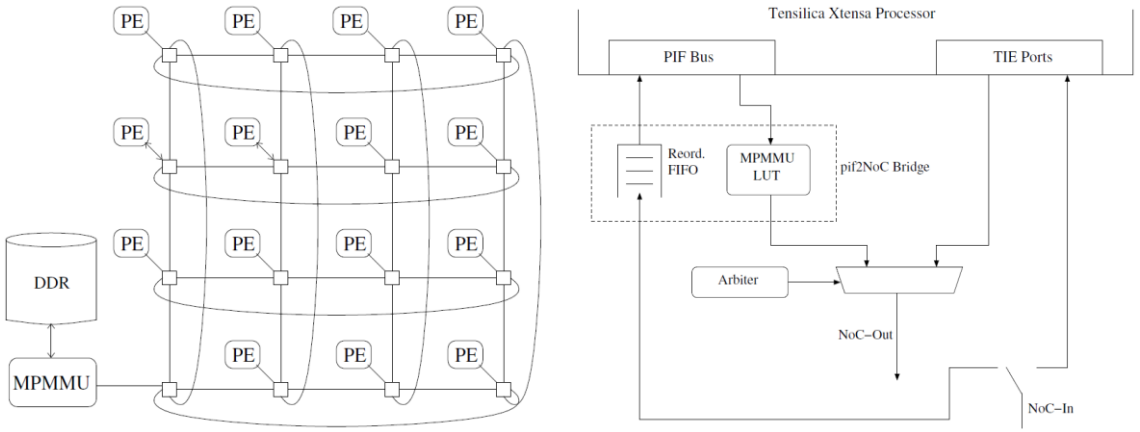


Source: Tota (2010, p. 46).

As mentioned, the MEDEA architecture offers a shared memory communication solution as well. A shared portion of the memory can be accessed using synchronization instructions *load* and *store*. The request data operations are mediated by a Multiprocessor Memory Management Unit (MPMMU) which is connected to a specific router in the system (as presented in Figure 2.24a). As shown in Figure 2.24b, the Tensilica Xtensa has a PIF (Processor Interface), which is used to access the shared memory segment. An arbiter is used to decide whether the NoC access is used for shared memory access or message passage send/receiving.

Each data request sent to the MPMMU follows a handshake protocol and some memory accesses can be granted in different order, depending on the availability of the memory block. This is used because cache coherence is maintained by means of lock/unlock atomic instructions.

Figure 2.24. MPMMU in a MEDEA architecture and shared memory/message passing interface.



a) b)

Source: Tota (2010, p. 46).

The authors show some space exploration results concerning cache size and cache write policy (where the write-back approach is better in every case). In addition, a set of speedup results is presented showing that the cache size has a major impact in the performance in the single benchmark presented (a Jacobi algorithm). In the end, the MEDEA architecture is a hardware solution to provide message passage communication (in order to provide scalability) and shared memory access, in an attempt to facilitate programmability. This solution is heavily supported by modifications in the processor and compiler.

Task mapping and migration are presented in a very large number of papers in the literature. These techniques are key in this new MPSoC era, where systems feature dozens or maybe hundreds of cores. Resource management depends heavily on these techniques, and, therefore, a resource-driven task migration is necessary. Task mapping and migration must consider hardware aspects of the system, such as spatial placement (with larger systems, the distance between cores become critical), nature of the application (degree of communication, memory access, parallel distribution) and also take into account the possibility of failures. In this thesis proposal, some of these aspects are explored using task migration decisions based on them. To the best of our knowledge, these platform-based metrics are unique in the literature.

2.4 Proposed work

In this thesis, a resource-aware cluster-based solution is presented. As shown in this chapter, this work is amongst some current solutions regarding the improvement of resource usage. Some papers present clusters that are not as flexible as the one presented here. This is due to the fact that the concept of clusters to be presented in the next chapter is based on logical mapping of applications. This increases the chances of better matching between applications (and their distinct numbers of tasks) and hardware architecture (with a variable number of available processors).

Other architectures focus heavily on resource-awareness in order to give higher priority to critical applications. As presented here, this kind of work can be very costly

regarding the methods used to identify the best resource allocation. By doing this at execution time, the overhead cost of such approach can be prohibitive and not scalable for a few dozens of cores. In this thesis a set of migration policies and redistribution algorithms is presented, based on a set of very simple intuitive ideas that do not require runtime negotiations.

3 PROCESSOR CLUSTERING

As presented in the previous chapter, a number of works have dealt with the resource management area. Typically, the problem to solve is to maximize the system performance by using the resources available in the best way possible. This concern has become a trend due to a change in the paradigm of embedded systems. In the past, these systems counted with two or three cores, when multi-core designs were used at all (several embedded designs used only one core). On top of that, the applications executed were, in most cases, known beforehand, thus giving space for design-time approaches for task mapping. Nowadays, embedded systems can have a dozen cores, and, in the near future, we are looking for hundreds or a thousand cores (SILVA JR, 2008). Regarding the software layer, the advent of smartphones has opened the possibility of several (and possibly very distinct) applications running at the same time. These same applications can also be added to the system at runtime. This heterogeneous software scenario also brings the possibility of applications with distinct parallel programming paradigms. This is a real possibility since the variety of applications brings a variety of developers with their own software development platforms or simply their fondness for a specific paradigm.

Given these recently emerged requirements, resource management efforts have been developed, especially directed to dynamic approaches to deal with runtime changes in the applications being executed. This chapter presents the proposal of a resource management approach that has as main features:

- An adaptive hardware-based mechanism to support distinct parallel programming paradigms;
- A dynamic virtual clustering mechanism to improve the use of resources without physical restrictions;
- A conservative approach in order to minimize overhead caused by task migration; and
- Exploration of hardware and software characteristics to improve the resource management.

As presented in Chapter 2, some of the works found in the literature establish the necessity of dynamic approaches. However, none of them exploited software and hardware aspects like task migration complexity for applications developed with distinct parallel programming models or memory latency. Also, the mechanism presented here relies on just a handful of assumptions on the applications and on the system, whereas most works in this area deal with pre-analyzed applications and knowledge on communication patterns. The solution presented here relies on a simple mechanism of

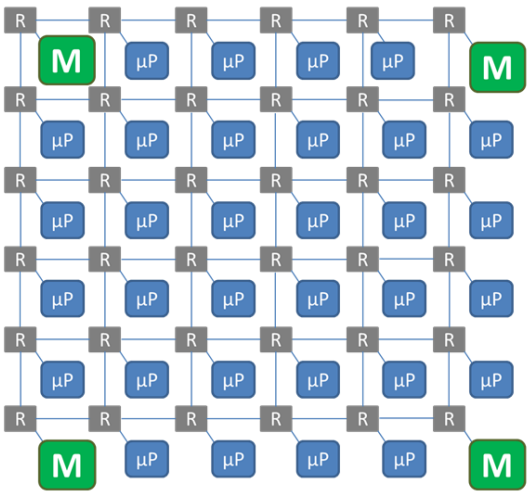
task migration (in order to minimize overhead) and decision based on availability of tasks as the applications are executed. In addition, the hardware support for distinct parallel programming models is another original contribution of this thesis.

3.1 Adaptable Hardware Support

As mentioned before, parallel programming models are closely related to memory organizations. For distributed memories, the use of message passing is largely used, while for shared memory the same goes for the use of shared variables. In any system where the parallel programming model does not fit the memory organization, a software layer is required to map the first one onto the other. Aiming at avoiding the overhead of this software layer, this thesis proposes a hardware solution that adapts itself, such that the same hardware platform may run applications written according to different parallel programming models without any overhead. Even the hardware overhead is negligible, since it occurs only during the scheduling phase of the application and not during its execution.

To provide hardware support for distinct parallel programming models, this work modifies the SIMPLE virtual platform, already used in previous works (SILVA JR, 2008). SIMPLE is a cycle-accurate virtual platform described in SystemC (SYSTEMC, 2006). It is configurable in terms of number of cores, memory organization (GIRÃO, 2011) (shared, distributed, distributed shared, and physically centralized but logically shared), cache size, and placement of components on the NoC. The Processor Element (PE) used is a MIPS R2000, and all components are interconnected through a mesh NoC called SoCIN (ZEFERINO, 2003). SoCIN implements a wormhole packet switching to reduce energy consumption. It also uses XY routing to avoid deadlock situations and a handshake control flow. Additionally, each router has five bi-directional ports with input buffer size of four phits. The phit size is four bytes. Figure 3.1 presents this architecture with its core nodes and the routers. The small μP boxes represent regular core nodes while bigger M boxes represent nodes with the same core plus an off-chip memory controller.

Figure 3.1. Simple virtual platform.



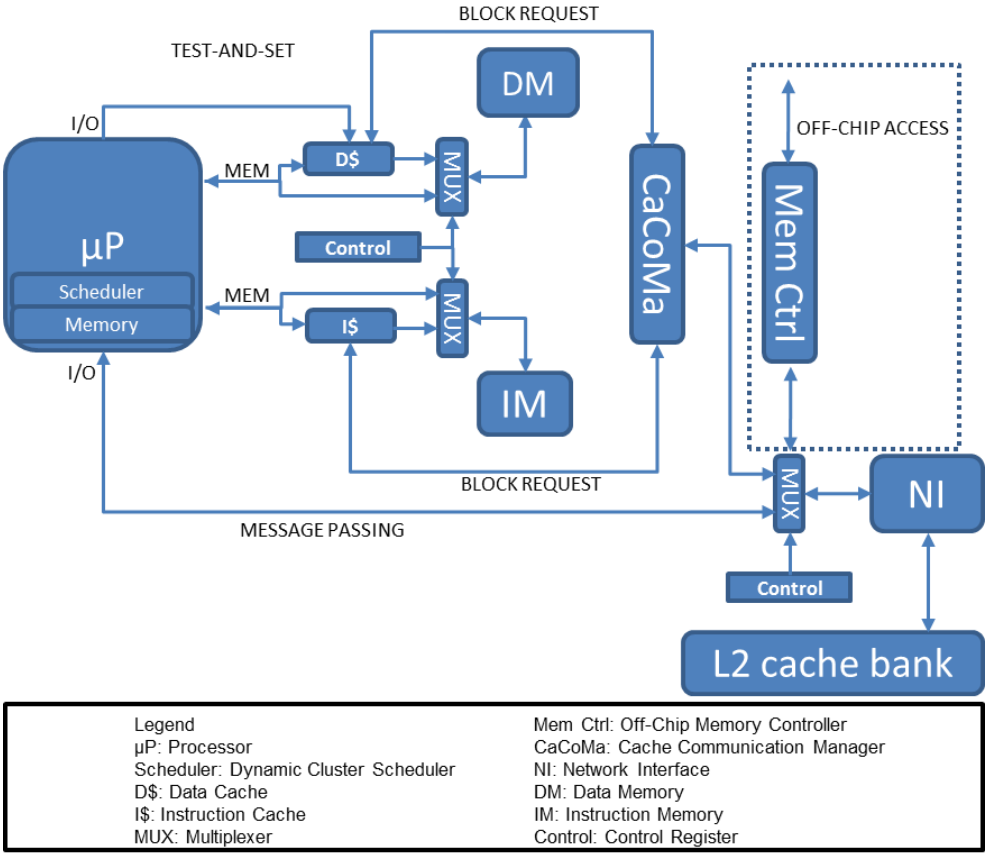
Source: Girão (2011, p. 65).

Figure 3.2 presents the core node of the platform. In this node there are a MIPS processor, a local memory organization, and the Network Interface (NI). For cores placed in the corners of the platform the memory controller (inside the dotted rectangle) is added. When the MIPS requests a read or write operation in one of the memories (instruction or data memory), multiplexers select if the request is either forwarded directly to the local memory (in case of a distributed memory organization) of if it passes through the cache controller (in case of a shared memory) (GIRÃO, 2011).

When a new application is launched, a scheduler that allocates application threads to each core must have information about the programming model of the application. Once the scheduler is aware of this model, it writes adequate values in the Control register. This register controls if the memory is accessed by the processor or by the cache controller. It also controls if either only the processor has direct access to the Network Interface (to send a message) or if the Cache Communication Manager (CaCoMa) is the only component to use the NI to request a block from the remote memory.

It is important to notice that (1) the switching between memory organizations occurs only when the scheduler changes the applications running on the processor and (2) each application runs on a given memory organization during its whole lifetime. This is due to the fact that each application is developed using a fixed parallel programming model defined by the programmer before compilation.

Figure 3.2. Adaptable hardware support for different programming models.



Source: Girão (2011, p. 65).

The solution presented here provides flexibility and transparency to the programmer since he/she does not need to know the underlying hardware memory organization in order to develop an application. The work presented here is part of an overall solution that provides management of a hardware platform with full awareness of its resources. These aspects are presented in the next sections.

3.2 Virtual Dynamic Clustering

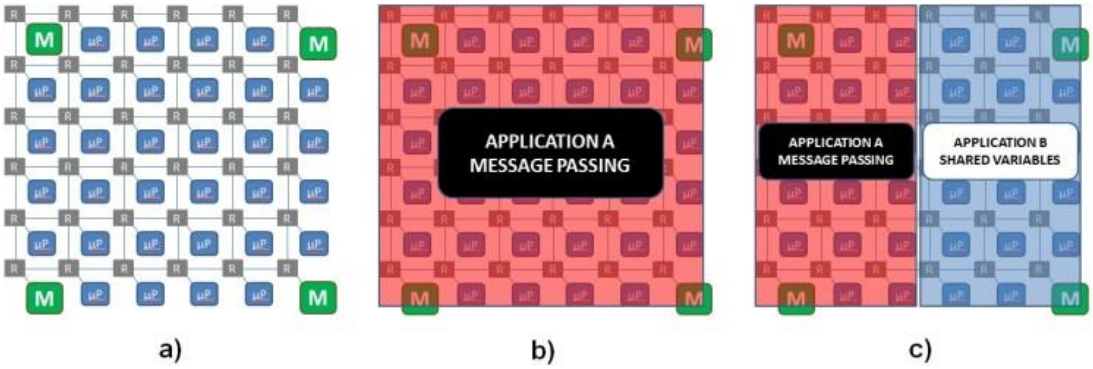
In this work, each allocated application generates, and it is treated as, a cluster. That means that, when an application is admitted into the system, its tasks are allocated on contiguous processors. Therefore, this aggregation of processors executing tasks from the same application is considered as a cluster. These are considered virtual clusters in the sense that there are no physical structures in the architecture that aggregate these tasks and processors. The concept of clustering is given logically by the aggregation of tasks from the same application. Thus, the number of virtual clusters in the system is equal to the number of applications being executed.

These clusters are also dynamic regarding the number of processors (hereafter referred to as the size of the cluster). This occurs based on a work stealing policy in which, every time a processor is below a certain threshold of occupation (number of tasks allocated in it), the scheduler tries to steal a task from a processor that has more tasks. Hereafter, this virtual dynamic cluster mechanism will be referenced as Processor Clustering.

The following example illustrates the principle of Processor Clustering (GIRÃO, 2011): Figure 3.3a presents a situation where, initially, there is no application allocated in the system. Eventually, an Application_A must be allocated and the scheduler may provide all the resources it needs (Figure 3.3b). Later on, a second application arrives and the scheduler must re-allocate some resources from Application_A in order to run Application_B. Considering that this new application may have been developed using a different programming model than Application A, the scheduler also has to switch the memory organization on the nodes allocated to B to the one suitable for this application. It is important to notice that the nodes allocated to Application_A do not switch from one memory organization to another. Only the nodes that now run Application_B need to change. The applications run only on those cores allocated to them. This creates a logical level of separation between applications, referred in this work as Processor Clusters.

Another part of this work is a multi-level scheduler that not only adapts the memory organization to the application programming model but also allocates tasks according to the available resources (processors and memory) in the platform. This allocation takes into account, amongst other characteristics, the dynamic change of degree of parallelism in an application (XU, 2000).

Figure 3.3. Processor allocation and creation of virtual clusters.



Source: Girão (2011, p. 65).

In order to provide intra-cluster scheduling, a copy of this multi-level scheduler resides in a small memory of each processor and runs when a quantum expires or the current running task finishes. Eventually, this scheduler must also set the memory organization that suits the task that will be running on it. This is performed by writing on the control registers, as seen in Figure 3.2.

Every time a task finishes its execution, the scheduler modifies a Resource Occupation Map in order to keep track of how busy a processor is. This map holds the information of which resources are currently allocated to each application and how many tasks each processor has. Table 3.1 presents the contents of a hypothetical Resource Occupation Map.

Table 3.1. Resource Occupation Map

Resource	NoC Address	Application	Number of tasks allocated
Processor 01	0x0	0	3
Processor 02	3x5	2	2
Processor 03	3x6	2	1
Processor 04	6x8	4	2
Processor 05	7x4	5	1

Reads and writes on this map are serialized by a directory-based cache coherence mechanism used for regular memory requests in the system. This cache coherence mechanism uses a hardware block attached to each L2 cache block to keep track of every block in the L1 caches. The idea is to prevent multiple caches from writing on the same block at the same time. In order to do that, this directory protocol makes use of invalidations and write-back requests. This cache coherence protocol is detailed in Chapter 4. It is important to notice that, since this cache coherence mechanism has a block-based granularity, two schedulers can potentially read and write on this table if the information required is placed in different memory blocks. In addition, another set of information regarding the clusters status is stored. This information is gathered in a table called Cluster Descriptor, as shown by the example in Table 3.2. Here, each entry lists certain features of a cluster, such as number of resources, number of tasks, the workload rank of each cluster, the parallel programming model (Shared Variables or Message Passing), and coordinates of the farthest points of the cluster, to give an idea

of how spread it is. This last information will be useful to avoid fragmentation or restrain the average minimum distance between nodes.

Table 3.2. Cluster Descriptor

Cluster	Number of resources	Number of tasks	Workload	Programming Model	Higher		Lower	
					X	Y	X	Y
Cluster 01	3	6	2	SV	1	1	0	0
Cluster 02	4	8	3	MP	3	1	2	0
Cluster 03	2	3	1	MP	1	2	0	2
Cluster 04	5	8	5	SV	4	3	0	3
Cluster 05	4	6	4	MP	7	5	6	4

Every time a task finishes, the scheduler verifies if the number of current tasks allocated in this processor is below a certain threshold (initially, half of the initial number of tasks, in an effort to spread the tasks equally among the cores). If this happens, the scheduler looks in the Resource Occupation Map for a task to be migrated to this currently idle processor. In order to diminish the overhead of the task migration, the candidates to share the load of tasks (target processors) are limited to processors in the neighborhood. The extension of this neighborhood is given by a certain number of hops (*NH*) to be determined at design time. Primarily, the choice of the task to be migrated is based on the following criteria:

- The candidates are confined in a pre-defined proximity area of *NH* hops;
- Preference is given to a migration from the most overloaded processor (number of tasks) in the same cluster; and
- Preference is given to a migration between two processors that are closer to each other.

The processor that best fits these conditions will receive a work-stealing message and a task migration will take place. This mechanism is illustrated in Algorithm 1.

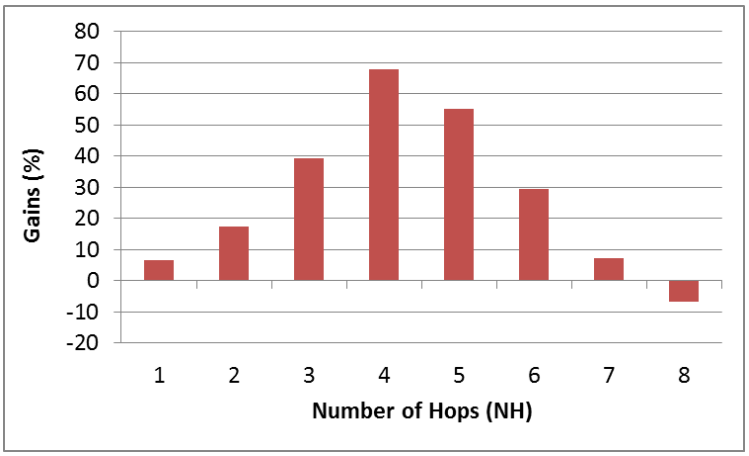
Algorithm 1: Baseline resource task migration

```
01 Selects the cores in a NH hops radius and put them in target_list;
02 Ranks the processors by cluster;
03 Ranks the processors by number of tasks executing;
04 if target_list.elements > 1 then
05        chose randomly from the remaining list;
06        sends a work-stealing packet;
07        receives tasks;
08 end if;
```

This is a very simple way to decide the task migration, and, in this work, it will be used as the baseline against which more elaborated policies will be compared.

An important parameter in this method is the number of hops that defines the radius in which the scheduler searches for a target processor. Figure 3.4 shows the results of a set of experiments with the intent to establish the most suitable value of *NH*. The bars represent the percentage of performance gain, in average, for different MPSoC sizes (16, 32, or 64 cores).

Figure 3.4. Exploration of the *NH* parameter.

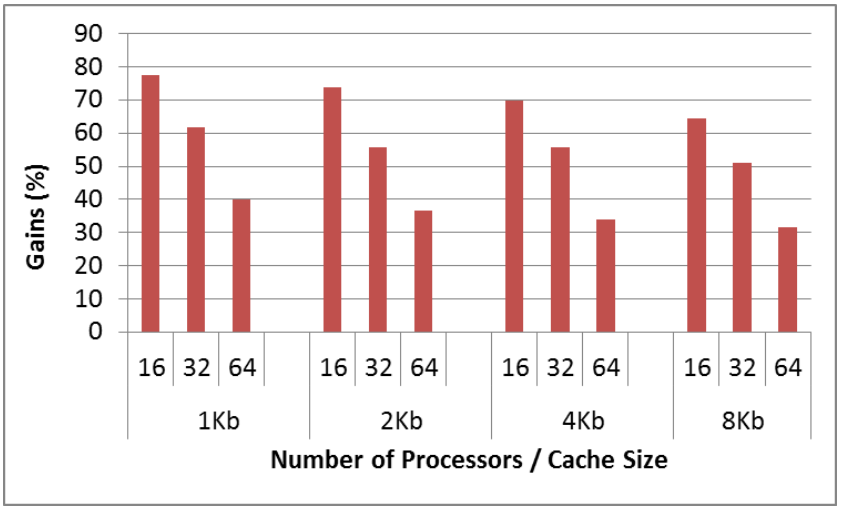


As presented in the figure, 4 seems to be the best number of hops for the searching radius and it will be used for the *NH* parameter in the subsequent experiments. Other values like 3 or 5 could be used, but here we are considering a single best fit for all MPSoC sizes.

In order to give an idea of how this dynamic clustering can improve performance and energy savings, an experiment comparing the execution of a set of applications in SIMPLE is presented. For this experiment, we consider three applications: a matrix multiplication, a motion estimation, and a Mergesort algorithm. These applications will be used in future experiments and are detailed in Section 3.4.1. In addition, we consider the execution in three MPSoC sizes (16, 32, and 64 processors) and four different L1 cache sizes (1Kb, 2Kb, 4Kb, and 8Kb).

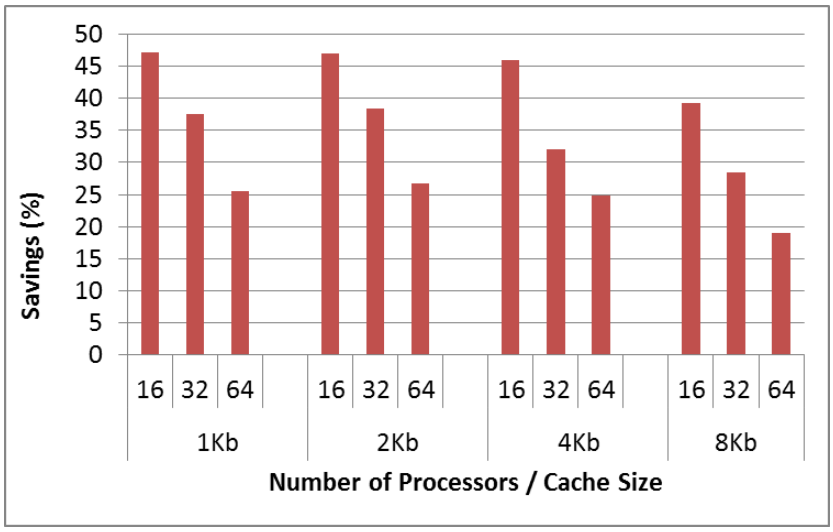
Figure 3.5 and Figure 3.6 present the performance gains and energy savings, respectively, when comparing the results with and without Processor Clustering.

Figure 3.5. Performance improvements using dynamic clustering.



Source: Girão (2011, p. 68).

Figure 3.6. Energy savings using dynamic clustering.



Source: Girão (2011, p. 68).

For this experiment, the scheduler is aware of tasks that finish, which could lead to an increase in the number of processors for a cluster, which would eventually generate performance gains and possible energy savings.

These results show that the improvement of performance can reach almost 80% and energy savings up to 47%. This shows that, as the execution progresses and the threads fnish their execution, the remaining clusters dynamically increase their size and become computationally more powerful, up to the point where obvious overheads such as resource concurrency and task migration are overcomed by the increase of execution speed. The scenarios with 64 processors present the smallest improvement due to the fact that more processors in the system means less tasks per processors, which leads to a lower need for extra resources if compared to the other two scenarios.

It is important to notice that, in this comparison, the baseline system is executing the same applications, but, when a processor finishes the execution of the tasks allocated in it, other tasks do not migrate to use this newly available resource. The improvements presented here suggest a reasonable opportunity to leverage on these resources, culminating in performance gains and energy savings.

Other two aspects contribute to the results presented so far. Both are connected to the choice of executing two out of three applications programmed with shared variables (with a shared memory organization). This has two consequences: the cache size becomes more relevant, and the overhead caused by task migration decreases. Task migration becomes less costly due to the fact that, in this case, the migration is performed by only sending the context from one point to another. In the case of distributed memory, the migration incurs in transferring all the contents of the instruction and data memories from one node to another.

Further on, four different policies that enhance these criteria to achieve greater efficiency are proposed (GIRÃO, 2013) and discussed in the sequence. It is important to notice that the criteria presented in this section are used as a tie-break (giving preference for keeping the resource in the same cluster) in situations where the use of one of the

policies results in more than one cluster candidate to own the resource recently available.

3.3 Resource Mapping Policies

This section presents four policies used to map resources that become available when the tasks that were once running on the processor have finished. These are policies based on characteristics of the applications that certainly impact on the overall performance of the system. Each one of these policies determines the main rule to be applied to perform the resource allocation. It is possible to imagine other policies that are combinations of the key ideas of each of these policies. However, in this work we are trying to investigate the impact of these individual aspects, and the formulation of new policies is considered to be a future work.

3.3.1 Shared Variables First (SVF)

This policy tries to take advantage of the fact that Shared Variables applications (shared memory) are easier to migrate than Message Passage applications (distributed memory). This is due to the lack of large amounts of data to be sent from one local memory to another when this programming model is used. This intuition comes from the fact that the task migration of Message Passage applications must send the whole task code and data from one processor's memory to another. On the other hand, for Shared Variables applications, the context is the only data that need to be sent. Also, Message Passage applications typically execute faster due to the availability of data in their local memories (as shown in (GIRÃO, 2011)).

In this policy, illustrated in Algorithm 2, the first criterion is to migrate a Shared Variable task to occupy the resource that becomes available. With this policy, it is expected that scenarios where the largest applications are based on Shared Variables will take more advantage of the resources.

Algorithm 2: Shared-Variables policy

```
01 Selects the cores in a NH hops radius and put them in target_list;
02 Ranks the processors by programming model;
03 Ranks the processors by number of tasks executing;
04 if target_list.elements > 1 then
05        choose randomly from the remaining list;
06        sends a work-stealing packet;
07        receives tasks;
08 end if;
```

3.3.2 Higher Workload (HWL)

Typically, applications with higher workloads take more time to finish, and the intuition in this policy is to prioritize this kind of application by giving it as much resources as it needs (provided the resources are available). In this case, the scheduler will have a pre-determined rank of the applications based on their workload. This will create a behavior where, as resources become available, they will be allocated to clusters that are running the application with higher workload and that have at least one overloaded processor. The tendency is that the applications with higher workload will have all resources as soon as they become available in order to accelerate their

execution. If the cluster does not have any overloaded processor, the next available resource will be given to the application with the next higher workload, and so on. The implementation of this policy is shown in Algorithm 3.

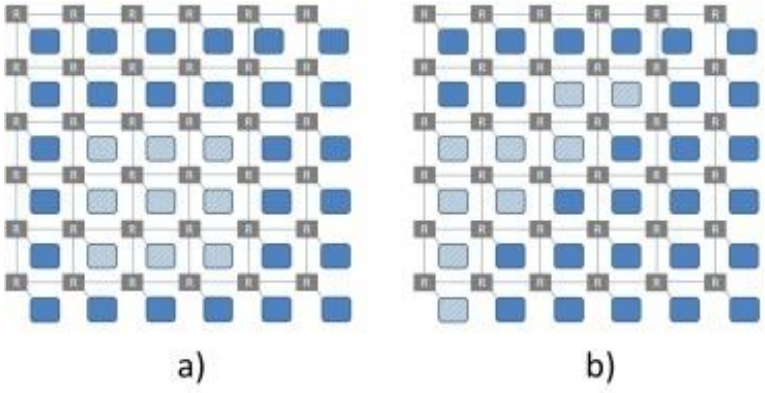
Algorithm 3: Higher Workload policy

```
01 Selects the cores in a NH hops radius and put them in target_list;
02 Ranks the processors by workload;
03 Ranks the processors by number of tasks executing;
04 if target_list.elements > 1 then
05          chose randomly from the remaining list;
06          sends a work-stealing packet;
07          receives tasks;
08 end if;
```

3.3.3 Cluster Shape (CS)

Given the nature of the Processor Clustering mechanism, the cluster size may change (having more or less nodes). However, given the fact that the architecture presented is interconnected by a Network on Chip, this dynamic cluster size can lead to a non-efficient shape of the cluster. It is well known that the minimal average distance between nodes of a 2-D mesh NoC is obtained when the system topology forms a square. Any other rectangle topology is less efficient and, therefore, less desired. Figure 3.7a shows a cluster with minimal average distance between their internal nodes (hashed cores). Figure 3.7b shows some topologies where the average distance is higher.

Figure 3.7. Cluster Shape Policy.



With that knowledge in mind, the Cluster Shape policy tries to give the available resources to a cluster as long as the resulting topology becomes as close as possible to a square, thus leading to smaller fluctuation on the average distance between the nodes of the same cluster. In order to do that, the scheduler calculates an impact value for the candidate target core. This impact is proportional to the increase in dimensions (X or Y) that the addition of this core will bring. For instance, if the target core is located between the most distant cores of the cluster in both directions, the impact will be zero. However, if the target core is located beyond the edges of the cluster, the impact will be

equal to the minimum number of hops to reach it. The implementation of this policy is shown in Algorithm 4.

Algorithm 4: Cluster Shape Policy

```
01 Selects the cores in a NH hops radius and put them in target_list;
02 Ranks the processors by number of tasks executing;
03 For each target_list.core do
04          cluster = target_list.core.cluster
05          impactX = impactY = 0;
06          If (coordX > cluster.maxX and cluster.minX) then
07                    impactX = coordX - cluster.maxX;
08          Elsif (coordX > cluster.maxX and cluster.minX) then
09                    impactX = coordX- cluster.minX;
10          endif;
11          If (coordY > cluster.maxY and cluster.minY) then
12                    impactY = coordY - cluster.maxY;
13          Elsif (coordY < cluster.maxY and cluster.minY) then
14                    impactY = coordY- cluster.minY;
15          endif;
16          target_list.core.shape_impact = impactX + impactY;
17 endfor;
18 chooses processor with lower shape_impact;
19 Sends a work-stealing packet;
20 Receives tasks;
```

3.3.4 Off-Chip Memory (OCM)

Typical embedded systems have very limited on-chip memory due to their constraints of area, power and energy. However, nowadays, applications have ever increasing amounts of data to be dealt with. Typical streaming applications have to store chunks of the stream in an off-chip memory (most commonly a flash memory). Due to limitations on the number of pins in a chip and also on the number of memory ports, the number of memory controllers must also be limited. In large MPSoCs the distance between the actual core and the memory controller location on the NoC may have an impact on the overall latency. In this policy, we try to take advantage of that fact by mapping tasks as close as possible to a memory controller. For this particular experiment, we consider four memory controllers on the corners of the NoC in order to have an overall worst case scenario. Algorithm 5 shows the implementation of this policy.

Algorithm 5: Off-Chip Memory policy

```
01 Selects the cores in a NH hops radius and put them in target_list;
02 Rank the processors by number of tasks executing;
03 Ranks the processors by distance to a memory controller
04 Chooses first processor with largest distance from a memory controller;
05 Sends a work-stealing packet;
06 Receives tasks;
```

3.4 Experimental Setup

3.4.1 Applications

The experiments consider four applications: a matrix multiplication, a motion estimation algorithm, a Mergesort algorithm, and a JPEG encoder.

The Matrix Multiplication (MM) was parallelized in such a way that each processor multiplies a subset of lines of matrix A by a subset of lines of matrix B. Each matrix used in simulations has 256 x 256 elements. All processors compute their part and at the end only one processor puts all the results together.

In the Motion Estimation (ME), every PE searches a macroblock (a subset of an image) in a different part of the reference image. In the simulations, a macroblock of 128x128 pixels and an image in PAL/SECAM format (1024x576 pixels) have been used. Since the reference image is divided equally among all processors, they typically finish their execution roughly at the same time.

For the Mergesort (MS), the parallelism takes advantage of its divide-and-conquer nature. Initially, each PE performs the Mergesort on a subset of the vector. Afterwards, one PE is responsible for assembling the whole vector, using the subsets already ordered by the various PEs. In these experiments a 16384 element vector was used as input.

A JPEG encoder can be seen as a three step algorithm. The first two steps (2-D DCT and Quantization) can be performed in parallel for different parts of the image. However, the third step (Entropy coding) can only be correctly performed with the whole image. Based on that, this parallel approach divides a QCIF image (176 x 144 pixels) in thirty two 22x36 blocks, and each PE is responsible for executing the first two steps on an equal amount of those thirty two blocks in a software pipeline fashion. At the end of those steps, each PE sends the resulting blocks to a master PE, which performs the final step with the complete image.

Table 3.3 shows the resource requirements (in this case, only processors) of each application throughout its execution (*NT* means the initial total number of tasks). With this table, we are trying to predict how fast an application releases processors, which, in turn, can be allocated to other applications that need more processors at the same time window. This is important to be established since it plays a key role on the advantage that dynamic clustering might have. If all tasks in all applications spend the same time to perform, there will be no idle processors and therefore no resources to be exchanged between clusters in order to improve performance. This is a design time analysis, but the actual behavior and duration of the tasks in an application will also be influenced by memory latency (cache hit and cache coherence) and communication latency, among other things. Therefore, some tasks can finish much earlier than others even in applications with predicted constant progression.

Table 3.3. Resources required by each application.

Application	Sequence of resources	Progression
Matrix Multiplication	NT, 1, 0	Constant
Motion Estimation	NT, 0	Constant
Mergesort	NT, NT/2, NT/4, …, 1, 0	Exponential
JPEG	NT, 1, 0	Constant

Source: Girão (2013, p. 682).

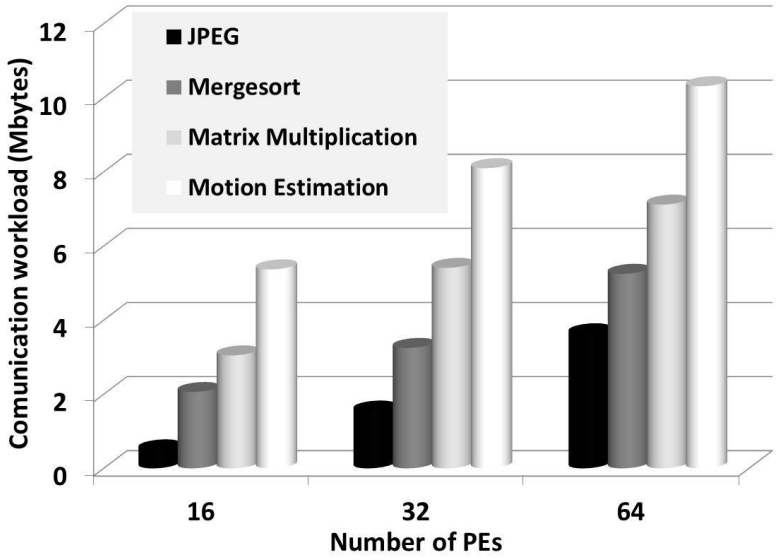
According to this table, the Mergesort algorithm, due to its divide and conquer nature, has tasks finishing at an exponential pace. Meanwhile, Matrix multiplication, Motion Estimation, and JPEG are applications in which the degree of parallelism is almost constant throughout their execution. This means that, for these three applications, if one task finishes, it is quite probable that all tasks of that application will end very soon.

Considering the data inputs for the applications described above,

Figure 3.8 represents their communication workload regarding different numbers of processors. Based on this chart, it is expected that the Motion Estimation algorithm will generate a larger amount of data exchanges.

Experiments evaluate two characteristics: performance, measured by the total execution time of each application, and the overall dynamic energy spent, including processors, network, and memories. For the energy of the processors, a cycle-accurate power simulator (CHEN, 1997) is used. For the network (including buffers, arbiter, crossbar, and links), the Orion library (KAHNG, 2009) is applied, and for the memory and caches the Cacti tool (WILTON, 1996) is used.

Figure 3.8. Communication Workload.



Source: Girão (2013, p. 682).

There are some tradeoffs in the scenario presented here. This is mainly due to the fact that, in order to better explore the resources, tasks must be migrated from overloaded processors in order to distribute the workload and minimize the average execution time. As presented in

Figure 3.8, some applications have lower workload and are expected to finish earlier than others.

Overall, the mechanism of Processor Clustering tends to leverage on processors that become idle because tasks running on it already finished. With that in mind, it is reasonable to expect that the larger applications, like Matrix Multiplication and Motion Estimation, will have their tasks migrating to occupy processors that initially were executing tasks from JPEG and Mergesort applications. On the other hand, it is expected that tasks from JPEG and Mergesort will never migrate to occupy other

clusters. This is important because the task migration is one of the major overheads of this solution. Also, depending on the parallel programming model in which those applications were developed, this migration can be costly. For instance, when shared memory applications migrate, they only transfer their context, whereas distributed memory ones have to transfer the whole contents of the instruction and data memories. At the same time, memory organizations that lead to low-cost migrations (such as shared memory) can also intrinsically have higher memory latency and, therefore, generate higher execution times. The next section explores this tradeoff in this set of applications.

3.5 Results

In order to evaluate the proposed policies above, a set of experiments was performed using as input the four applications detailed in Section 3.4.1. All four applications were allocated in four regions of equal numbers of processors. The performance and energy results consider the total execution of the applications in the system. This means that the execution time goes from the start of the first task until the last one (from all applications) is finished.

Table 3.4 presents the details of the SIMPLE virtual platform setup used in these experiments. The applications were simulated in three different MPSoC sizes (number of cores) for the same amount of tasks per application. An important consequence of this choice is the fact that less cores (16, for instance) would mean more tasks per core. This would increase the established threshold (which, for now, we assume as half of the cores' tasks, as an attempt to equalize the number of tasks among cores) and the costs of task migration. It is important to notice that, using L2 cache sizes of 64Kb per core, parts of the applications' code and data will remain in the external memory (which is considered here as large enough to store all code and data from all applications).

Table 3.4. SIMPLE setup.

Number of processors	16; 32; 64
L1 cache size (data + instruction)	8kb (4Kb + 4Kb)
Block size	64 bytes
L2 cache size (data + instruction)	64Kb (32Kb + 32Kb)
Number of initial tasks per application	32
Technology	0.9 µm

Source: Girão (2013, p. 683).

3.5.1 Results without applying policies

Table 3.5 presents the combinations of programming models and applications. The terminology used to define each combination is in the format *nD-mS*, meaning that, in this specific situation, the MPSoC was running *n* Distributed memory applications and *m* Shared memory applications. Excluding the situations where all applications use the same parallel programming model, the other three groups (3D-1S, 2D-2S, and 1D-3S) can have multiple permutations of applications. The idea here is to investigate all combinations of these applications for each group. It is important to notice that each of these applications is different in terms of communication demands and workload, as shown in

Figure 3.8.

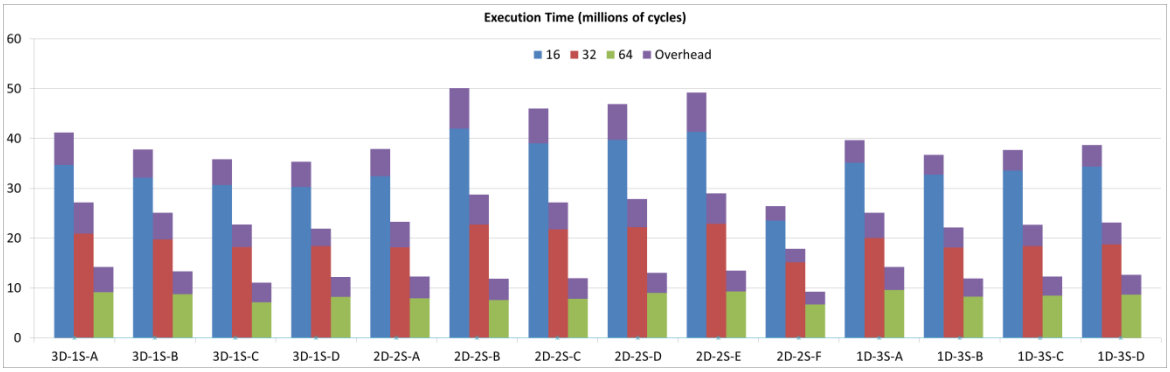
Table 3.5. Combinations of applications according to the parallel programming model.

Grouping	Permutation	Distributed	Shared
3D-1S	A	MM, ME, Mergesort	JPEG
	B	MM, ME, JPEG	Mergesort
	C	MM, Mergesort, JPEG	ME
	D	ME, Mergesort, JPEG	MM
2D-2S	A	MM, JPEG	ME, Mergesort
	B	MM, ME	Mergesort, JPEG
	C	MM, Mergesort	ME, JPEG
	D	ME, JPEG	MM, Mergesort
	E	ME, Mergesort	MM, JPEG
	F	Mergesort, JPEG	MM, ME
1D-3S	A	ME	MM, Mergesort, JPEG
	B	JPEG	MM, ME, Mergesort
	C	Mergesort	MM, ME, JPEG
	D	MM	MM, Mergesort, JPEG

Source: Girão (2013, p. 683).

Figure 3.9 presents the execution time results for all combinations (higher means worst) for 16, 32, and 64 cores. The overhead as indicated represents the amount of time spent on task migrations. Inside each grouping it is possible to see very different results, depending on the combination of applications. In the 3D-1S group, the best combination overall is the one where Motion Estimation uses a Shared Memory environment (3D-1S-C). This is mainly due to the fact that ME has a higher workload and the Shared Memory model provides the fastest task migration. Therefore, the migration overhead is lower.

Figure 3.9. Execution Time results for all combinations.



Source: Girão (2013, p. 683).

In the 2D-2S group, some of the combinations resulted in the worst results. This is due to the fact that, with two applications using each parallel programming model, it is easier to have situations where migration costs will be higher, depending on which applications are developed using these high migration-cost parallel programming models. At the same time, there is also room for situations where the workload-heavy applications were developed in low migration-cost programming models. This is the specific case of the 2D-2S-F case. In this case, smaller applications like Mergesort and

JPEG use the Distributed Memory organization and, therefore, will finish even earlier than the two other applications. Also, the larger applications use the Shared Memory organization and can thus migrate faster. According to the results, the overhead of having higher average memory latency is compensated by having more resources available early on throughout the simulation.

As for the 1D-3S group, the situation is fairly even, but the best results are the ones where the Motion Estimation and Matrix Multiplication applications use the Shared Memory organization (1D-3S-B). This reduces the task migration cost, and the migration happens earlier due to the fact that the smallest application (JPEG) was developed using Distributed Memory organization.

Overall, these results show that, by changing which parallel programming model an application uses, the execution time can vary in up to 50% (the difference between 2D-2S-B and 2D-2S-F).

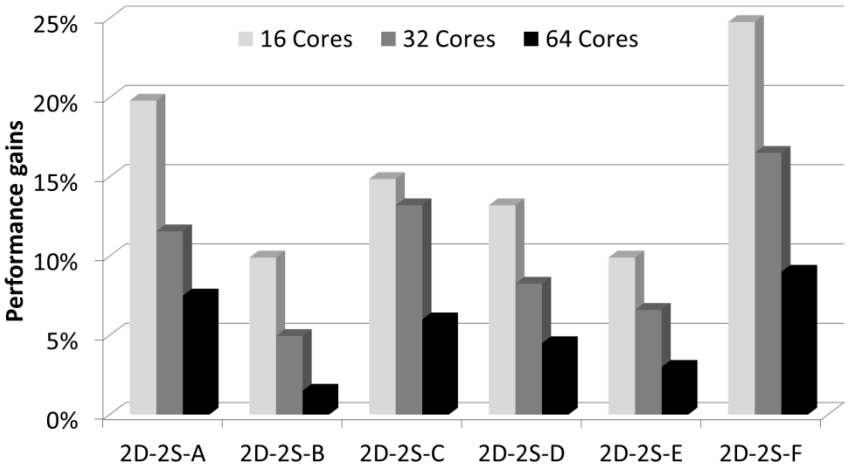
3.5.2 Results on the use of policies

Since the grouping 2D-2S presented the most diverse results, we simulated the use of the three policies only for permutations in this group. These results are a normalized performance improvement based on the results of Figure 3.9, which use a more simple migration policy, as explained in Section 3.2. This means that these are further improvements over the already established gains presented previously.

Figure 3.10 presents the results of performance gains regarding the use of the Shared Variables First policy, when compared to the baseline. It is possible to see a reasonable improvement especially in the cases where the applications with higher workload are programmed using shared variables, creating a situation where the tasks chosen to migrate are the lighter ones. These results show that combinations 2D-2S-A and 2D-2S-F present better results. These two combinations have ME as a shared variable application (as described in Table 3.5), which, as seen in

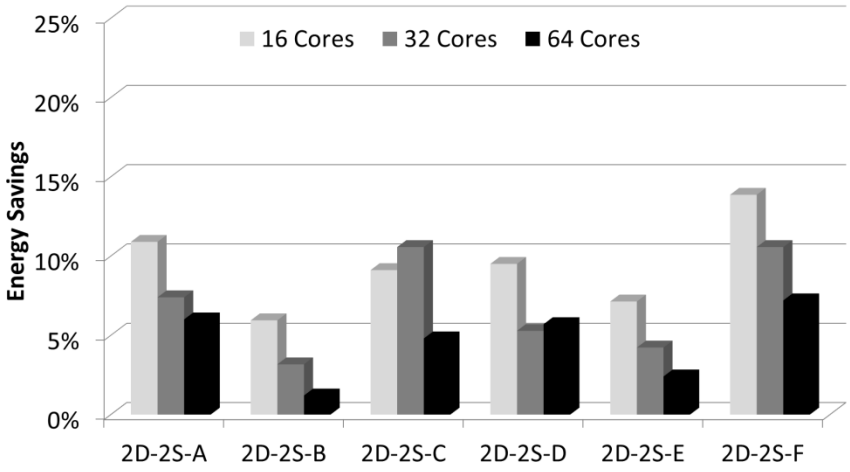
Figure 3.8, is the application with higher workload. The energy savings are shown in Figure 3.11, and it is possible to see small gains that follow the pattern of the performance results. These energy results are smaller due to the fact that the task migration energy overhead is higher than the overhead regarding execution time. Proportionally, the overhead of time spent accessing the memories to send application code and data is smaller than the energy overhead. Also, there is more fluctuation on the energy results for each MPSoC size (2D-2S-C for instance, where the 32-core MPSoC presented better results), showing how sensitive this energy consumption can be.

Figure 3.10. Performance gains for Shared Variables First policy.



Source: Girão (2013, p. 684).

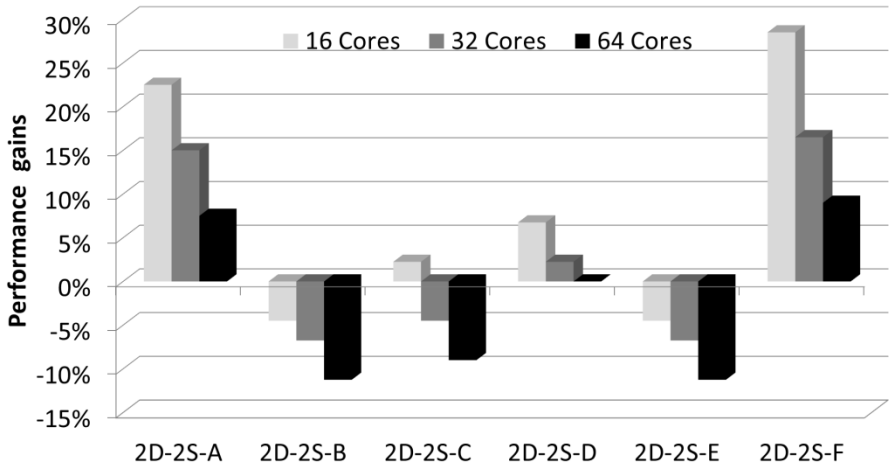
Figure 3.11. Energy savings for Shared Variables First policy.



Source: Girão (2013, p. 684).

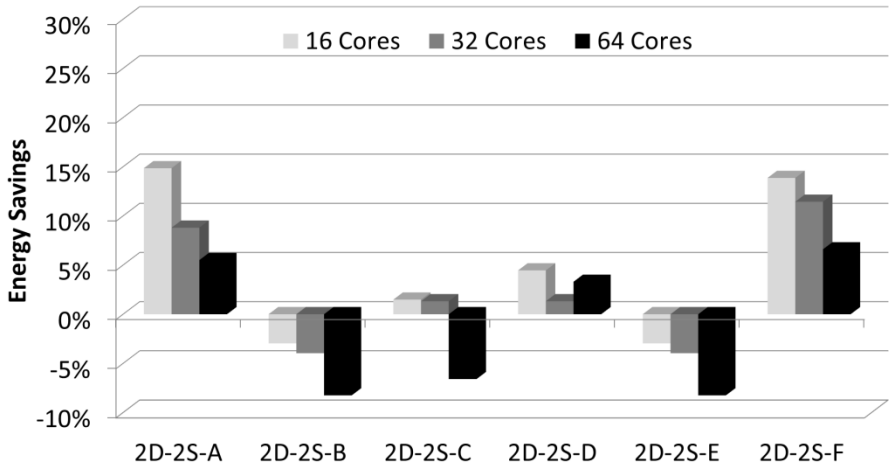
Analyzing the results of the Higher Workload policy, presented in Figure 3.12, it is possible to notice that in some cases the use of this policy hurts the overall performance of the system. This is due to the fact that this policy creates a high priority for applications with higher workload. This may lead to a situation where high workload applications may have higher migration costs depending on their parallel programming paradigm. On the other hand, the results for the 2D-2S-A and 2D-2S-F combinations show that, depending on the combination of cheaper task migration and high workload, the results of this policy can be even better than those of the SVF policy for the same combination. Figure 3.13 presents the energy results, and also, in some combinations, one can notice too much energy spent up to the point that there are energy losses instead of savings. Although the performance results presented better results than the SVF policy in the 2D-2S-F combination, for the energy the results are practically similar.

Figure 3.12. Performance gains for Higher Workload policy.



Source: Girão (2013, p. 684).

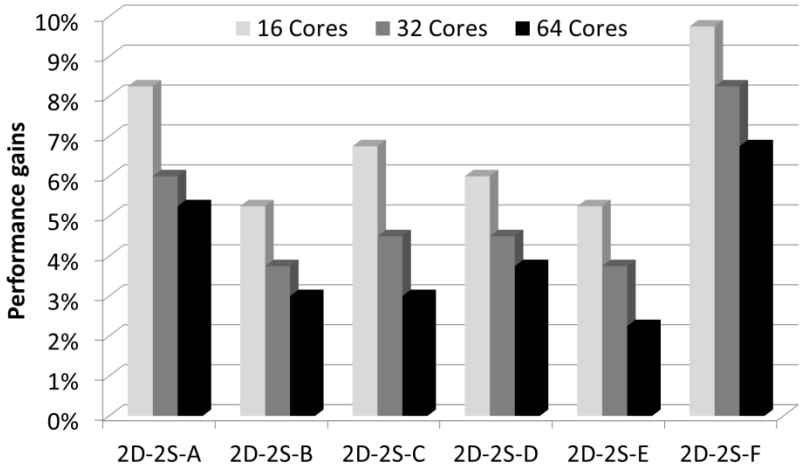
Figure 3.13. Energy savings for Higher Workload policy.



Source: Girão (2013, p. 684).

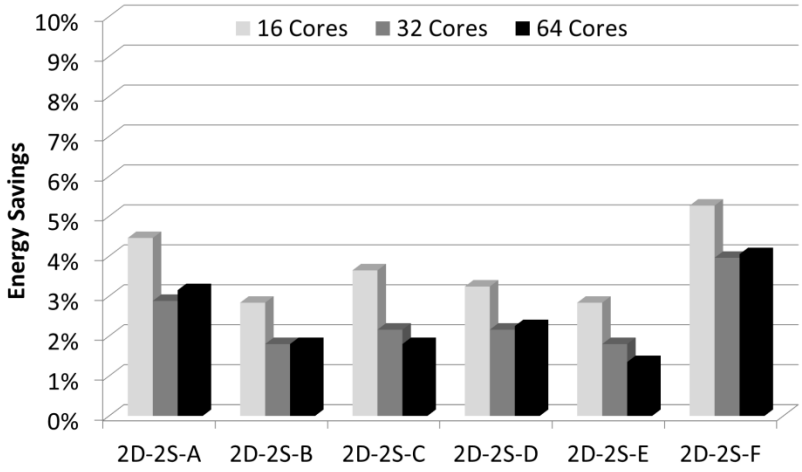
Figure 3.14 presents the results for the Cluster Shape policy. In this case, the best combination (2D-2S-F) does not present better results than the SVF or HWL policies. These small gains suggest that the concerns of a better cluster shape do not impact much on the average minimum distance in a cluster. Two aspects can be pointed to explain this behavior. First, the use of a limited number of neighbor processors (*NH* parameter) already limits the possible damage caused by a bad cluster shape or fragmentation. Also, the MPSoC size does not reach a point where this cluster shape metric could have significant impact. Even for the highest number of cores, the number of tasks being the same (reaching a point of two initial tasks per processor) limits the possibility of spreading the clusters. However, the average results for each policy presented in Table 3.6 shows that this policy still has better results than HWL in situations of 32 and 64 cores. The energy results presented in Figure 3.15 reflect this similar behavior, although the cases with 32 cores and 64 cores present much more similar results.

Figure 3.14. Performance gains for Cluster Shape policy.



Source: Girão (2013, p. 684).

Figure 3.15. Energy savings for Cluster Shape policy.

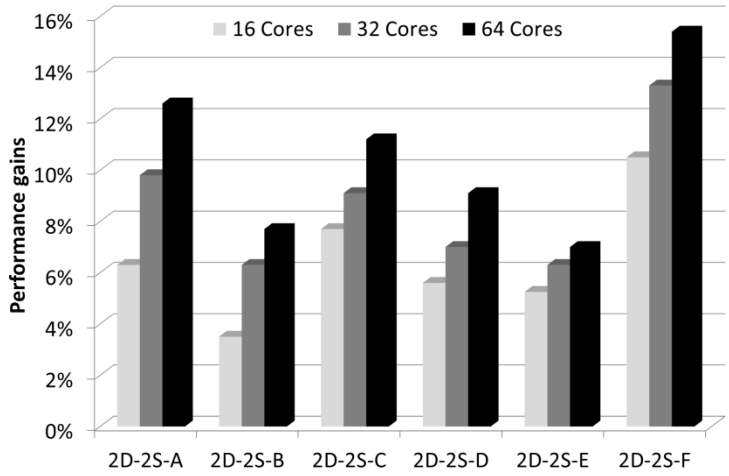


Source: Girão (2013, p. 684).

Figure 3.16 presents the results for the Off-Chip Memory policy. In this case, combination 2D-2S-F still presents the better results. However, differently from the previous policies, the 64-core scenario presents the best results. This is most likely due to the fact that a larger amount of cores increases the average distance between two cores in the NoC. Therefore, the latency resulting from the off-chip memory access has a larger impact on the overall results. By keeping tasks closer to the memory controller, we manage to reduce this latency. Figure 3.17 presents the energy results, and it also presents the same pattern shown in the performance results. Yet, these energy achievements, in most cases, are better than the performance gains, which could be explained by the fact that energy consumption of off-chip memories is proportionally more critical than the access time.

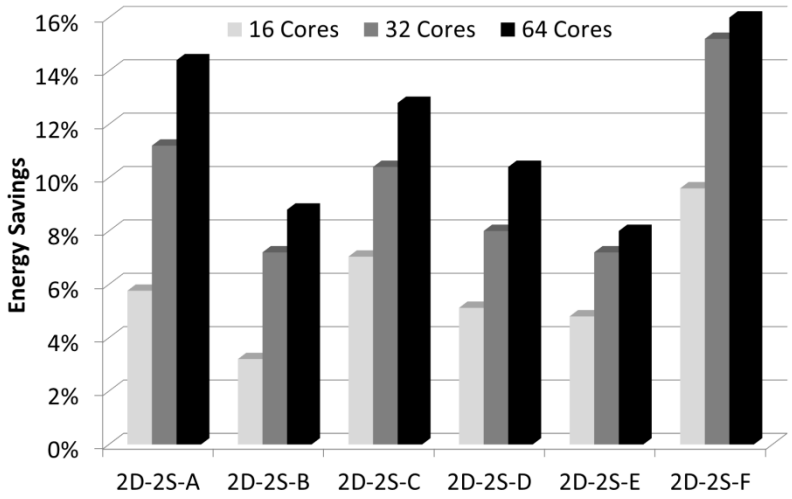
As the Cluster Shape policy presented earlier, this policy could be more effective with a higher *NH* parameter. In this policy, the tendency is for the tasks to move away from the center of the MPSoC towards the edges, where the memory controllers are placed. However, these edges could be better used if the scheduler considered a task migration in a larger area of the chip.

Figure 3.16. Performance gains for Off-Chip Memory policy.



Source: Girão (2013, p. 684).

Figure 3.17. Energy savings for Off-Chip Memory policy.



Source: Girão (2013, p. 684).

As seen in Table 3.6, the best performance gains come from the 16-core experiments (except for the Off-Chip Memory policy). This can be explained by the fact that the execution using 16 cores without the Processor Clustering mechanism offers a much larger improvement window than the 32 or 64-core cases. Particularly in this last one it is hard to find more parallel improvement due to the already high amount of cores per task. The exception is the Off-Chip Memory policy, which improves the performance based on how close the tasks are from the memory controllers on the corners.

Table 3.6. Average Performance gains for all combinations.

Policy	**Average Improvement**		
	16-core	32-core	64-core
Shared Variables First (SVF)	15.4%	10.1%	5.7%
Higher Workload (HWL)	8.5%	2.6%	-2.5%
Cluster Shape (CS)	6.8%	5.1%	4%
Off-Chip Memory	6.4%	8.6%	10.5%

Table 3.7 presents the average energy results. Even though these are small savings, it is important to remember that these are improvements over an already profitable situation, as presented in Section 3.2 with the comparison between this Processor Clustering approach and a regular MPSoC with no task migration for a better use of resources.

Overall, the Shared Variables First policy presented the best results in average. This can be explained by the combination of lighter task migration by the Shared Variables applications and the fact that, typically, the Message Passing applications execute faster. Nonetheless, the other policies also presented interesting results in specific combinations and MPSoC sizes.

Table 3.7. Average Energy savings for all combinations.

Policy	**Average Improvement**		
	16-core	32-core	64-core
Shared Variables First (SVF)	9.4%	6.8%	5.1%
Higher Workload (HWL)	4.7%	2.4%	-1.2%
Cluster Shape (CS)	3.71%	2.5%	2.4%
Off-Chip Memory	5.9%	9.8%	12%

3.6 Final remarks

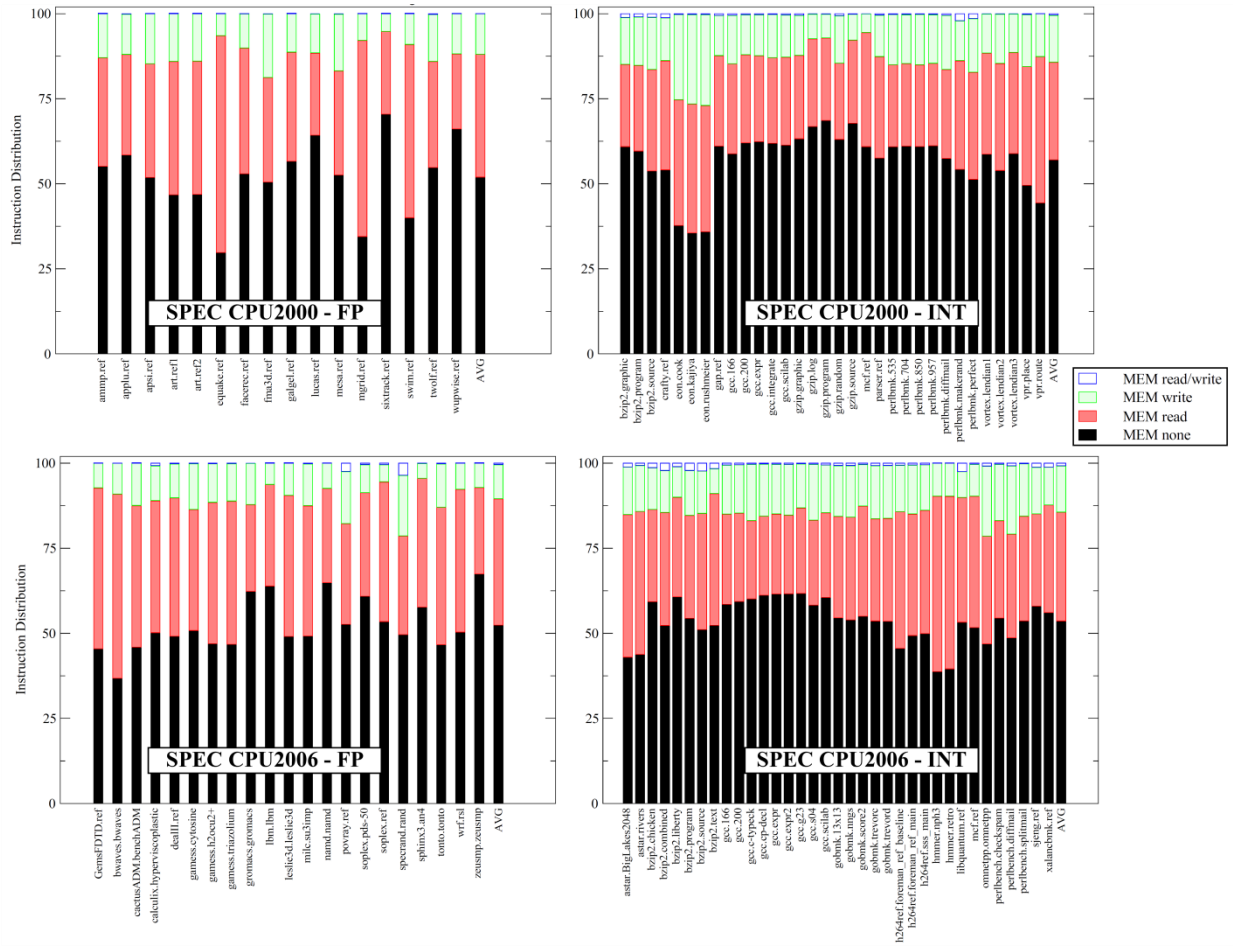
The work presented in this chapter introduces the concept of Processor Clustering, which tries to take advantage of resources with lower amount of tasks in order to increase overall performance. This idea tries to take advantage of a simpler task migration decision mechanism and leverage on intrinsic characteristics of the application and of the hardware platform. This approach also tries to aggregate the applications from the same task together in adjacent cores, in a logical structure known as Virtual Clusters.

The experiments presented here explore an extensive amount of combinations of applications with distinct communication and memory demands. When compared to a system with no sense of opportunity regarding the use of available resources, this approach shows substantial benefits. Further exploration shows that this Processor Clustering can be improved if considering other aspects of the system, such as average network latency, workload and memory latency. Particularly, the system seems to benefit considerably (up to 15% of extra performance gains) when considering the parallel programming models of tasks. This is explained by the fact that these distinct parallel programming models assume a specific memory subsystem, which influences the task migration overhead.

4 MEMORY CLUSTERING

As discussed in the previous chapter, the aggregation of resources as well as their efficient distribution may lead to substantial gains for the overall system. The model of Processor Clustering presented is based on the insight that some characteristics of the application (software level) or of the platform (hardware level) should be taken into account to determine this efficient distribution. However, it is a well-known fact that applications may have different needs of computational requirements and memory requirements. Computational requirements refer to the needs of the application to execute its code faster, for instance using TLP and ILP techniques. Eventually, these computational demands must be addressed by assigning more processor elements (e.g. cores or specialized hardware) to these applications. On the other hand, certain applications manipulate considerably higher amounts of data and, therefore, the use of more memory resources (e.g. L1/L2 caches) would increase their efficiency. Figure 4.1 shows the memory demands of applications from the SPEC benchmark (JALEEL, 2007). By considering the execution of memory access instructions, it is possible to see that the memory requirements can range from 30% up to 70%. With that information it is clear that an efficient memory resource distribution system, in an environment that can run several applications like those, can be advantageous.

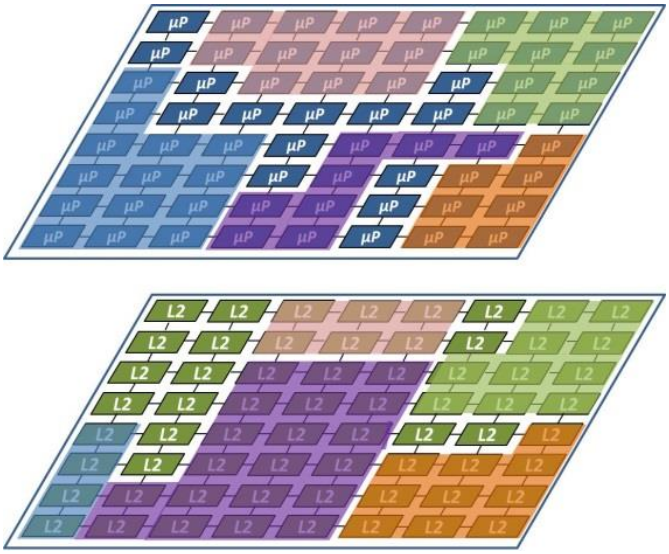
Figure 4.1. Memory requirements from SPEC.



Source: Jaleel (2007, p. 3).

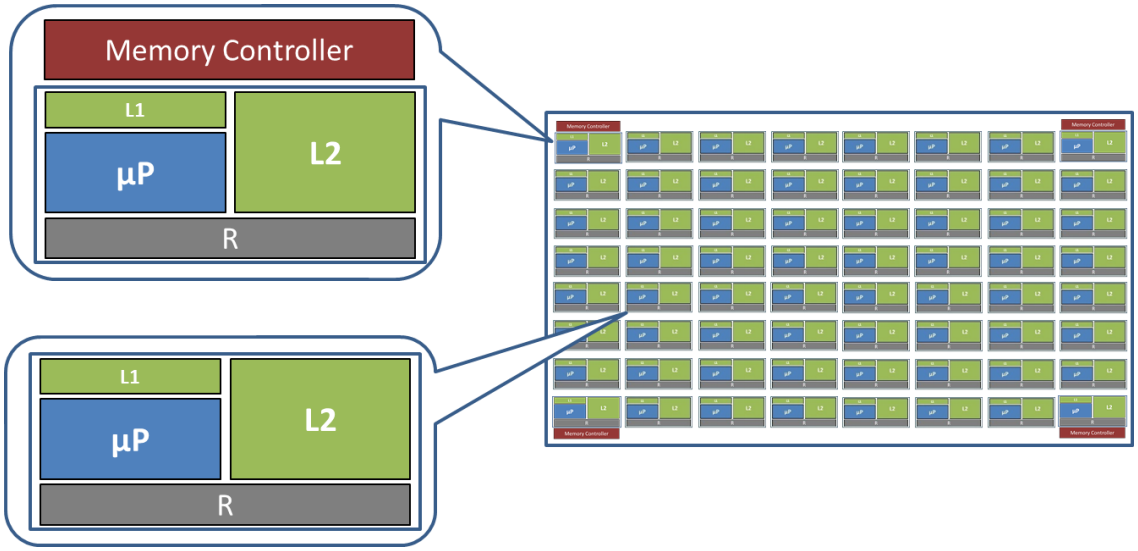
Moreover, the best aggregation of processing and memory resources for improved performance or energy savings may not be the same. Figure 4.2, at the top, shows an example of how clusters may be formed for the executing tasks of one application. At the bottom of the same figure, the memory resources, such as L2 caches, may be arranged in a very different manner, concerning not only the number of resources but also their location in the system.

Figure 4.2. Memory Clustering



Again, let us consider the baseline architecture for this mechanism as an MPSoC whose nodes are as presented in Figure 4.3. In this case, the memory subsystem is also homogeneous and composed of local and private L1 caches and distributed shared L2 caches. Inside the four nodes in the corners, there is also an external memory controller, which has a very important role in this approach.

Figure 4.3. Homogeneous architecture.



The key idea of this methodology is to reserve the L2 memory blocks according to the needs of the application. The intuition here is that providing more memory resources to applications that deal with more data will lead to a faster execution overall. In order to do that, we need to define what does a reservation means and how to do that. To understand how this approach works, one must first understand how the memory subsystem works in this platform as well as the cache coherence mechanism.

The mechanism proposed in this chapter is another cluster-based resource-aware approach. However, as seen in Chapter 2, no papers deal with memory resources in a way of dynamically assigning them to applications according to their memory needs. Some papers consider clustering entire SoCs by means of isolation through hierarchical buses. The problem with this kind of approach is the fact that redistribution of memory resources, if desired, would come with higher latency costs due to the traffic on different levels of the interconnection mechanism. Due to the importance of the memory subsystem to the overall system performance, it is desirable to reduce memory latency for data-intensive applications. In addition, it is very rare for works on resource management on MPSoCs to consider external memory costs as we consider in this proposal. To the extent of our knowledge, this is the first work that explores redistribution policies for memory resources.

4.1 Memory Subsystem and Cache Coherence

When there is a miss in an L1 cache, a block request must be issued to an L2 cache. To perform this task, each node contains a hardware module called CaCoMa (Cache Communication Manager), as already introduced in Figure 3.2 in Chapter 3. One of the functions of this module is to send and receive messages regarding data accesses (blocks coming from L2 to L1 caches) and cache coherence control (writing permission, write-back and invalidation requests). CaCoMa is also responsible for translating memory addresses to NoC addresses in the system. That is why all memory access operations from the L1 cache go through this module. This translation occurs using a small memory called ATA (Address TAble).

In the baseline architecture, the L2 caches are connected to a hardware module responsible for the directory-based cache coherence (GIRÃO, 2007). This module (henceforth called Directory) keeps a table called STA (Status TAble) that gives the status of each block in a certain range of the memory. This table informs in which L1 cache each block currently resides and if they have been requested to be written (i.e. *dirty*) or only read (i.e. *clean*). Based on this information, the Directory performs a different operation when it receives the requests that comes from a CaCoMa. The Directory can issue the following messages:

- **The block itself**: in case of a read request of a clean block or a write request of a non-shared block.
- **An invalidation request followed by the block itself**: in case of a write request of a non-shared clean block.
- **An invalidation request followed by a write permission**: in case of a write request of a shared clean block already owned by the cache that issued the request.
- **A write-back (with invalidation) request followed by the block itself**: in case of a write request of a dirty block.
- **A write-back (without invalidation) request followed by the block itself**: in case of a read request of a dirty block.
- **A write permission**: in case of a write request of a non-shared clean block already owned by the cache that issued the request.

The Directory, as much as CaCoMa, acts in the data link layer generating data packets that must be sent in the network as well as receiving requesting packets that come from other nodes. This Directory, as seen here, can be classified as a Full-Map directory, since it uses n bits per block (where n represents the number of processors) to know which L1 cache has which block, plus one bit to signal the state of this block.

As said before, the Directory keeps information about all the blocks from that L2 cache address ranges whose copies are in other caches spread in the system. With this information, it is capable of enforcing coherence among these L1 caches through control operations. In order to do that, this Directory keeps two tables: STA (Status TAble) and PTA (Processor TAble).

The STA informs which L1 caches hold a copy of a certain block from the memory. This table contains B lines and P+1 columns. B represents the number of blocks in the memory. In this particular case these blocks represent only the memory blocks from the main memory (the external memory) that are in the range of addresses for the L2 connected to the Directory. P is the number of processors, and each column holds the information about which blocks are in the L1 caches. Therefore, column 0 informs which blocks are in the L1 cache connected to processor 0, column 1 the blocks in processor 1, and so on. An extra column indicates if the block in question is dirty or not. A dirty block means that some processor made a write operation on this block, and, therefore, the memory block is not updated, therefore not coherent with its copy from the L2 cache or external memory. This situation enforces the L1 cache that holds the dirty block to perform a write-back operation (in case of a block requisition from another L1 cache). This operation means to send a copy of the modified block back to the L2 cache. Figure 4.4 presents an example of an STA table according to an exemplar instance of a memory subsystem configuration.

Figure 4.4. STA example.

BLOCK	P0	P1	P2	P3	Dirty
0	1	1	1	0	0
1	0	1	0	0	1
2	0	0	1	0	1
3	0	0	1	0	0

Source: Girão (2007, p. 290).

To make possible the corresponding action from the L1 cache, it is necessary for the Directory to know the NoC address of all caches in the system. For that, the Directory keeps another table called PTA (Processor TAble), which relates the network addresses and the L1 cache in this particular address. An example of a PTA table is illustrated in Figure 4.5.

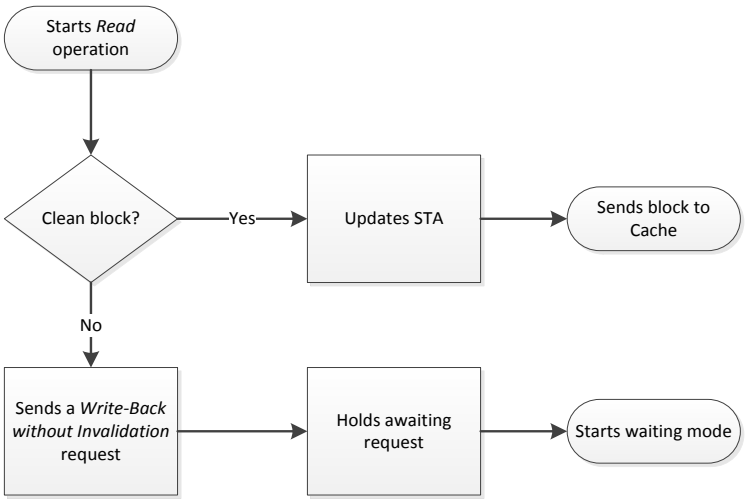
Figure 4.5. PTA example.

Cache	NoC Address
L1 Cache 0	0,0
L1 Cache 1	0,1
L1 Cache 2	1,0
L1 Cache 3	1,1

Source: Girão (2007, p. 290).

For each request coming from an L1 in the system, the Directory makes modifications on the STA and, depending on the situation, makes write-back and invalidation requests. Figure 4.6 presents a flowchart that defines the actions taken by the Directory depending on the state or the block required for reading (read miss). In case the block is clean, the Directory simply updates the STA table indicating that another processor is sharing that block. Finally, the block is sent to the L1 cache. On the other hand, if the required block is dirty, the Directory sends a write-back without invalidation requisition to the L1 cache that modified the block. Once the requisition is sent, the Directory waits for the arrival of the block so it can be sent to the L1 cache that made the original requisition. The update request sent does not include an invalidation request since the cache that originated the request does not intent to modify the block, so both caches can have it.

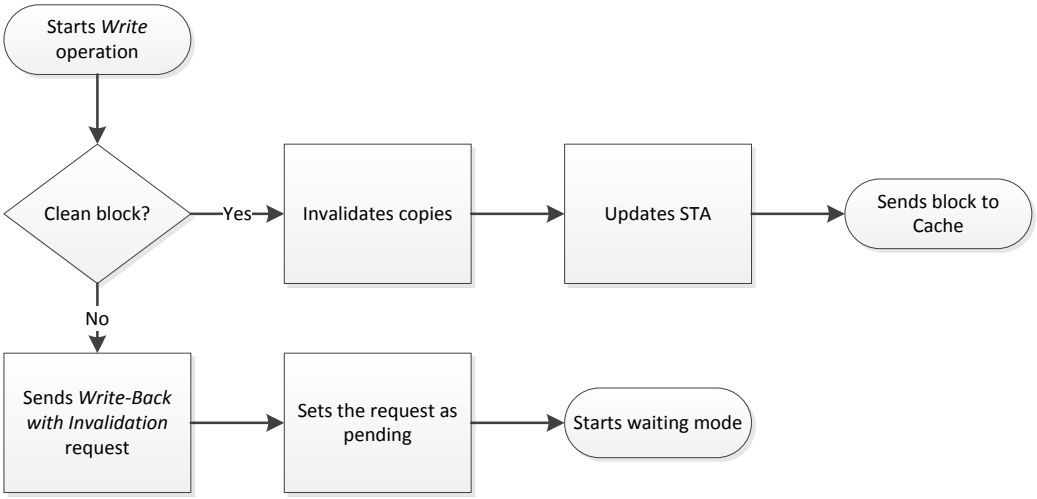
Figure 4.6. Read operation using the Directory.



Source: Girão (2007, p. 290).

In a write miss situation, the Directory again must check if the required block is clean or dirty. In case it is clean, the Directory starts sending N block invalidation requisitions to the N caches that share such block. This is made to guarantee that only one cache at a time can write on a block. Once this requisition packets are sent, the STA is updated indicating that only the L1 cache that required to write on the block has permission to do it and that the status of the block is now dirty. In case of a write requisition on a block that is already dirty, the Directory sends an update request, this time along with an invalidation order to the L1 cache that currently holds the block. Again, this is necessary so the cache coherence is guaranteed. After that, the Directory also waits for the arrival of the block, and, when that happens, the block is updated into the memory and sent to the L1 cache with the initial write miss situation. This whole process is illustrated in Figure 4.7.

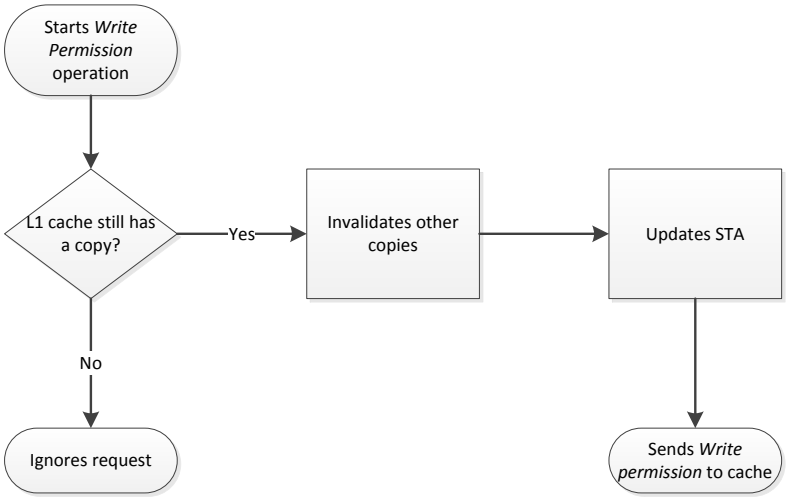
Figure 4.7. Write operation using the Directory.



Source: Girão (2007, p. 290).

Another possible requisition in this cache coherence solution is a write permission request. This situation occurs because after a read miss the caches receive the block only for reading. This means that, if the processor decides to write on a block that is already present in the cache, this cache must require a permission to write on it first. In this case, a packet requesting to modify the block must be sent to the Directory, and this is important because, as explained in previous cases, the Directory takes different actions depending on the status of the block (dirty or clean) on both read and write requisitions. The flowchart in Figure 4.8 presents the actions taken by the Directory in this case.

Figure 4.8. Write-Permission operation.

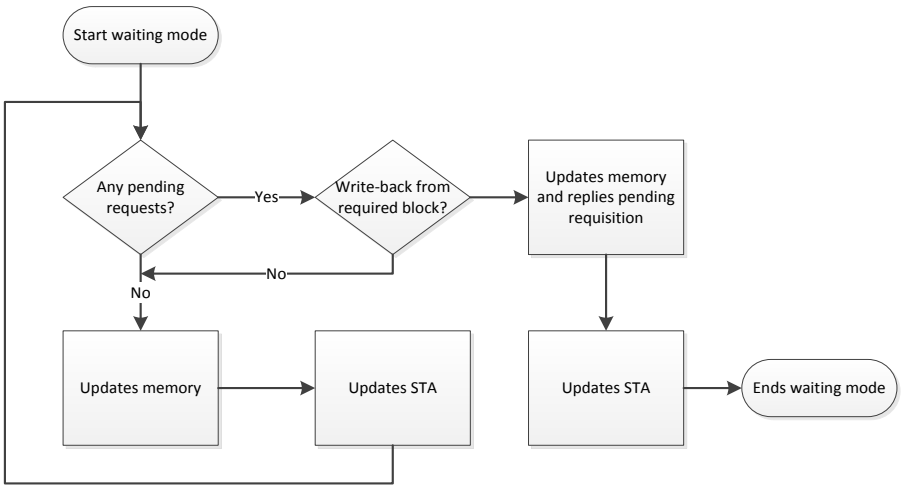


When a write permission request arrives at the Directory, it is necessary to verify if the cache still possesses the block. It is possible that the cache no longer has the block in a case where the write permission request is sent before the arrival of an invalidation request coming from the Directory itself, due to a write request from another cache (as a

result from a write miss situation in this cache). Thus, when the Directory checks that this situation happened, it ignores the write permission request from the cache. In turn, when the cache realizes that it made a write permission request and received an invalidation request, it generates a new request. This time, the request will be for writing the entire block: the write miss request.

As seen before, situations of read and write requisitions (read miss and write miss) may result in write-back requests from the Directory. When this happens, the Directory needs to wait a reply from these requests and do not take any other new requests until the block is received back. This is established in this way to avoid that the Directory starts dealing with a new request without guaranteeing the end of the previous request. If the whole process from the previous request is not finished, there is the possibility of dealing with a non-coherent memory. However, the Directory answers requests as they arrive at the receiving buffer. Therefore, the write-back message that the Directory is waiting for may arrive and not be taken because another request arrived first and this request cannot be taken. This creates a deadlock situation, since the Directory cannot take new requests (which will keep the L1 caches and processors stalled) and the write-back reply cannot move along the receiving buffer. In order to deal with this situation, the Directory works with two buffers: one for write-back packets (which will be given high priority) and another one for the other packets. As presented in Figure 4.9, when the Directory is waiting, it only reads from the priority buffer until the required write-back arrives. Every time a write-back packet is taken, the Directory updates the STA accordingly.

Figure 4.9. Waiting mode when a *Write-Back* request is pending.

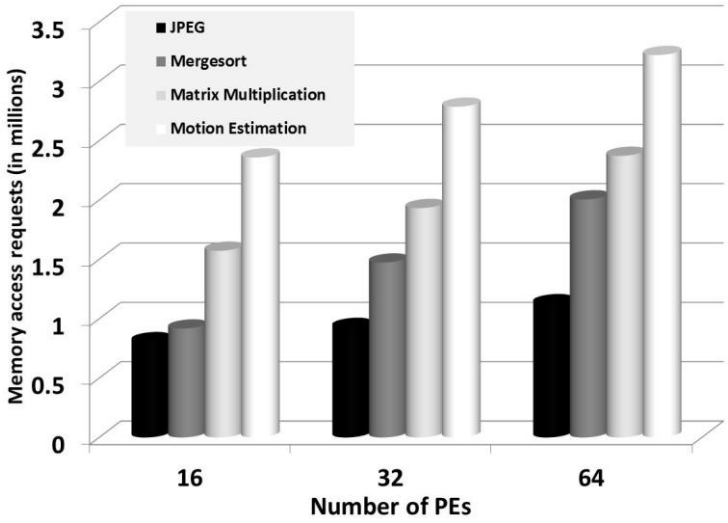


4.2 Proposed mechanism

The process of redistributing the memory resources proposed in this thesis occurs as follows. Since the beginning of the execution, the memory controllers count the number of external memory accesses made by each application cluster. At some point in the system execution, these memory controllers (located in the corners of the MPSoC, as presented in Figure 4.3) use this statistic to define which application has more cache misses. This synchronization step is made when one of these memory controllers (henceforth known as *master controller*) receives messages from the other controllers informing the number of cache misses. After that, this *master controller* uses some redistribution policy (to be detailed in the next sections) to establish how many memory

resources (L2 memory banks) each application cluster should have. This number of memory resources is equal to the percentage of cache misses. For instance, if one application had 50% of all cache misses in the system, it should be awarded with 50% of all memory resources. This strategy is an attempt to reproduce, at runtime, the knowledge presented in Figure 4.10, which shows the amount of memory accesses extracted from an application profile at design time.

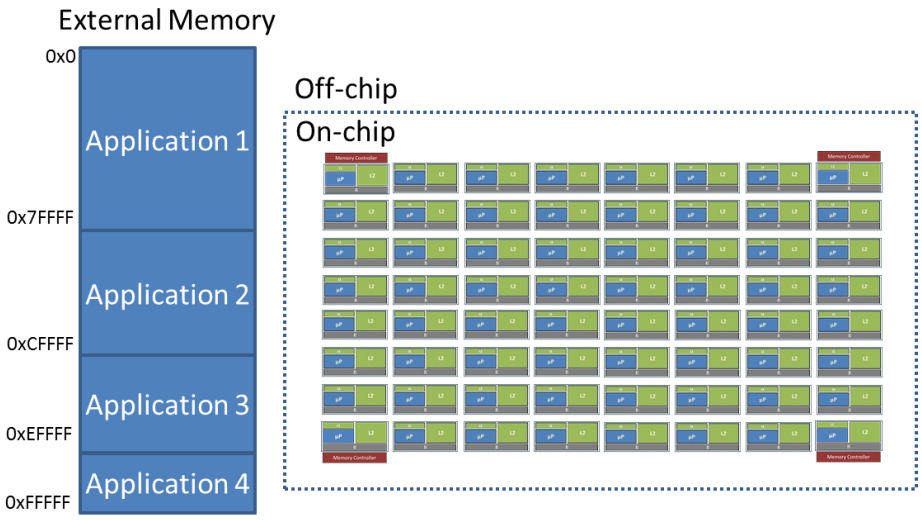
Figure 4.10. Amount of memory access requests per application.



With this information at hand, the memory controller can redistribute the memory resources by sending control messages to all L1 caches in order to change their ATA tables and to specify a new set of L2 caches to which they must now send requisitions when there is a miss. L2 caches must also receive these control messages in order to change the range of addresses they should deal with.

As explained before, at the beginning of the execution, each application cluster has its own set of L2 caches and all L1 caches in this cluster have an ATA table that indicates which range of addresses is possibly available in each L2. The L2 cache controllers also have a notion of which addresses should be placed in its own memory. This is pre-established based on the application data space in the external memory, as we can see in an example in Figure 4.11.

Figure 4.11. Data space for each application in the external memory.

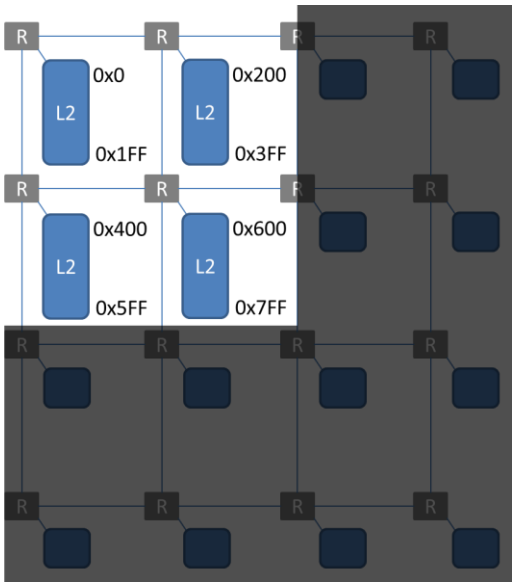


In this example, each application data region is pre-defined. It is important to notice that this information can be quite different from the information on Figure 4.10. That figure presents the amount of data accesses per application. This kind of information can only come to light after an application profiling, which is not always available. The information in Figure 4.11 shows the amount of data sections available for each application, and it is the kind of information available for any operating system at runtime. These two pieces of information are different in a sense that one application can have a certain amount of input data and many memory accesses, depending on how the algorithm works, while another application can have the same amount of input data and yet less memory accesses than the previous one.

Let us assume a small scenario of only four L2 caches for an application. As presented in Figure 4.12, each cache bank can have a certain contiguous range of addresses from the external memory address space. On the left side of the figure, there is a representation of the ATA table of L1 caches in this particular cluster.

Figure 4.12. ATA table and MPSoC before resource distribution.

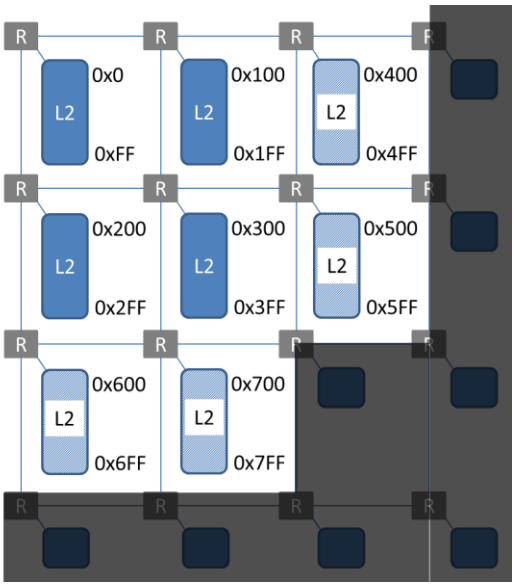
Initial Address	Final Address	NoC Address
0x0	0x1FF	(0,0)
0x200	0x3FF	(1,0)
0x400	0x5FF	(0,1)
0x600	0x7FF	(1,1)



After the memory cluster operation, as illustrated in Figure 4.13, the *master controller* may reach the conclusion that this cluster needs four additional memory banks. After choosing which memory banks in the system shall be aggregated into the cluster (the dashed L2 cache banks represent these recently added resources), the ATA tables must be updated. On the left side of Figure 4.13 the new ATA table is illustrated. It is important to notice that the addresses from the application address space are equally divided among the L2 cache banks. With that said, the range of addresses that each L2 bank can possibly have is smaller. This creates a partial cold-start problem. Let us use as example the L2 cache bank in Figure 4.12, placed in NoC address (0, 0), that is allocated with addresses 0x0 to 0x1FF. In Figure 4.13 we see that this same cache bank can only have addresses from 0 to FF. Therefore, when an L1 cache has a miss on the address 0x155, for instance, it will not send the request to address (0,0) but to address (1,0) instead. Thus, due to this re-alignment in the L2 cache banks, some of these blocks must be marked as invalid and this will cause some overhead in the system.

Figure 4.13. ATA table and MPSoC after resource distribution.

Initial Address	Final Address	NoC Address
0x0	0xFF	(0,0)
0x100	0x1FF	(1,0)
0x200	0x2FF	(0,1)
0x300	0x3FF	(1,1)
0x400	0x4FF	(2,0)
0x500	0x5FF	(2,1)
0x600	0x6FF	(0,2)
0x700	0x7FF	(1,2)



4.3 Redistribution policies

Once established how to modify the number of L2 caches associated to each L1 cache, the next decision for implementing the mechanism of memory clustering regards the moment when to take and give resources (in this case, L2 caches), creating a resizable cluster. Two main approaches are presented here:

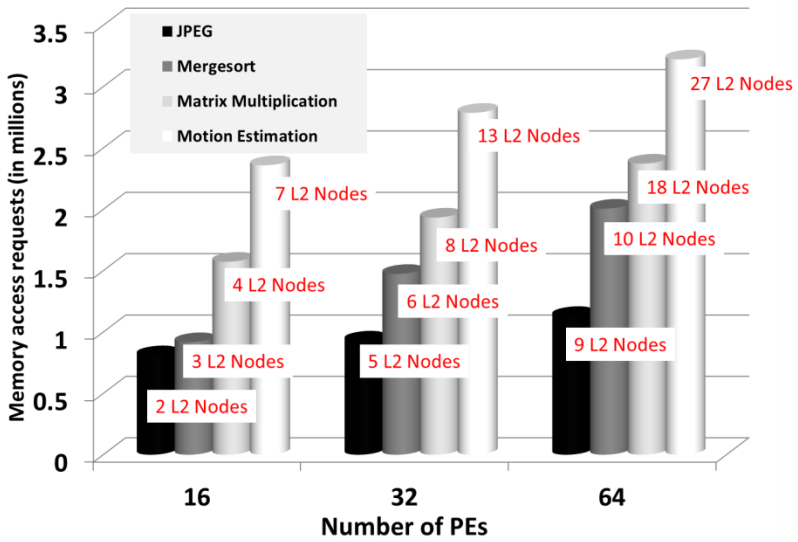
- Pre-defined Distribution
- On-demand Distribution

4.3.1 Pre-defined distribution

The pre-defined distribution is the model used as baseline for the Memory Clustering experimental results. The idea is, based on a design time profiling of the application, to establish beforehand the number of L2 memory modules that each application should have. This distribution takes into account the amount of input data for each application. Since external memory accesses can be very costly, the goal here is to minimize this situation by giving more cache memory for the most data-intensive applications.

The graph on Figure 4.14 presents the amount of memory access for each application and each scenario with different MPSoC sizes (number of nodes). This information can be obtained by profiling the applications at design time. The memory resources (in this case, L2 cache memory banks) can be divided amongst the applications proportionally to the amount of data that each one handles.

Figure 4.14. Pre-defined distribution.



4.3.2 On-demand distribution

However, the information about the memory accesses of each application can only be obtained if the applications are known beforehand. This information may not always be available. Therefore, some runtime approach should be considered. In the On-demand distribution, all clusters initially use the same amount of L2 caches. After a certain amount of cycles, the external memory cache controllers (located on the corners of the MPSoC) exchange information, and, based on the number of external memory accesses, the *master controller* defines which clusters need more memory nodes. Therefore, this master memory controller redefines the ATA tables of L1 caches (potentially, all of them), redistributing the memory resources in the system. The biggest question here is how to define this time at which the memory controllers must take action to redistribute resources. Intuitively, this point in the execution must not be too soon, because what is needed here is a statistical value that characterizes the memory demands for each application. Hence, by taking this information in a too early stage we can have a very premature, and probably wrong, idea of the memory access pattern. On the other hand, by doing it too late in the execution time, the pattern will be well defined, but it may be too late to overcome the overhead caused by the redistribution and manage to gain performance until the end of the applications. Therefore, some trade-off should be considered in this approach. Section 4.5 presents results that can enlighten this problem.

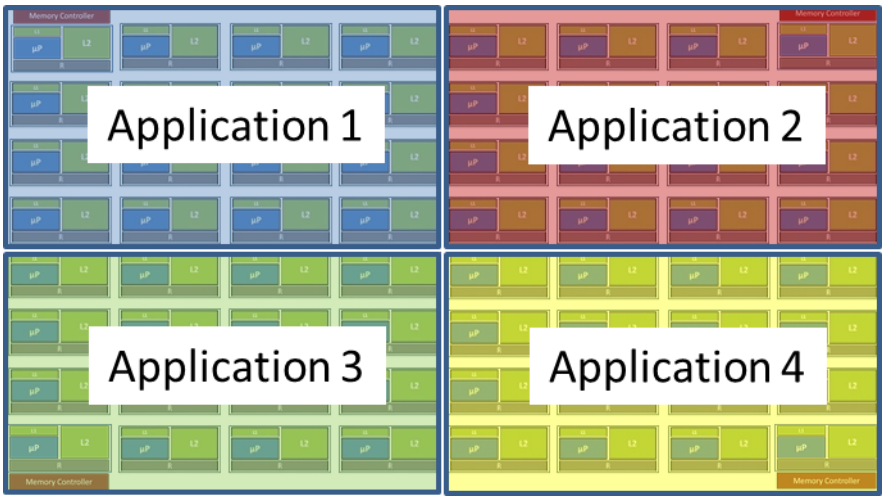
4.4 Distribution Mapping

Another question that should be taken into account is how the memory resources should be redistributed in order to reduce the amount of external memory accesses and try to improve the overall system performance.

First, let us establish the overall system scenario application-wise. In this study, we consider the applications being allocated uniformly and equally throughout the MPSoC, using the same amount of resources. As presented in Figure 4.15 as an example, considering an MPSoC of sixty-four cores, the number of cores to execute each application is sixteen and the tasks are disposed in a square area of cores. What we are trying to do here in this first study is to isolate the problem of clustering the

applications. None of the approaches presented in the previous chapter will be used here. That means that no task will migrate for any reason, even if a processor is completely idle. That being said, the goal here is how to give these applications more memory resources when they need.

Figure 4.15. Applications uniformly distributed.



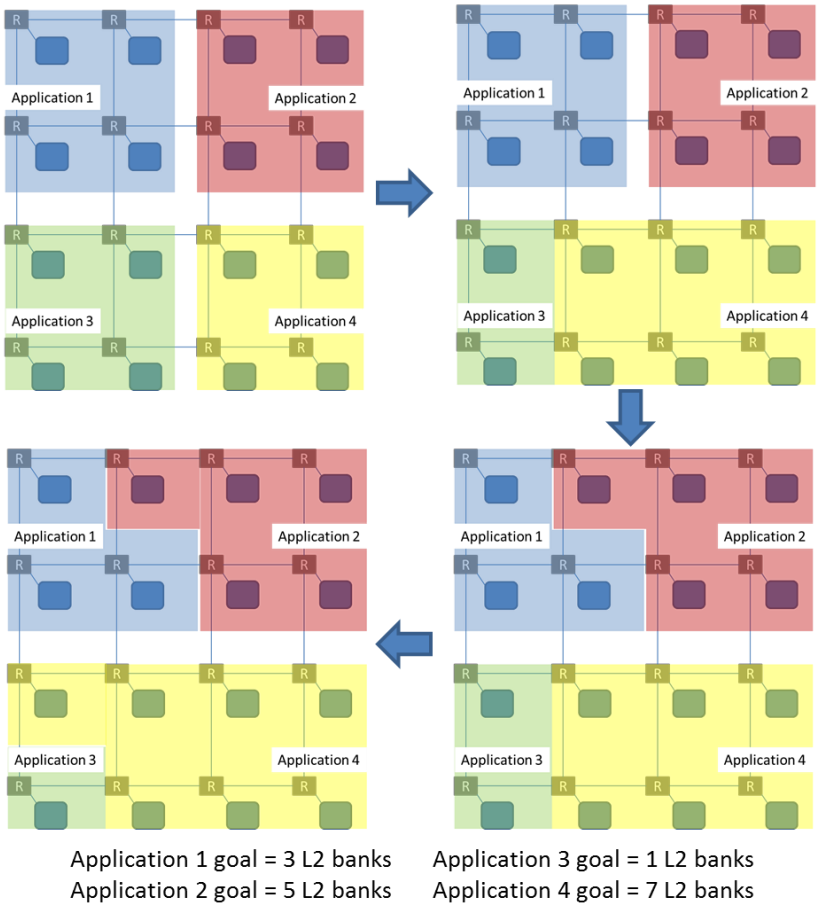
It is unlikely that the hardware resources reach a point where an application does not need the L2 cache anymore. Differently from the study presented in the previous chapter, the use of these resources is not limited by intrinsic computational problems, such as the limitation of parallelism. Therefore, in this approach we simply try to add more L2 caches when applications need it. One of the problems is the fact that the number of L2 memory banks in the system is limited and all tasks from all applications need these resources. Thus, by giving more L2 memory banks to some applications, there is no alternative but to take these resources away from other applications. The hope is to efficiently take the correct amount of L2 caches from applications that need them less and giving them to applications that need them most. The aspect to be explored is which L2 cache must be taken at the moment of the redistribution.

The policy used in this case is similar to the one presented in Chapter 3 (Section 3.3.3): an approach where the main concern is to gather new resources that are closer to the resources originally allocated to the application. It is important to notice that there is a difference between the assumptions of the Cluster Shape approach presented in the Processor Clustering and the one presented in this Memory Clustering mechanism. In the Processor Clustering, the schedulers are constantly looking for overloaded processors that could share the load by migrating tasks to more available resources. This opens up the possibility for fragmentation or scattering of resources to the point of creating a highly irregular topology. For the case of Memory Clustering, the assumption is that the redistribution occurs at once. Therefore, when deciding which resources to give to an application, it is possible to redistribute any memory resources in the system (not only the ones “available”, as in the Processor Clustering).

In this policy, it is reasonable for the applications to exchange resources placed on the edge of their clusters. That condition has the goal of keeping the resources closer together, thus avoiding fragmentation. The master memory controller starts by assigning the new resources to the cluster with more memory needs and gives resources by row (expanding the cluster vertically) or by column (expanding horizontally) towards the cluster that supposed to give up the highest amount of resources. Next, this process is

repeated with the second cluster with more memory needs. This process goes on, with each cluster taking turns on the resource assignment, until the number of resources to be distributed is reached. This expansion initially occurs towards the middle of the MPSoC. Figure 4.16 represents the step-by-step expansion of an example situation. Each application has, initially, the same amount of resources and its "ideal" amount of resources according to the decision taken by the *master controller*. The term "ideal" here means a certain amount of memory resources (i.e. L2 cache banks) that the *master controller*, in this example, has decided to give for each application. This decision is taken in different ways, depending on the policy chosen (as presented in Section 4.3), which is, ultimately, based on the number of external memory requests of each application.

Figure 4.16. Example of memory redistribution.



4.5 Results

As introduced previously, this section will present the results regarding the memory clustering experiments using the Pre-defined and On-demand Distribution. In this second policy the experiments were performed using four distinct checkpoints. These experiments have the goal of evaluating the best moment during the execution of applications to perform a memory distribution, in a way that, after this point, the system can have an overall performance boost. Based on the overall execution time of all applications, the time of the checkpoints chosen to perform the redistribution of the memory subsystem depends on each case. Since we are dealing with different MPSoC sizes, the execution time can vary drastically. Therefore, the checkpoints for the four

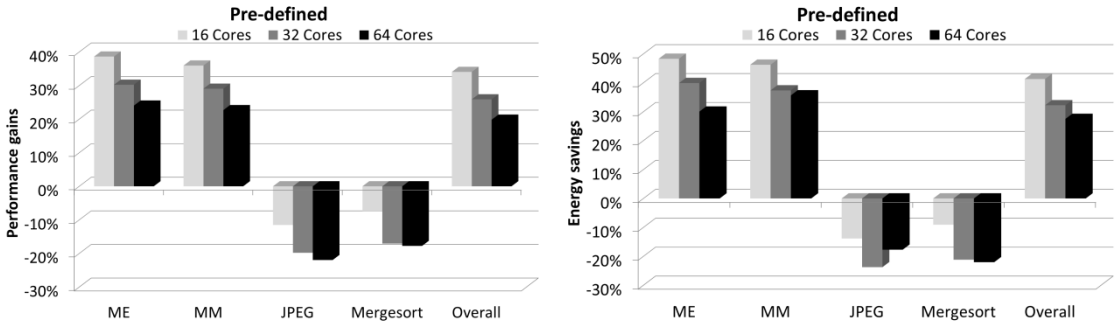
experiments were at 10%, 25% 50% and 75% of the overall execution time in each case. What we are trying to do here is to explore the impact of different values for this parameter, which establishes when this checkpoint must occur.

The experiments presented in this section use as benchmarks the same applications presented in Section 3.4.1. In addition, the MPSoC presents the same configurations listed in Table 3.4.

4.5.1 Pre-defined distribution

Figure 4.17 presents the performance and energy results using the pre-defined distribution. One can notice that the applications with higher input data (Motion Estimation and Matrix Multiplication) have large benefits, as opposed to the JPEG and Mergesort applications. These last two applications actually present a drop in performance, down to 22%. However, there has been an increase in the overall system performance (meaning the amount of time to execute all applications) of 34% in the best case. As for the energy results, they present a very similar pattern when compared to the performance results, reaching up to 40% of savings. Memory Clustering is a very straightforward approach in the sense that the results are exclusively the consequence of a more efficient memory subsystem. Thus, these results are similar because they are affected by the same cause. These performance and energy improvements happen because the larger applications have higher impact on the system. These results show that the Memory Clustering mechanism offers the possibility of a very reasonable improvement. Obviously, due to the overhead discussed previously, these numbers are virtually impossible to achieve. However, these results should give a good basis for understanding how good the Memory Clustering approach is.

Figure 4.17. Pre-defined Distribution performance and energy results.



4.5.2 On-demand distribution

Figure 4.18 presents the results of the on-demand distribution policy considering redistribution checkpoints of 10%, 25%, 50% and 75% of system execution time. By the results of overall performance (last columns in the graphs) it is possible to see that the best results occur when a checkpoint of 50% is used. This means that the utilization of this checkpoint of 50% of the execution time of the applications is enough to determine a reasonable distribution of memory resources in the sytem. When this redistribution checkpoint is too short, as in the case of 10%, it almost does not affect the performance of the system. This seems to be due to fact that this short amount of execution time is not enough to clearly define the applications that need memory resources the most. Therefore, the new amount of resource memories to be assigned to each application does not change much from the initial point (where every application has the same amount of L2 cache banks). The intuition behind it is that fact that initial cold start

makes all applications request cache blocks too frequently. Even the ones that, if compared to other applications, are not data-intensive. Thus, the *master controller* can reach the conclusion (based on the number of requests being almost the same) that all applications have almost the same memory needs. Experiments with highest redistribution checkpoints do not present the best results as well. There are two explanations for this behavior. First, when leaving the redistribution to occur that late in the system, the applications that needed more memory resources were not prioritized for the most part of the execution. This would have a considerable impact on the overall performance. Second, at 75% of execution time, there is not much time left to compensate the overhead caused by the memory redistribution. In the future, we intend to further investigate these possible causes by evaluating the number of cache misses throughout the execution of applications in the system. These cache misses and the overall memory latency before and after the memory redistribution can shed a light on the consequences of choosing a certain redistribution checkpoint. Overall, one can notice that the results have a similar pattern as in the Pre-defined distribution: it favors the alocation of memory resources to Motion Estimation and Matrix Multiplication applications over JPEG and Mergesort. Moreover, it is important to notice that the Motion Estimation and Matrix Multiplication have a higher impact on the overal performance. This happens because as these are the applications that take the longest to finish, they are good targets for any optimization. If they finish faster, the average execution time of the system decreases.

Figure 4.18. Performance results for *On-demand* Distribution.

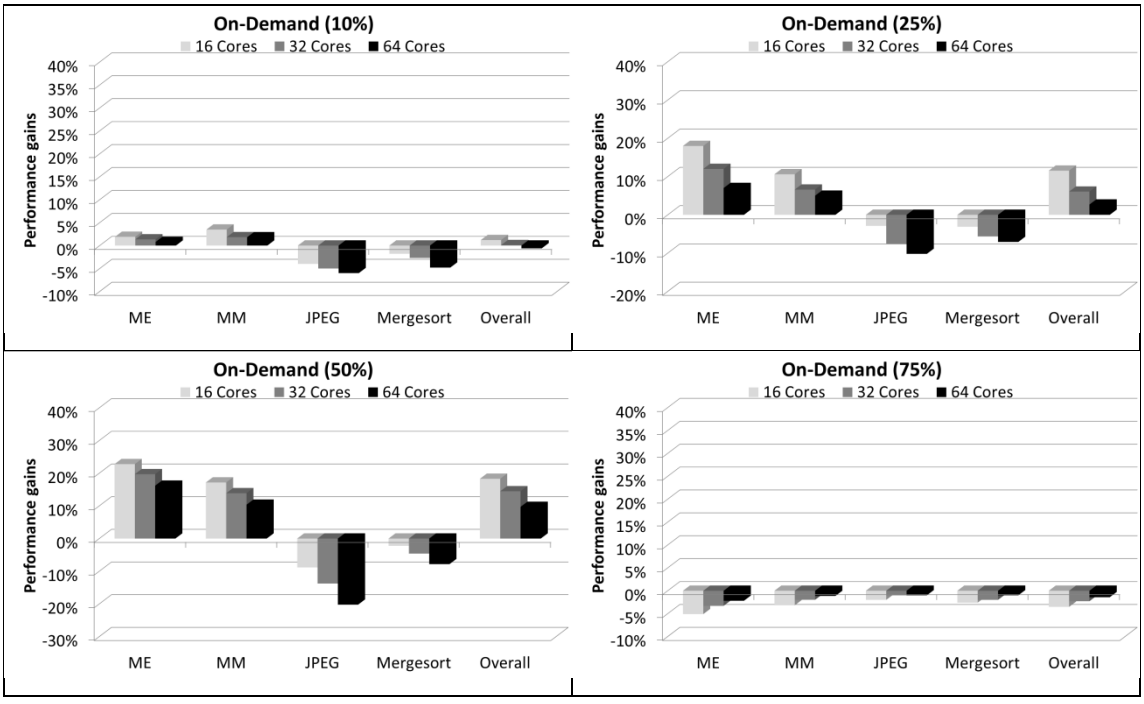
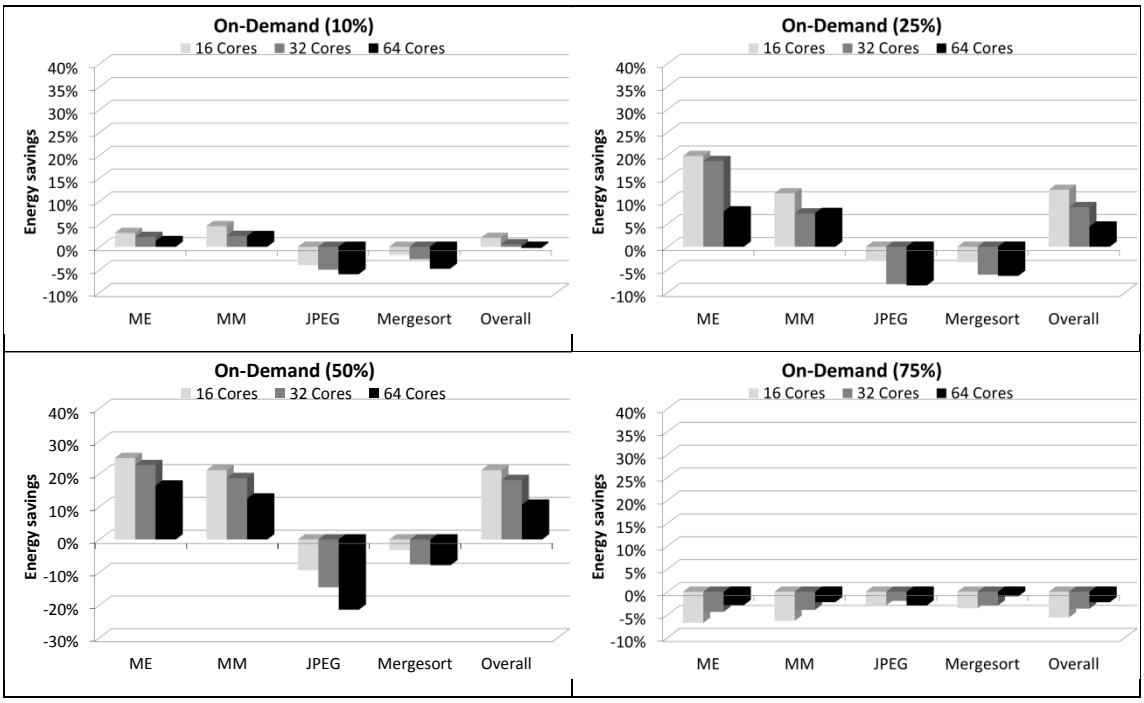


Figure 4.19 presents the energy results for the on-demand distribution. As in the case of the pre-defined distribution, the energy results follow a similar pattern when compared to the performance results. There are slightly higher energy savings than performance gains, probably due to the higher energy costs of accessing the external memory if compared to the time access penalty. Conversely, the penalty when using a checkpoint at 75% of execution is higher.

Figure 4.19. Energy results for *On-demand* Distribution.

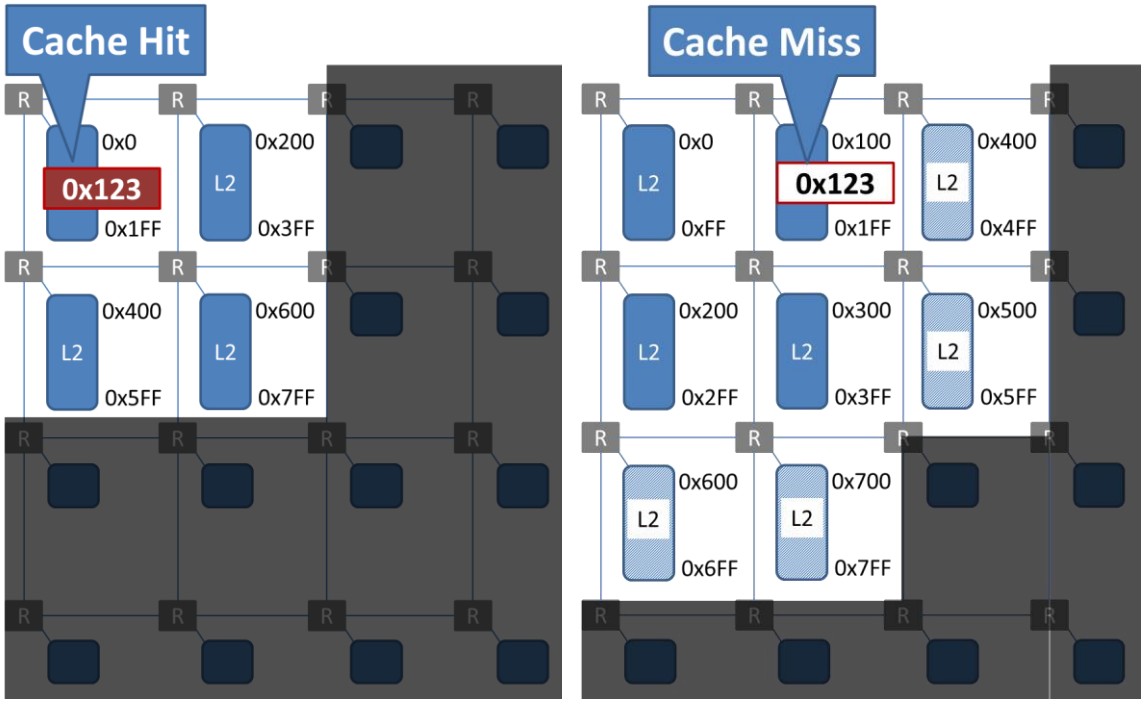


4.5.3 Improving memory redistribution

As described in Section 4.2, the proposed memory redistribution mechanism has the side-effect of creating potential cold-start behavior in the L2 memories. As explained, this is due to the fact that the redistribution process redefines the address range of each physical L2 module. When that happens, some blocks placed in those memories are no longer valid since they do not fall into the new address range. This situation represents the main overhead of the proposed solution.

Figure 4.20 presents an example to help explain how this side-effect of redistribution creates an overhead. Considering the same example presented in Figure 4.12 and Figure 4.13, the redistribution will double the number of L2 cache modules from four (Figure 4.20a) to eight (Figure 4.20b). Now, let us assume that a processor in this cluster needs to access memory address 0x123. In this case, before the redistribution, the physical address would be present in the L2 module located in NoC address (0,0) (holding address range 0x0 – 0x1FF). After redistribution, this same address 0x123 would have to be present in NoC address (1,0). At this point, it is important to remember that this is a logical redistribution. Thus, the content of address 0x123 will not be moved from L2 cache (0,0) to L2 cache (1,0). In this scenario, the processor will now request the data from the L2 module located in NoC address (1,0) which will lead to a cache miss.

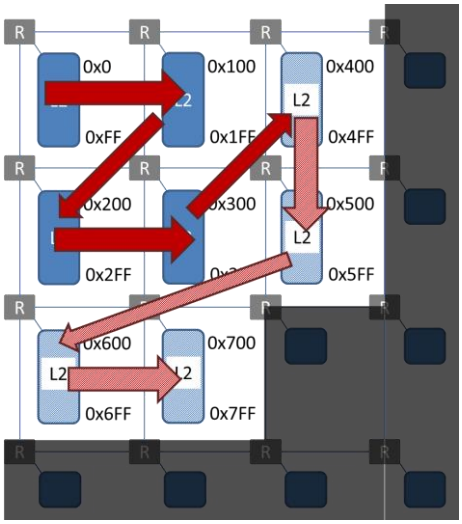
Figure 4.20. Cold start effect after redistribution.



In order to reduce the impact of this cold-start on the overall execution, there is a smarter way to choose the redistribution.

In the example given, the redistribution algorithm first updates the address range of the L2 modules that currently belong to the cluster (NoC Addresses (0,0), (1,0), (0,1) and (1,1)). After that, the new L2 modules joining the cluster are assigned with the rest of the address ranges in order of NoC Addresses from lowest to highest (NoC Addresses (2,0), (2,1), (0,2) and (1,2)). This sequence is indicated in Figure 4.21 (hashed arrows represent the sequence of assignment for new modules).

Figure 4.21. Sequence of assignments in two steps (current modules, then new modules)

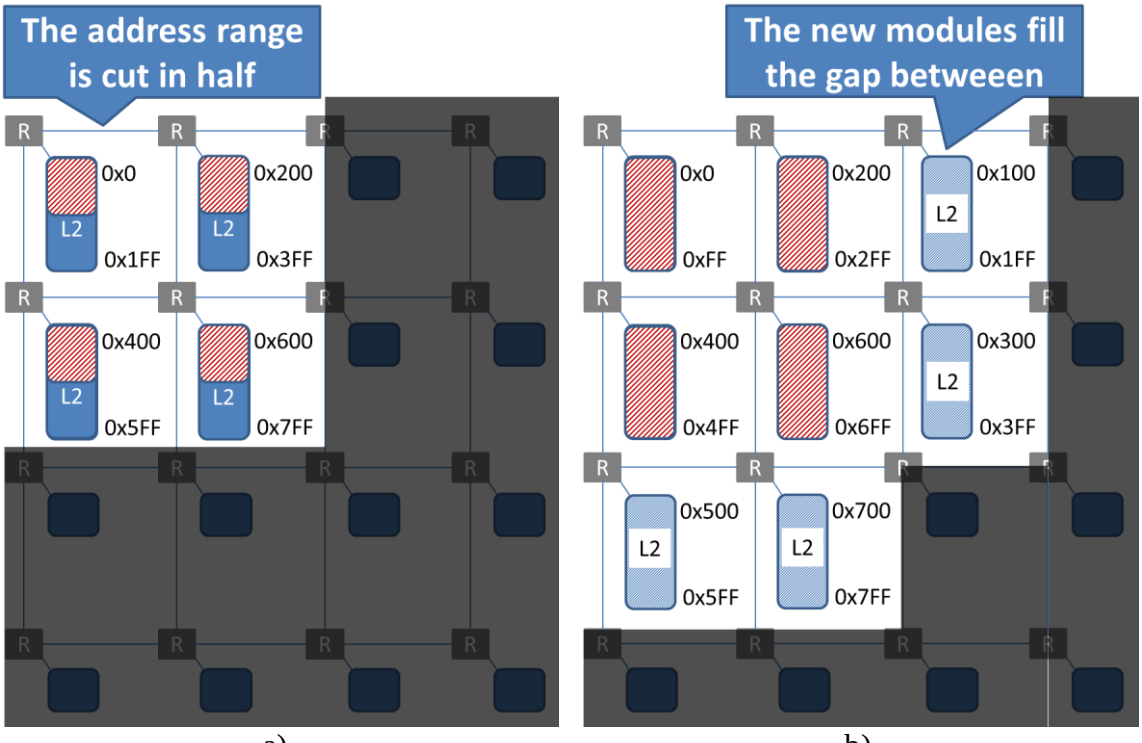


A new algorithm to assign the address range to the L2 cache modules in the cluster could reduce the number of cache misses. Observing Figure 4.20, it is possible to notice

that some addresses in the L2 cache located on NoC address 0,0 do not change their physical location after redistribution. That happens because the algorithm divides the total address space equally among the L2 caches. In this particular case, each one would have their address range cut in half. Therefore, the L2 cache in 0,0 will still have the first half of addresses (from 0x0 to 0xFF). By forcing this pattern to each existing L2 cache in the cluster, it is possible to diminish the number of addresses that will change from one physical module to another.

Figure 4.22 presents a new assigning pattern. In this pattern, each existing L2 cache in the cluster would still have part of the original cache address range. Hence, processor requests for these particular addresses would not incur in cache misses. Using the redistribution example presented in Figure 4.20, the assigning pattern would have to divide each address range of the four original L2 caches (Figure 4.22a) in two. The L2 cache in 0,0 would still have the address range 0x0 – 0xFF. However, the L2 cache in 1,0 would not start as 0x100. We could have a subset of the original address range (0x200 – 0x3FF) in order to improve the chances of cache hit. Therefore, the address range for this module would be 0x200 – 0x2FF (again, half of the original one). The same logic would apply to the other two original L2 cache modules. As for the new L2 cache modules (Figure 4.22b), there is no reason to be careful with the address range assigned to them. The reason is that these modules came from a different cluster and, therefore, contain addresses from a different logical address space.

Figure 4.22. New assigning algorithm.



a) b)

Figure 4.23 and Figure 4.24 present some new results considering the use of this new assignment pattern. The results presented here use the *On-Demand* Distribution at the 50% checkpoint, which was the one presenting better results as presented in Section 4.5.2. In these results it is possible to see an improvement of 5% in performance and 7% on energy savings, considering the overall execution of all applications. Also, one can notice that this new algorithm affects only the applications that benefit from the

redistribution of memory resources (in this case, Motion Estimation and Matrix Multiplication).

These are relevant results when considering that this was achieved only by changing the assignment algorithm. It is important to notice that the improvement provided by this assignment pattern is inversely proportional to the number of new memory modules aggregated. With more memory modules the address range of each module will be smaller and the number of addresses that could be "saved" would be smaller as well.

Figure 4.23. Performance results for *On-demand* Distribution with a new assignment pattern.

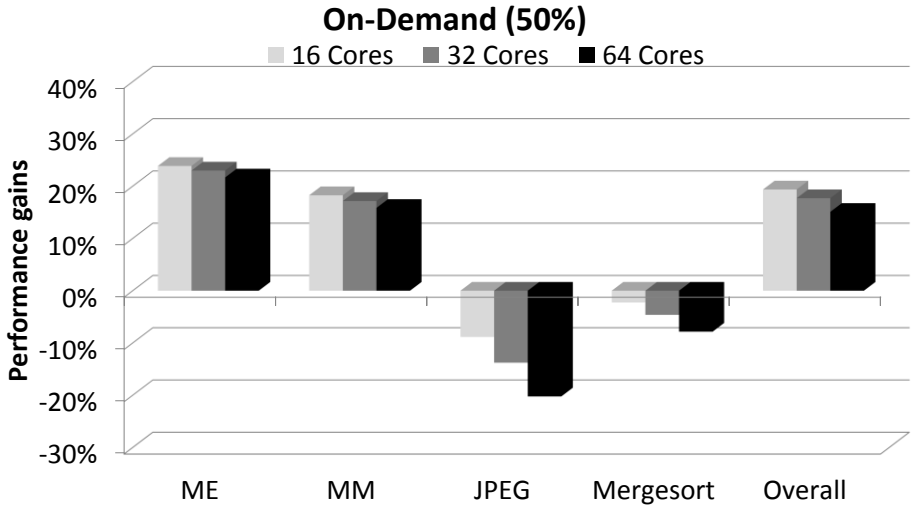
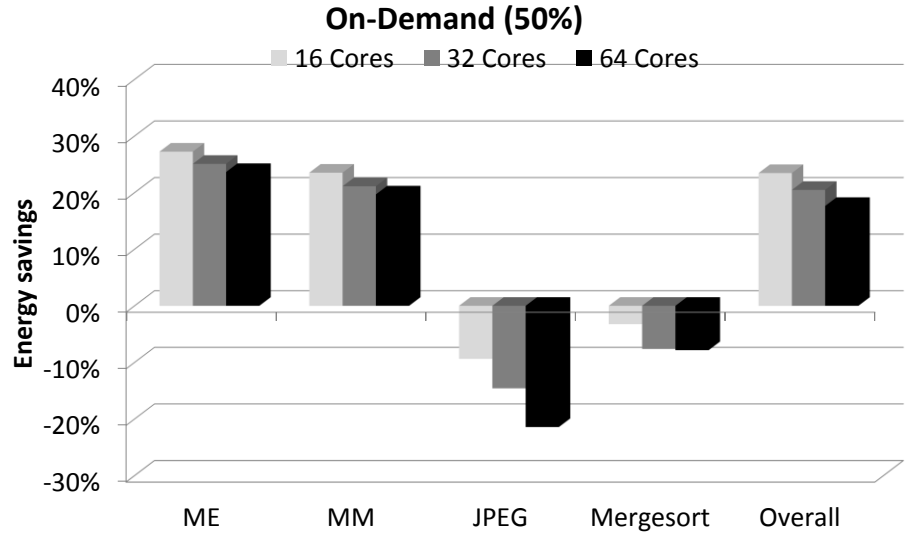


Figure 4.24. Energy results for *On-demand* Distribution with a new assignment pattern.



This algorithm could be more elaborated if one considers having the information of which addresses are present in the cache and where in the address range they are located. This would help to make the decision on which subset of the address space to "save". For instance, in Figure 4.20, if the L2 cache module located in 0,0 (before redistribution) had more cache lines in the address range 0x100 – 0x1FF, the best decision would be to assign to it this address range, instead of 0x0 – 0xFF.

As explained, the gains are dependent on the number of new cache modules aggregated. So, the decision (made by the memory controller) regarding how many memory modules to give to a certain cluster could take that behavior into account.

4.6 Using Processor Clustering

As presented in this chapter, memory clustering is a mechanism that tries to redistribute memory modules to clusters according to their needs. This concept is very similar to Processor Clustering, as presented in the previous chapter. Essentially, these mechanisms try to manage the distribution of resources using some sort of fairness based on aspects of software or hardware level.

A noticeable feature of these approaches is the fact that they can be seen as independent mechanisms, since one of them deals with processors as resources, whereas the second one deals with the memory subsystem. With that in mind, this section presents initial experiments using both mechanisms in a single environment. The idea is to use policies based on this new scenario, where applications can migrate from one point to another in the system and the memory subsystem available may also change at runtime.

Based on the policies presented in Section 3.3, we use slightly modified versions in order to deal with the new scenario of using memory clustering concurrently.

As explained at the beginning of this chapter, the concept of memory clustering was designed specifically to be used in a distributed shared memory environment. At this moment, this concept is not fit for private and distributed memory architecture. For that reason, the key idea behind the Shared Variables First (presented in Section 3.3.1) is not suitable anymore. Therefore, we shall not use its principles as a police.

Again, all experiments presented in this section use as benchmarks the same applications presented in Section 3.4.1 and MPSoC configurations presented in Table 3.4. The results are presented as performance gains and energy savings over the execution of the same applications without the use of any clustering mechanisms.

4.6.1 Higher Workload (HWL)

The Higher Workload policy presented in Section 3.3.2 is very straightforward and considers that some external information determines beforehand the amount of workload for each application. In this case, we do not consider that to be the exact absolute amount of workload but simply a ranking of the applications based on their workload. This principle can also be applied to the scenario using both processor clustering and memory clustering. At the processor level, we can apply this policy directly. At the memory clustering level, we can use it to help determine the memory redistribution rank, i.e. which application should take more memory nodes and which one should give them up. In a similar way, the memory clustering mechanism can use external information estimating the amount of data workload used by each application. By using both metrics, the expectation is that the application execution will improve.

Figure 4.25 presents the results of the Higher Workload policy combining both clustering approaches. These results show that HWL improves the performance overall by giving the most computational complex applications (Motion Estimation and Matrix Multiplication) higher priority. The memory clustering redistribution takes away some memory resources from the JPEG and Mergesort applications, but the improvement at the processor layer seems to compensate this lack of memory resources at least to the

point of not causing performance loss. The energy results presented in Figure 4.26 also present the same behavior as expected. Also, the energy savings on 32-core architecture seems to be much closer to the 16-core than to the 64-core.

Figure 4.25. Performance results for HWL using both Processor and Memory Clustering.

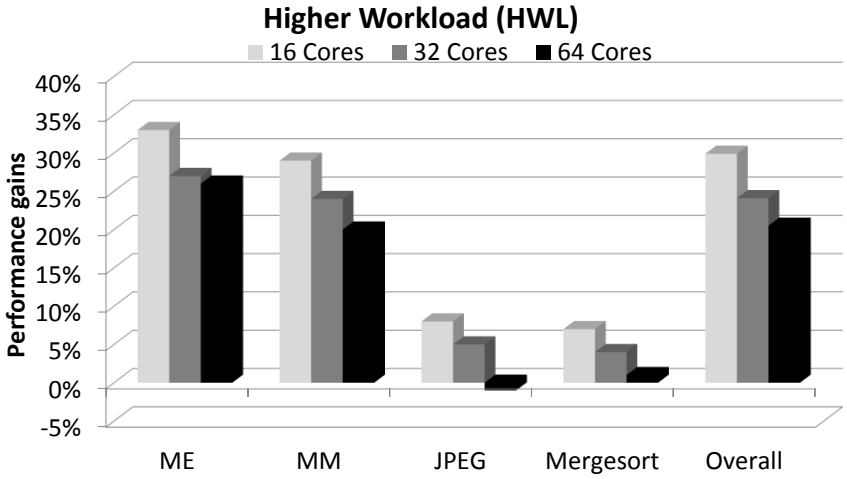
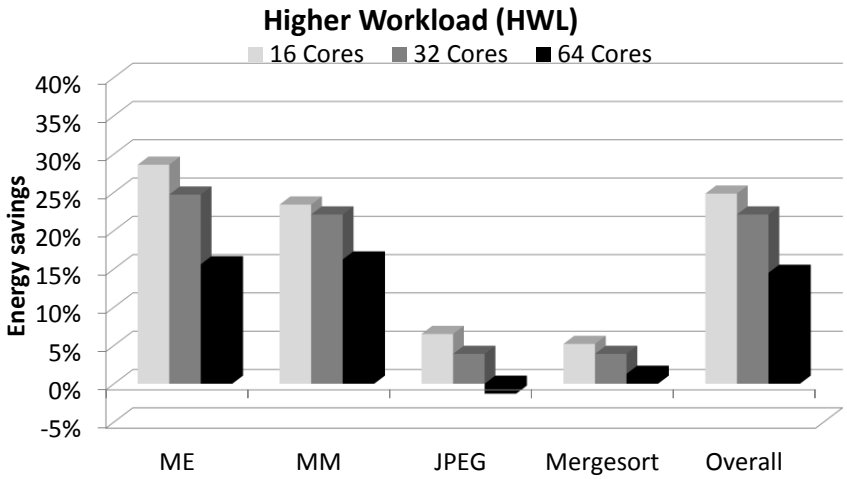


Figure 4.26. Energy results for HWL using both Processor and Memory Clustering.



4.6.2 Cluster Shape (CS)

The Cluster Shape policy presented in Section 3.3.3 has the key idea of keeping the aggregation of resources closer together. The metric to be achieved here is to minimize the average distance between nodes of the same cluster. At Section 3.3.3 we debated this notion taking into account only the distance between processors. Now, we must consider the memory resources as well. Thus, the concept of cluster shape can be applied independently in both layers. The memory controller must also keep track of the memory cluster expanded area. The idea is the same as presented in Table 3.2. In that case, the scheduler has a notion of the covered area of a processor cluster. If the cluster has a large area (given by the Higher and Lower X and Y fields) but few processors, it is considered to be sparse, or low in density. However, a more dense aggregation of

cores (more resources in the given area) is a good metric because it guarantees closeness and minimum average distance (denser areas tend to become squares). This same idea is applied to the memory cluster and, now, the memory controller keeps track of the density of the clusters.

Figure 4.27 and Figure 4.28 show the results for the cluster shape policy used on both the processor and memory layers. As presented, the CS policy does not present as good results as the HWL one. Applications with higher memory demands (ME and MM) do not seem to take full advantage of the resources as expected. The overall gains are good and there are fewer variations on the results amongst the applications than in the HWL policy.

Figure 4.27. Performance results for CS using both Processor and Memory Clustering.

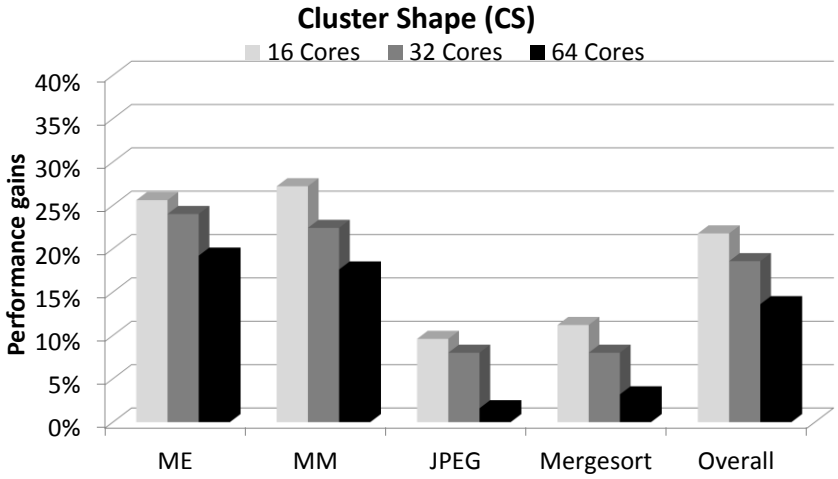
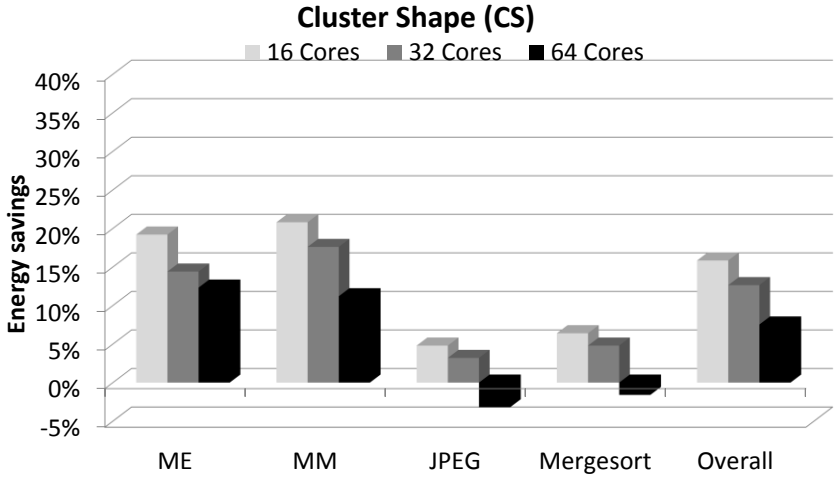


Figure 4.28. Energy results for CS using both Processor and Memory Clustering.



4.6.3 Off-Chip Memory (OCM)

As presented in Section 3.3.4, the key idea of the Off-Chip Memory policy is to diminish the distance between the cluster and the memory controllers located on the corners of the chip. The original idea of this policy already considered the effort taken by the memory hierarchy in the case of a cache miss. In this new joint scenario, this

principle will be also used in the cluster memory, which makes very much sense since this layer will benefit the most from a smaller latency to access the memory controller.

As presented in Section 3.5.2 (Figure 3.16 and Figure 3.17) the Off-chip Memory policy presents better results as the number of cores increases. This is due to the fact that larger NoCs (with higher number of cores) lead to higher average distance of every node to the corners (where the memory controllers are located). The Off-chip Memory policy tries to decrease this distance and, therefore, it is more beneficial for higher number of cores. As presented in Figure 4.29 and Figure 4.30, this behavior impacts the overall results in an interesting manner. Even though the OCM at the processor-level leads to better results for the 64-core scenario, the memory redistribution (as presented in Figure 4.18 and Figure 4.19) greatly improves the 16-core architecture. The results below show the consequence of some compensation between the two behaviors. Overall, the OCM presents better results for the 32-core architecture and there is lower variation among the applications.

Figure 4.29. Performance results for OCM using both Processor and Memory Clustering.

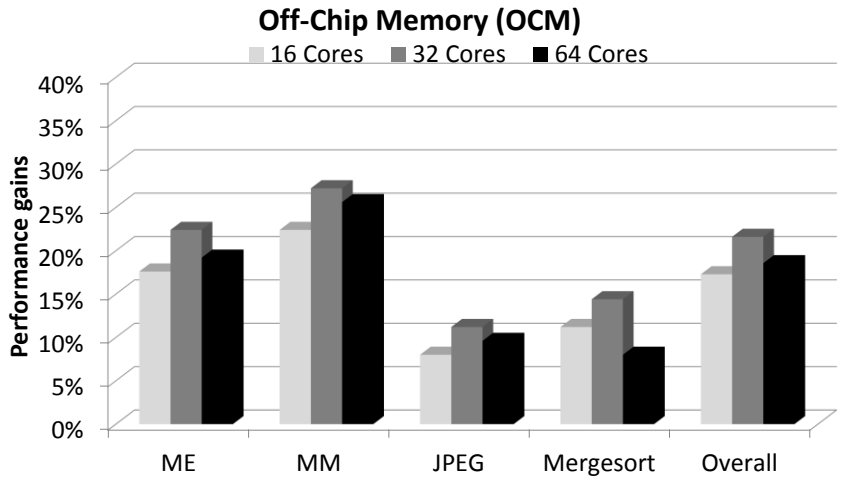
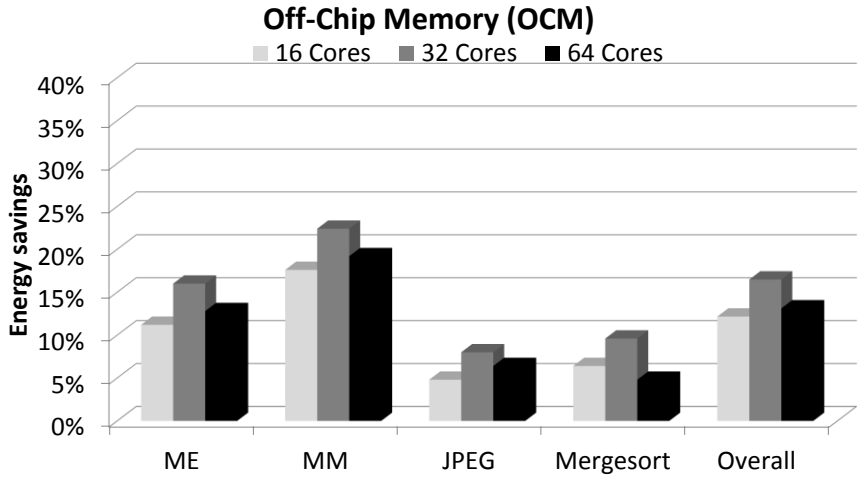


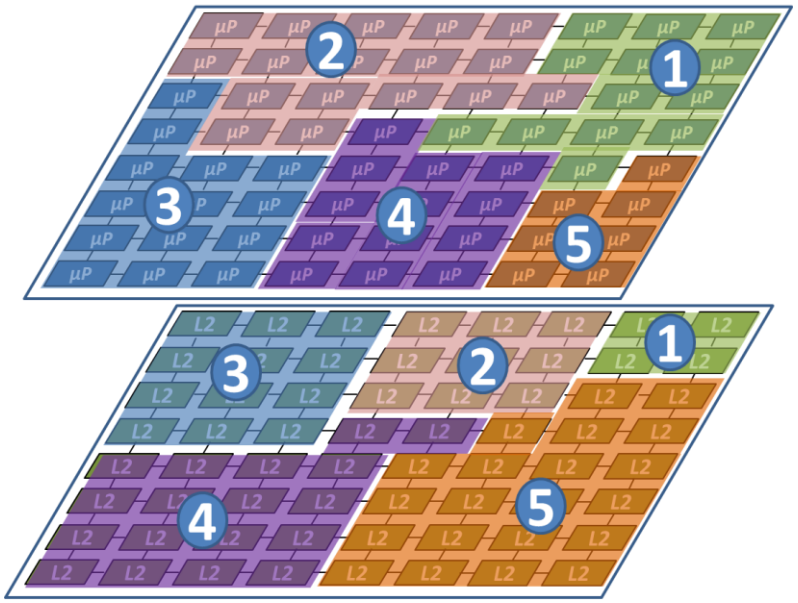
Figure 4.30. Energy results for OCM using both Processor and Memory Clustering.



4.6.4 Cluster Mirror (CM)

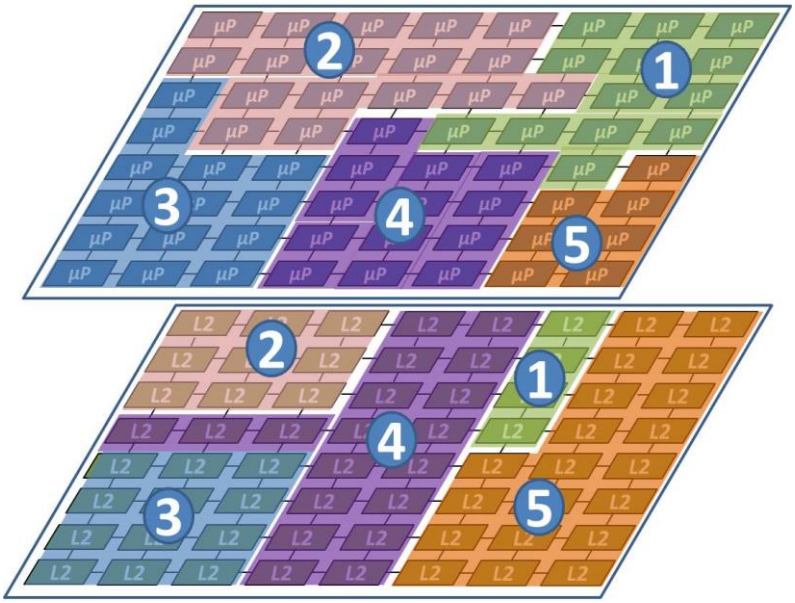
All policies presented so far are the same ones used in the processor clustering mechanism and apply key ideas of workload, communication latency and memory latency in order to improve performance when expanding clusters. However, these decisions are made in an isolated fashion. This means that, although the processor clustering uses the same key principles to decide which cores must belong to each cluster, the memory clustering may end up choosing a memory module located in a different node (even from a different cluster). This could potentially lead to a complete positional mismatch between processors and memories running the same application. Figure 4.31 illustrates how distant processors and memories can be if there are no directives keeping them together. In this example, the processor clustering and memory clustering executing application 3 barely share any nodes.

Figure 4.31. Distance between processor clusters and memory clusters.



Considering this scenario with processor clustering and memory clustering, a new policy could be used in order to create a pattern of cluster expansion (aggregation of resources) in which both memories and processors are closer together. That does not necessarily means that the same results apply in both clusters, i.e. not all nodes have processors and clusters executing the same application. Figure 4.32 presents an example different from the one shown in Figure 4.31. In this case, the memory clusters of the same application are closer together (but not strictly the same). The area of the processor cluster matches the respective memory cluster in a better way. Also, it is important to consider that, given the nature of the memory architecture, it is better to find cache lines in L2 modules in the same node as the cache that triggered the cache miss. This minimizes the latency since there is no need to send requests through the network. By keeping this clusters closer together, this latency can be minimized. From now on, we will refer to this policy as Cluster Mirror (CM).

Figure 4.32. New clustering using Cluster Mirror policy.

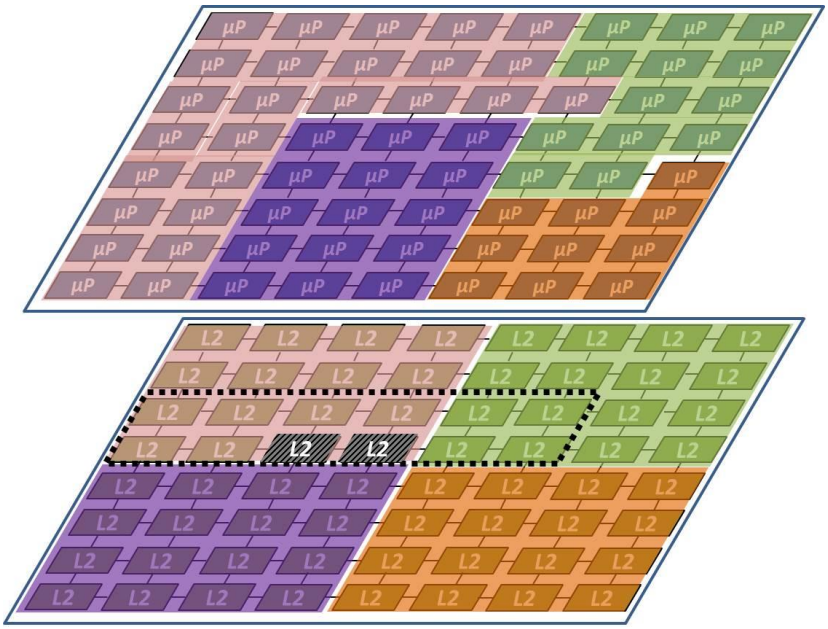


This policy can be very complex and can branch out other policies. For instance, we could consider that the processor clusters use one policy and the memory clusters use a different one. The difference is that these policies would only apply to a certain radius of a few nodes (maybe 2 or 3) in order to keep the resources together. At this point we are not considering all these possibilities and the exploration required to investigate these solutions is left as future work.

In the Cluster Mirror policy, we consider the baseline algorithm presented in Section 3.2 (Algorithm 1). By using this algorithm, the processor clustering would simply consider aggregating new resources depending on the number of tasks executing and proximity. On the other hand, the memory clustering would use the On-demand distribution with a checkpoint of 50% (since it was the one with the best results).

As the processor clustering occurs periodically and the memory redistribution happens only once during execution, we establish that the memory redistribution will be changed in order to give priority to memory modules placed in nodes where the processor is already executing the same application. Figure 4.33 presents an example of a decision following these rules. The memory controller (responsible for the redistribution) has the choice of aggregating four more memory nodes to the cluster. After considering the clusters that are supposed to give up resources, the closest candidates are the ones marked inside the dashed lines in the figure. Since there are three nodes in which the processor already belongs to that cluster, these are given priority. However, only two of them belong to a cluster that is supposed to give up memory resources (hashed blocks in the figure). The memory controller then chooses these two as part of the four and chooses another two in the group.

Figure 4.33. Cluster Mirror policy.



It is important to notice that this memory redistribution policy acts only once during the execution, but new policies with periodical redistributions could be used. Thus, the Cluster Mirror policy would be much more important in the decision making.

As presented in Figure 4.34 and Figure 4.35, the Cluster Mirror policy presents better results than Cluster Shape and Off-Chip Memory policies and also presents the best results for the smaller applications (JPEG and Mergesort). As this policy tries to keep processors and memories as close as possible, all applications can benefit from it. This is a policy that could be used in a system where there is not priority between applications or they are simply not known beforehand.

Figure 4.34. Performance results for CM using both Processor and Memory Clustering.

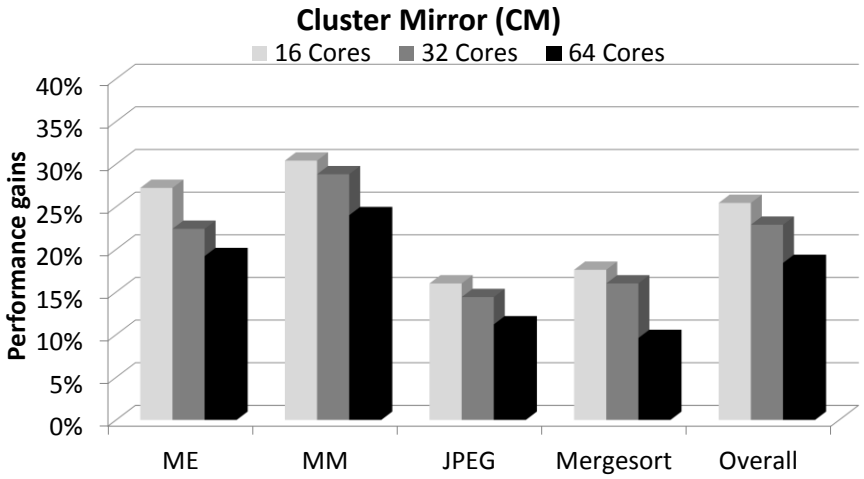
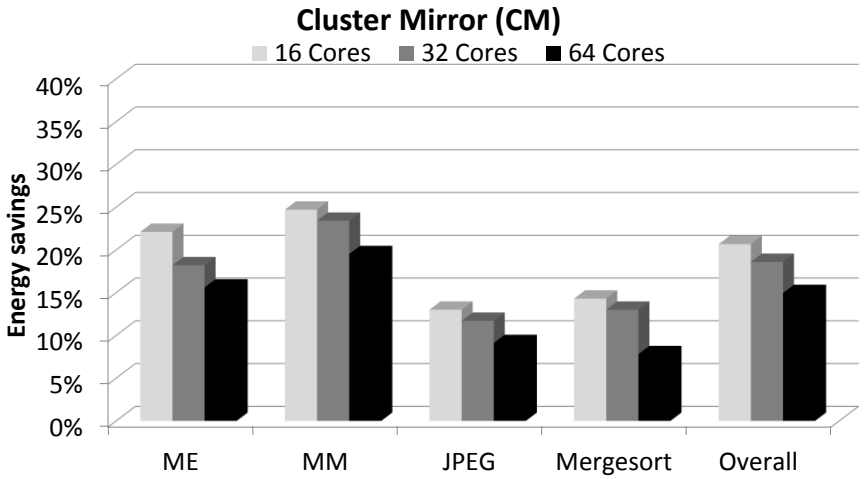


Figure 4.35. Energy results for CM using both Processor and Memory Clustering.



4.7 Final Remarks

In this chapter we introduce the concept of Memory Clustering. Since some applications have more memory needs than other, the key idea of this mechanism is to reserve memory resources for these applications taking them away from other applications that do not need them as much. The mechanism is composed of steps for finding how and when this redistribution is going to take place.

Overall the results presented show that, depending on the point where this redistribution occurs, there is room for performance and energy gains for the system as a whole. In the future, we intend to investigate other methods to determine the right moment for the redistribution. In the experiments presented here, the best checkpoint values (25% and 50%) are only known after the experiments. Therefore, in a situation where there is no previous knowledge on the applications, the best redistribution checkpoint can be different. In the future, we will investigate further the cache misses before and after the redistribution on each case. However, this investigation is not a simple task since some of these cache misses, after the redistribution, are caused by the L2 cold start and we must tell them apart from regular misses in order to come up with the exact overhead caused by this mechanism. As future works it is possible to investigate further the cache misses before and after the redistribution on each case. However, this investigation is not a simple task, since some of these cache misses, after the redistribution, are caused by the L2 cold start and we must keep them apart from regular misses in order to come up with the exact overhead caused by this mechanism. In addition, one can evaluate other mechanisms to find this ideal moment for the checkpoint without the necessity to go through experimentation.

This chapter also presented explorations regarding the combination of Processor Clustering and Memory Clustering. Some policies used in Chapter 3 were used as a common ground for the two clustering mechanisms. Overall the results show improvements over the isolated use of either one of the techniques. As mentioned, it is possible to combine other types of policies of each clustering mechanism in order to improve the gains even further, but we leave these investigations as future works.

5 RELIABILITY CLUSTERING

As presented in Chapter 2, some works in the literature have been starting to show resource awareness as one of the main concerns in the current MPSoC research. One of the reasons for a resource-driven design is the increase of failures in the chip as technology shrinks. As embedded systems become full of resources, it is natural to adopt redundancy-based approaches to deal with this reality. This chapter introduces the Reliability Clustering, which is a mechanism to reduce the impact of faulty memories in the system by means of cluster-based redundancy. In essence, this approach tries to leverage on the NoC infrastructure to support fault-tolerant memory schemes. As explained in Chapter 1, this work is a direct consequence of using cluster-based resource-aware approaches. Other aspects to be explored include programming support, intercommunication clustering and virtualization support. In this thesis, we chose to deal with another aspect that could take advantage of cluster-based mechanisms: Fault-tolerance and reliability.

This work is the result of a cooperation with the Research Group of Prof. Nikil Dutt, from the University of California at Irvine (DUTT, 2013). This group has been working with issues regarding fault-tolerance on memories (BANAIYANMOFRAD, 2011) on embedded systems, variability issues (BATHEN, 2012; TAJIK, 2013) and low-power design (BATHEN, 2011; HOMAYOUN, 2010), amongst other areas. The addition of resource-aware cluster-based approaches in an NoC-based environment led to the development of the studies presented here.

It is important to notice that this chapter is composed of two works (one as natural consequence of the other). In the first one we present an initial study on fault-tolerant memories supported by NoCs. The second one presents a similar fault-tolerant memory system also supported by NoCs, but which uses redundancy and a cluster-based approach.

As a first study we introduce a Last Level Cache (LLC) fault-tolerant method and present a set of modifications on NoC routers to support this mechanism (BANAIYANMOFRAD, 2012). This will the basis of the method presented. In this thesis, we present a novel solution at the on-chip network level for fault-tolerant design of LLC in MPSoC architectures. We leverage the network to implement a fault-tolerant scheme to protect the LLC cache banks against permanent faults. During a LLC access to a faulty area, the network detects and corrects the faults and returns the fault-free data to the corresponding core. We implement four different policies using this basic fault-tolerant method and perform cycle-accurate simulations using well-known NoC benchmarks. Results show that some of these policies take advantage of the intrinsic

parallelism of the communication mechanism in different ways, and, therefore, there is a tradeoff between the constant use of the NoC (increasing energy consumption) and lower latency.

Following, we introduce the Reliability Clustering (BANAIYANMOFRAD, 2013), which will use the fault-tolerance mechanism and explore the use of cluster-based aggregation patterns of redundant memories to improve the system.

Many research efforts have already investigated reliability and fault-tolerance of NoC platforms, mostly proposed in the areas of fault-tolerant interconnect, fault-aware routing, reliable communication, and fault-tolerant routers (PARK, 2006; PUENTE, 2004; PIRRETTI, 2004; BOGDAN, 2003; TSAI, 2011; KIM, 2006). However, research on coupling memory and NoC reliability is still in its infancy (MARCULESCU, 2009). Indeed, there are few works studying the reliability of on-chip memories at the network level (ANGIOLINI, 2007; WANG, 2010). Angiolini et al. (2007) modified the network interface of cores to protect critical data of on-chip memories in a NoC-based MPSoC. Wang et al. (2010) proposed a utility-driven address remapping technique to tackle the capacity loss in NUCA cache of NoC-based MPSoC architectures. However, to the best of our knowledge, there have been no previous efforts in the NoC reliability and fault-tolerance research area leveraging the NoC fabric to support fault-tolerance of on-chip memories or caches.

5.1 Basic Idea

Unlike traditional fault-tolerant cache schemes that perform all the bookkeeping in the cores, our approach migrates all of the fault-tolerant bookkeeping from the cores to the NoC fabric, by modifying the network routers to perform all the required tasks of a fault-tolerant scheme. As an example of a fault-tolerant scheme, we use a remapping-based technique to demonstrate the effectiveness of our proposed approach. Remapping-based schemes address permanent faults by remapping faulty blocks to other faulty/non-faulty blocks, thereby ensuring delivery of fault-free blocks. We propose four policies to configure the NoC, implement the fault-tolerant scheme, and based on each policy we add a fault map and a multiplexing layer to some of the NoC components. The fault map keeps information about the location of permanent faults and remapping data in the local cache bank of each router. The multiplexing layer performs the fault matching.

Using the NoC interconnection paradigm enables us to implement any fault-tolerant method in an efficient, modular, and scalable manner for several reasons. First, it supports modular and scalable implementation of fault-tolerant methods for emerging multi/many-core architectures. Different policies and configurations are possible with minimum overhead and design change. Second, using the NoC paradigm, it is easy to control access and management of redundancy resources. Also, redundancy for fault tolerance can be provided easily via the available interconnection network for any entity in the NoC. Third, adaptive and dynamic fault-tolerance of memory banks can be supported via NoC components. Since the routers have a key role in communication and management of data between all core and cache nodes in an NoC-based MPSoC architecture, by using their network information and modifying their routing algorithm, we can easily provide adaptive fault-tolerance of cache banks.

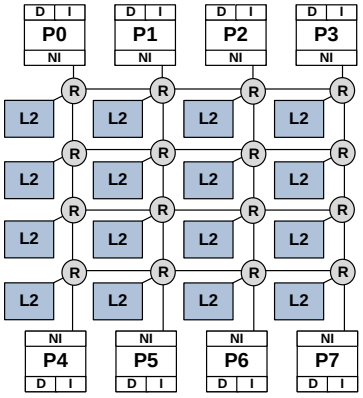
For this first study on the fault-tolerant method and NoC support, a simpler and smaller MPSoC was used. In future sections we use a more elaborated and larger MPSoC for the experiments.

5.1.1 MPSoC and NUCA LLC

We consider a non-tiled MPSoC architecture as our baseline, which is composed of 8 cores and 16 shared LLC banks. Cores and cache banks are interconnected as a 2-D mesh via a Network-on-Chip (NoC) architecture as shown in Figure 5.1. Each core is composed of a processing element and private L1 data and instruction caches. Based on our cache organization, each LLC bank is a portion of a larger distributed shared LLC cache.

The baseline design assumes a Non-Uniform Cache Architecture (NUCA) (KIM, 2002) LLC cache. With multiple banks within the LLC cache, we have the choice of either always putting a block into a designated bank (static mapping) or allowing a block to reside in one of multiple banks (dynamic mapping). We consider static mapping in our baseline design and model static NUCA policy for CMP architectures (CMP-SNUCA) (BECKMANN, 2004). Similarly to the original proposal (S-NUCA for uni-core processors in (KIM, 2002)), CMP-SNUCA statically partitions the address space across cache banks connected via a 2D mesh interconnection network.

Figure 5.1. Baseline Architecture.



Source: BanaiyanMofrad (2012, p. 64).

In static mapping, a fixed hash function uses the lower bits of a block address to select the target bank. LLC access latency is proportional to the distance from the issuing L1 cache to the LLC cache bank. By allowing non-uniform hit latencies, static mapping reduces hit latencies of traditional monolithic cache designs, which fix the latency to the longest path (AGGARWAL, 2007). Because a block can be placed into only one bank, the LLC access latency is essentially decided by the accessing block address.

5.1.2 NoC Architecture

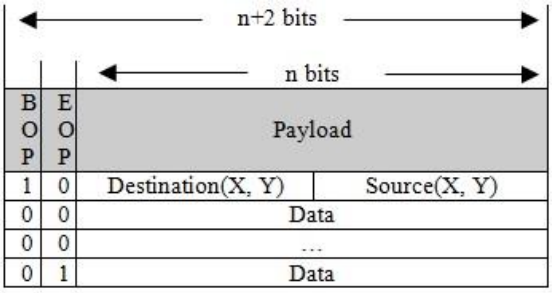
The network on chip architecture used in this work is the same used in the previous chapters. Here, we present details on the NoC routing and flow control mechanism.

SoCIN is a wormhole packet switched network, where each packet is divided into small units called flits (flow control unit). Each flit contains $n + 2$ bits where n is the payload and the 2 bits are flags for begin of packet (*bop*) and end of packet (*eop*). When

a flit arrives in the router it is stored in the input buffer. Once the arbiter reads the first flit (identified by the *bop* flag) it can determine through which output port the packet must go. This information is passed to the crossbar. If the crossbar determines that the output request is available, it begins to transmit the packet. The condition that defines the availability of the output is that the input buffer of the router connected to it needs to have enough space for at least one flit.

In addition, SoCIN implements a deterministic XY routing that avoids deadlocks. The routing is determined by the NoC address of the destination specified in the very first flit of the packet (the *n*/2 most significant bits are the destination NoC address and the *n*/2 less significant are the source NoC address). Once the first flit is routed through an output, the handshake protocol between the routers defines when there is enough space in the input buffer to send the next flit. This happens until the flit with the *eop* flag is routed. Therefore, all flits are routed in a pipeline fashion. Figure 5.2 shows the packet format in SoCIN.

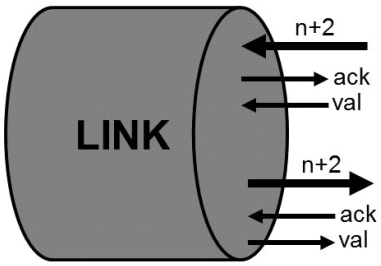
Figure 5.2. SoCIN packet format.



Source: Zeferino (2003, p. 171).

The flow control in SoCIN is handshake-based. When the sender writes the data on the link, it signals a valid (*val*) bit. When the receiver is ready to read the data, it activates the *ack* signal. Figure 5.3 depicts the Link structure with two unidirectional channels (to send and receive), each one of them with *n*+2 bits and *val* and *ack* signals.

Figure 5.3. SoCIN link signals.



Source: Zeferino (2003, p. 171).

5.2 Base Fault-tolerant Method

In general, all fault-tolerant schemes leverage some form of redundancy to tolerate the faulty components. As an example for fault-tolerant caches, we consider remapping-based techniques from the work in (BANAIYANMOFRAD, 2011) as our base fault-tolerant method. This method leverages a portion of faulty blocks as redundancy to mask faults in other blocks in cache memories. Here, we present a brief description of

this fault-tolerant method. This method uses a BIST module during boot to detect permanent faults (Hard Errors) in cache blocks and keeps the information of the location of faulty blocks in a fault map. Then, based on this information, it configures the address remapping for faulty blocks. This method divides each block of the cache into multiple sub-blocks that define the granularity at which failures are identified and remapped. A sub-block is labeled faulty if it has at least one faulty bit, as determined by BIST analysis. If two blocks have faults in the same sub-block, it means they have conflict and one cannot be used to mask faults in the other. During the configuration process, if a block is labeled as faulty, the system attempts to remap faulty portions of that block (Host block) to another block (called the Target block) that has already been marked as faulty and does not conflict with the host block. It then sacrifices and disables the target block to replicate all faulty sub-blocks of the host block. Thus, the fault-free block can be reconstructed from a combination of the two blocks; i.e., the host block together with the target block, via leveraging a multiplexing layer.

In many cases, a good choice of the target is another block in the same set (referred to as a local target block), so both host and target blocks can be read in a single access. Barring this case, we should always select target blocks from a different bank (a target bank), so that both blocks can be read without two serialized accesses to the same bank. We refer to this target block as a remote target block. Note that the target block can be selected from any bank in the LLC address space. Here, for each bank we limit the remapping policy to select target blocks from the adjacent banks that are within a one-hop distance in the NoC.

5.2.1 Proposed NoC Architecture

We extend the base fault-tolerant method to be leveraged for a shared multi-bank NUCA cache in a MPSoC architecture using the NoC backbone. Recall that the basic idea is that we distribute the fault map and multiplexing layer over routers of the NoC architecture. There exists one entry in a fault map for each set. We divide each entry in two sections: map of faulty sub-blocks and remapping data. The first section keeps the map of faulty sub-blocks in the set. The second section represents the remapping data of all blocks in the set. We keep the fault map of each bank in two separate blocks. We keep the first section of the fault map in a Fault Map Block (FMB). The second block is composed of the remapping data of the fault map, which is named Remapping Data Block (RDB). RDB includes one bit that indicates the faulty status of each block. We need the FMB data only to configure the address remapping and initialize the RDB. Therefore, there is no need to access this FMB data during the operation of the system, and we can keep it in main memory or in the hard drive. We only need to keep the RDB data at the cache level. Note that we assume a reliable 8T SRAM cell (CALHOUN, 2006) to protect RDB and also tag arrays.

To implement our approach, we perform modifications at three levels (cache access algorithm, the NoC routers, and cache banks) to support the fault-tolerant method, as described in the next sections.

5.2.2 Fault-tolerant Cache Access

We modify the LLC access algorithm to support the fault-tolerant method. In our fault-tolerant cache architecture, each LLC access first accesses the RDB of the fault map. Based on the fault status bit of the accessed block, if it is a faulty block it needs to have an extra access to a target block. The target block can be a local one within the same set or may be accessed from another bank (in case of a remote target block). Then

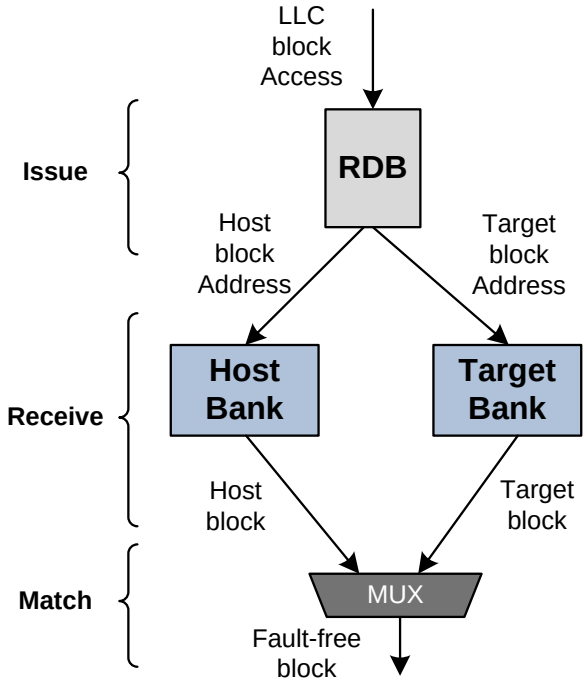
based on the location of the target block retrieved from the RDB, one or two levels of multiplexing are used to compose a fault free block, by choosing appropriate sub-blocks from both host and target blocks.

Based on our modified cache access, we define a fault-tolerant LLC access in three phases:

1. **Issue**: send access to both host and target blocks;
2. **Receive**: wait until receiving both host and target blocks; and
3. **Match**: match both host and target blocks via a multiplexing layer to tolerate faulty sub-blocks.

Figure 5.4 represents three stages of a fault-tolerant LLC access in our scheme.

Figure 5.4. Three phases of a fault-tolerant LLC access.



Source: BanaiyanMofrad (2012, p. 66).

Since the proposed fault-tolerant cache architecture increases the number of accesses to the LLC banks, it has a direct impact not only on network traffic but also on the LLC network access latency. In fact, the efficient implementation of a fault-tolerant cache architecture needs to consider its impact on both network traffic and latency, while trying to minimize its performance and cost overheads. Considering such design trade-offs, we propose three policies to implement a fault-tolerant cache architecture based on an NoC paradigm.

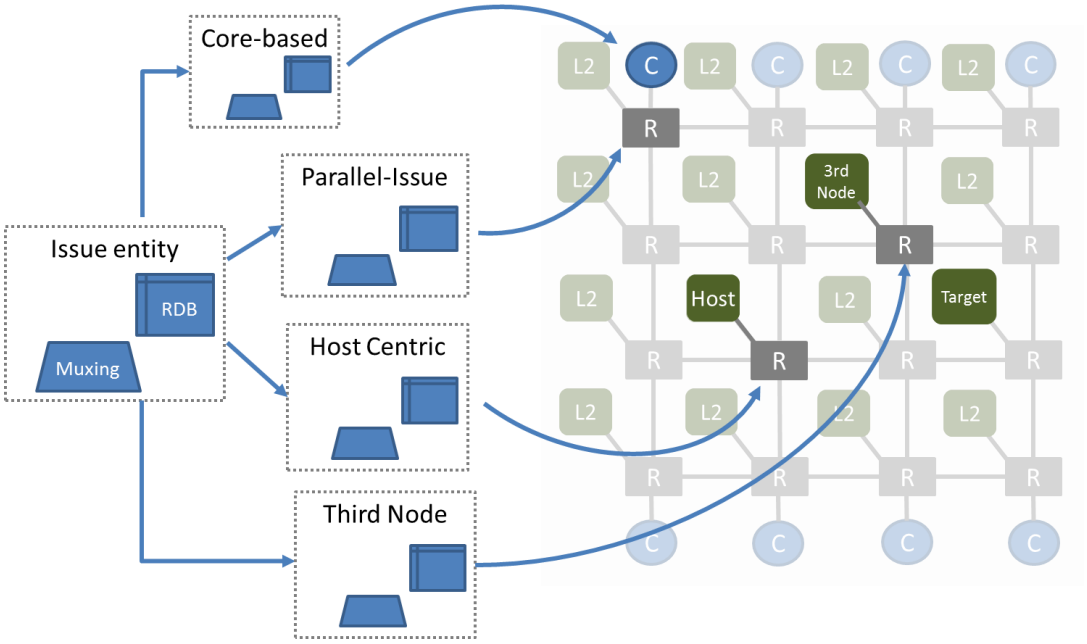
We define an entity as an NoC component (core, router, or cache bank) that can perform a phase of the fault-tolerant LLC access. Based on this definition, each one of the cores, routers, or cache banks can be an entity. Therefore, different phases of a fault-tolerant LLC access can be performed by different components (entities) in the NoC architecture. Based on the type and location of entities that perform different phases of a fault-tolerant LLC access, we can have different policies. However, in this architecture

we assume all phases of a fault-tolerant LLC access are done in one entity (Issue entity), and based on this assumption we define the four following policies:

1. **Core-Based**: This is the base policy, where all three phases of a fault-tolerant LLC access are done by the core that sends an access to the LLC bank. In this policy, we don't perform any changes to the NoC architecture. The NoC architecture is used to send accesses to both host and target banks and return the host and target blocks to the issuer core. One advantage of this policy is that it potentially minimizes the network latency by sending accesses to both host and target block in parallel.
2. **Parallel Issue**: In this policy, the local router of the core performs all three phases of Issue, Receive, and Match. In this policy, we don't change the cores. Only the local routers of the cores are changed. One advantage of this policy is that only a section of routers have to be modified. Also, it reduces the traffic between cores and NoC routers in comparison to the base policy.
3. **Host-Centric**: In this policy, the local router of the host bank performs all three phases of Issue, Receive, and Match and then sends the final fault-free block back to the requester core. One advantage of this policy is that it minimizes the network traffic in comparison with other policies. However, its network latency highly depends on the distance between the host and target banks.
4. **Third Node**: For this policy, an intermediary node is chosen to perform the three phases. The idea is to have a middle-way point that receives the request from the core and issues both requests to the host and target nodes in parallel. Eventually, host and target blocks will arrive at this same node. This would potentially decrease the amount of latency, since both host and target blocks are concurrently accessed and routed through the network in parallel; it does not serialize and lengthen the latency of the process as much as the Host-Centric policy.

Figure 5.5 represents different policies based on the Issue entity and shows the differences between them.

Figure 5.5. A general view of the NoC with different policies.

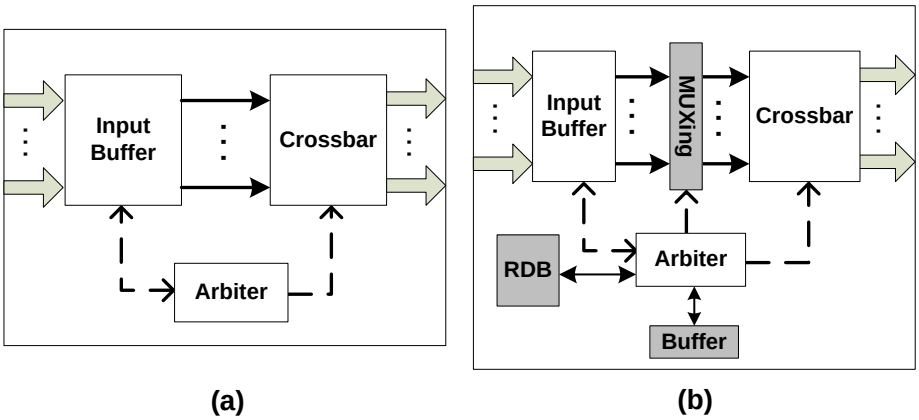


Source: BanaiyanMofrad (2014, p. 8).

5.2.3 Fault-Aware Router

As mentioned earlier in Section 5.1.2, the router has the arbiter that reads from the input buffer in order to determine the direction to which the packet must go (namely, north, south, east, west, or local). Then it informs the crossbar, from which input buffer it must read. This process is depicted in Figure 5.6a.

Figure 5.6. (a) Conventional Router; (b) Fault-aware Router.



Source: BanaiyanMofrad (2012, p. 66).

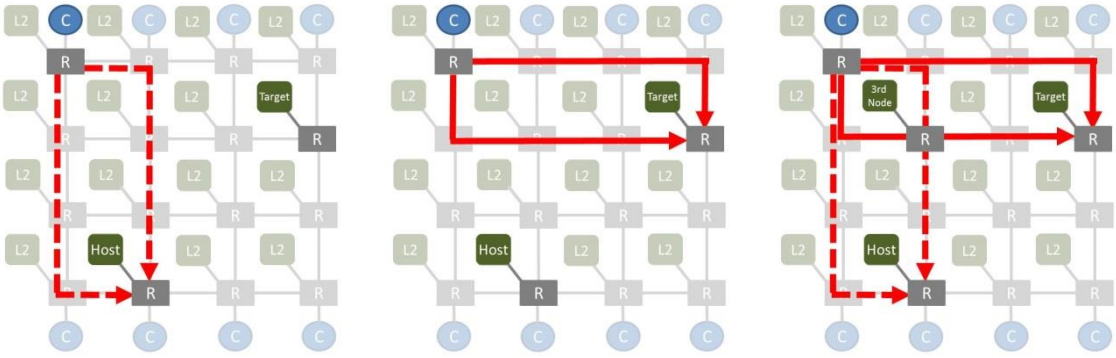
In order to support the fault-tolerant LLC access and the Parallel-Issue and Host-centric policies presented in the previous section, we need to change the router. Therefore, we propose a novel fault-aware router to support our fault-tolerant LLC access via the NoC architecture, which is detailed in Figure 5.6b. This new router has new components like an RDB, an extra buffer, and a multiplexing layer.

In a situation where a data request arrives, the arbiter verifies (using the RDB) if that particular address is a faulty access (access to a faulty block). In case of a faulty access, it checks if the target block is local (within the same set) or remote (another set in another bank). In case of a local target block, the cache bank handles the local remapping (more details in the next section). In case of a remote target block, the arbiter itself generates another request packet to be sent to the target bank. It keeps three pieces of information after that: 1) A flag that indicates that a matching operation is pending; 2) The direction from which the host and target blocks are expected; and 3) The address of the core that originally requested the data.

When a reply packet arrives, the arbiter checks whether this is an expected packet, as part of the original data (host block) request that is pending. If so, it stores the block in the extra buffer. When the target data packet arrives, the arbiter sends it and the contents of the extra buffer to the multiplexing layer for fault matching. The output of the multiplexing layer will be the final fault-free data packet (block) to be sent to the core that requested the data. It is important to notice that, for regular non-faulty data requests, the multiplexing layer is bypassed.

Note that, based on different policies, we replace none, some, or all of the NoC routers with the new fault-aware one to implement the fault-tolerant method. In case of the Core-Based policy, there is no need to use a fault-aware router and all of the processes mentioned above are performed inside the cores. For the Parallel-Issue policy, we replace all local routers of the cores (the routers directly connected to the cores) by the proposed fault-aware router. In case of Host-centric and Third node policies, since every router in the NoC can be an Issue entity, we replace all of the routers by the fault-aware router. Additionally, for the Third node policy, the first router (the one connected to the core) is responsible for choosing the third node. This happens by generating two lists of routers from the current router and the host router admitting the use of XY routing and YX routing (Figure 5.7a). In the same way, other two lists are generated considering the target router as final destination (Figure 5.7b). Finally, the farthest router that intersects at least two of these paths is chosen as the Third-node (Figure 5.7c).

Figure 5.7. Choosing the intermediary node in the Third Node Policy.



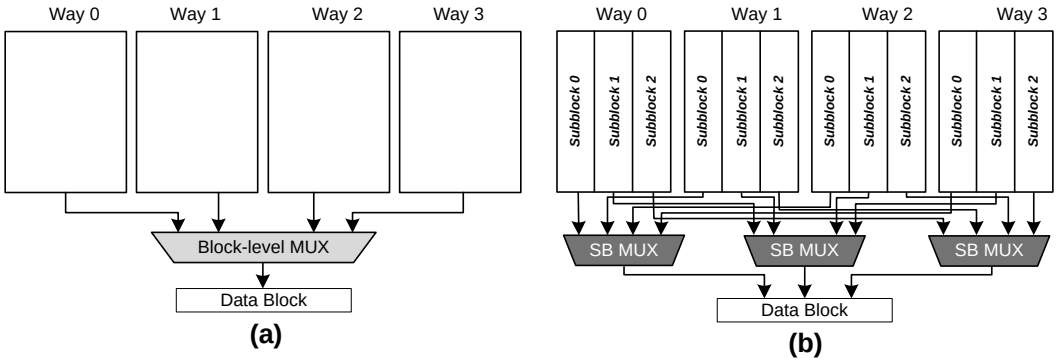
a) b) c)

Source: BanaiyanMofrad (2014, p. 9).

5.2.4 Fault-tolerant Cache Banks

We perform some changes in the LLC banks to support our fault-tolerant method. As mentioned earlier in Section 5.2, we divide each block into multiple, equally-sized sub-blocks. Since our LLC is a set-associative cache, each bank already has a large block-level multiplexer. We replace that multiplexer by a set of smaller ones with sub-block size scale. Figure 5.8 shows a conventional 4-way set associative LLC bank and the modified one with 3 sub-blocks in each way. The number of required multiplexers in each bank is equal to the number of sub-blocks per block.

Figure 5.8. (a) A conventional 4-way set associative cache bank; (b) Modified cache bank with 3 sub-blocks in each way.



Source: BanaiyanMofrad (2012, p. 67).

5.3 Design Space Exploration

We run the simulation process for our four proposed policies running different applications. Additionally, we consider a baseline case based on the original NoC with a fault tolerance similar to the one presented in Section 5.2. The restriction for this baseline is that there are no global targets. This means that the remapping for a faulty line will occur only by using blocks from the same bank. As a consequence, this baseline should have smaller opportunities for remapping, hence increasing the total number of disabled blocks in the system. The policies' results are all normalized according to the baseline. We evaluate the impact of the proposed NoC-based fault-tolerant approach by measuring performance, energy, and network latency and traffic.

We consider the network latency as the amount of cycles spent between issuing of the data request packet and receiving the data packet from the L2 cache bank. Network traffic is the amount of data that passes through the network taking into account the amount of time that the data spend travelling through the NoC. The idea here is to quantify the influence of the packets on creating possible contention situations. In order to have this notion, we present the traffic as the average buffer occupancy. This kind of result increases as more packets are sent and also increases as more packets flow through more routers in the network. Our performance metric is the number of Instructions per Cycle (IPC). An important observation is that, as we are using a cycle-accurate SystemC model of an NoC, all contentions are already taken into account by the simulator.

Our energy metrics include the LLC total energy and the energy-delay product. The LLC total energy reflects the total amount of static and dynamic energy from the L2 cache banks and NoC (routers and links) throughout the execution of all traces. The

energy-delay product is the product between the energy and the total execution delay for each application.

5.4 Experimental Results

We use a SoCIN interconnection as presented in Section 5.1.2. Each router is connected to a 1MB L2 cache bank. All L2 cache banks share the same address space of 16 MB LLC. Additionally, for each router, there is a module generating data and instruction requests based on memory traces obtained from a Simics (MAGNUSSON, 2002) simulation using 16 sparcV8 as cores. Table 5.1 summarizes the experimental setup of our architecture.

Table 5.1. Experimental Setup.

Processor Cores	8 SPARC V8
L1 Inst./Data Cache	2 Banks, 32KB each, 4-Way, 32B block, 2 cycle
L2 Cache (shared LLC)	16 Banks, 1MB each, 8-Way, 64B block, 10 cycle
Memory Latency	250 cycles
Interconnect	NoC of 2D mesh (4x4 for 16 banks) 32-byte links (2 flits per memory access), 1-cycle link latency, 2-cycle router, XY routing, 4 phit buffer size
Frequency	1.0 GHz
Integrated Technology	45 nm

Source: BanaiyanMofrad (2012, p. 68).

A workload of parallel programs with very distinct behaviors is created using benchmarks from SPLASH2 (WOO, 1995), PARSEC (BIENIA, 2008), and a parallel version of MiBench (GUTHAUS, 2001) suites. Using the Simics simulator, we extract all memory requests (data and instructions) by executing these applications in parallel. With these memory traces as input to our framework, we are able to extract performance and energy results. The performance results are the total number of cycles to execute all 8 parallel applications, and the energy results are obtained from Cacti 6.5 (WILTON, 1996) for the memory subsystem and from Orion 2.0 (KAHNG, 2009) for the network-on-chip.

For the sake of this study, faulty LLC banks are modeled randomly since SRAM cell faults occur as random events. These random faults are due to the major contribution of the random dopant fluctuation to the process variation (AGARWAL, 2005). A recent work in resilience roadmap reports the predicted probability of failure for inverters, latches, and SRAM cells as a function of technology node (NASSIF, 2010). Based on their results, the probability of failure for SRAM cells can be up to 2.6e-4. Here, since our case study is a block-redundancy fault-tolerant scheme and the analysis is at system level, we model failures at the block level. Based on that, the probability of block failure would be up to 7e-2. To enable a fair analysis during our evaluation, we consider 8 fault rates ranging from 1e-3 to 7e-2. In this work, we consider redundancy for any value ranging from 2% to 5%.

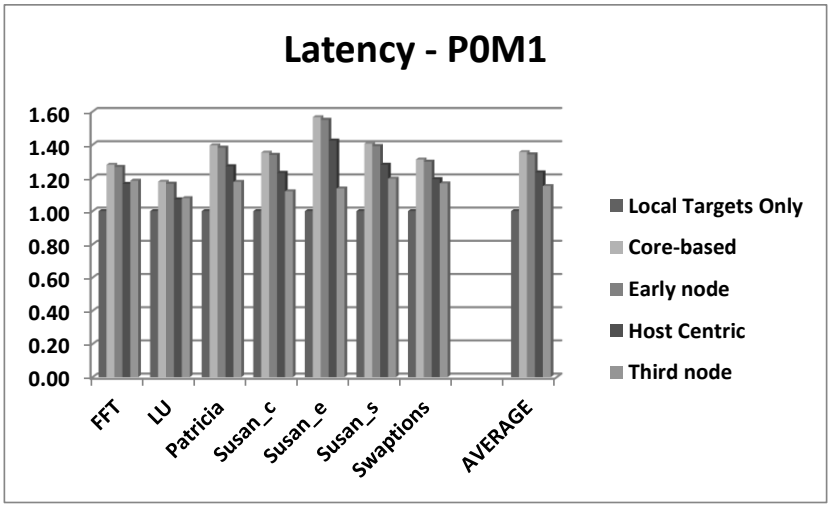
We evaluate the impact of system-level reliability clustering on performance, energy, and area overhead of the system using simulation runs of the experimental platform described earlier. For different system configurations, we change one design parameter, such as fault rate or amount of redundancy, and study its effect on different

design metrics of the memory subsystem. We consider the design space at two levels: intra-cluster and inter-cluster.

5.4.1 Network Latency and Traffic

Figure 5.9 shows network latency for the baseline and our three proposed policies. The Core-Based and Parallel-Issue policies present very similar results. This happens because of the difference between these two policies in terms of traversing the NoC. In the Core-Based policy, the packet must pass through one hop more than in the case of the Parallel-Issue policy. The Host-Centric policy, in contrast to our expectation, presents less latency than these two policies. Finally, the Third Node policy presents better results than the Host-Centric one in most benchmarks. However, for the FFT and LU applications, the Host-Centric policy seems to have a slight advantage. This may be explained by the fact that these are smaller applications and therefore there`s not much space for improvement by using the intuition behind the Third Node policy.

Figure 5.9. Network latency of the benchmarks normalized to that of Baseline.



Source: BanaiyanMofrad (2014, p. 15).

These results can be non-intuitive since the Host-Centric policy, essentially, requests the host and the target blocks sequentially, differently from the two other policies. However, it is important to notice that the RDB configuration plays a key role in these results, especially for the Host-Centric policy. In this study, by managing the allocation of host lines and target lines in such a way that they be placed in cache banks only at most 1-hop apart each other, we create a situation where the serialization of requests in the Host-Centric policy does neither affect the latency nor degrade the performance at all. Additionally as shown in Table 5.2, the RDB configuration generated for these experiments favors the Host-Centric policy, since the majority of the target banks are 1-hop apart from their respective host banks. This means that, because of the XY routing in the NoC, in most cases, cache access requests and response packets in the Core-Based and Parallel-Issue policies flow through the network using almost the same set of routers as the Host-Centric policy. This affects directly the major advantage of these policies, which is the parallel use of the network.

Table 5.2. RDB remapping results.

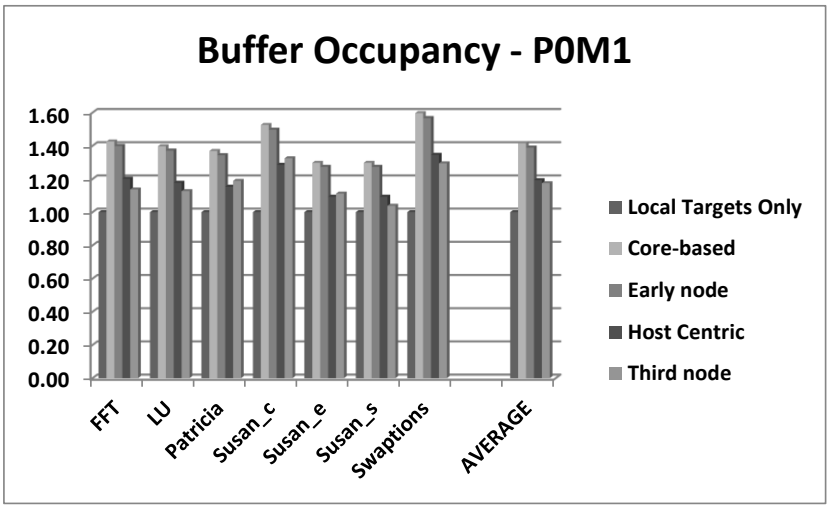
Global Targets	17702 (54.02%)

Lines remapped in 1-hop banks	17092 (96.55%)
Average number of hops to target banks	0.96

Source: BanaiyanMofrad (2014, p. 15).

In order to verify the impact of each policy on the NoC traffic, we analyze the average buffer occupancy in each case as presented in Figure 5.10. The results show not only the amount of data packets in each case but also how long they stay in the NoC. In the end, these results show the potential to generate contentions that each policy may have. Since Core-Based and Parallel-Issue policies only perform the multiplexing between the host and target lines in the last node and router of the return path, respectively, the network becomes busier and therefore they have higher average buffer occupancy. On the other hand, the Host-Centric policy matches the lines at the host node and, because of that, it causes less traffic and less possible contention in the network overall. Finally, for the Third-node policy, the results show that it presents results similar to the host-centric policy because it tries to reduce the amount of usage of the NoC by performing the multiplexing phase earlier than the Core-based and Parallel-Issue policies.

Figure 5.10. Average Buffer occupancy of different policies normalized to the Baseline.



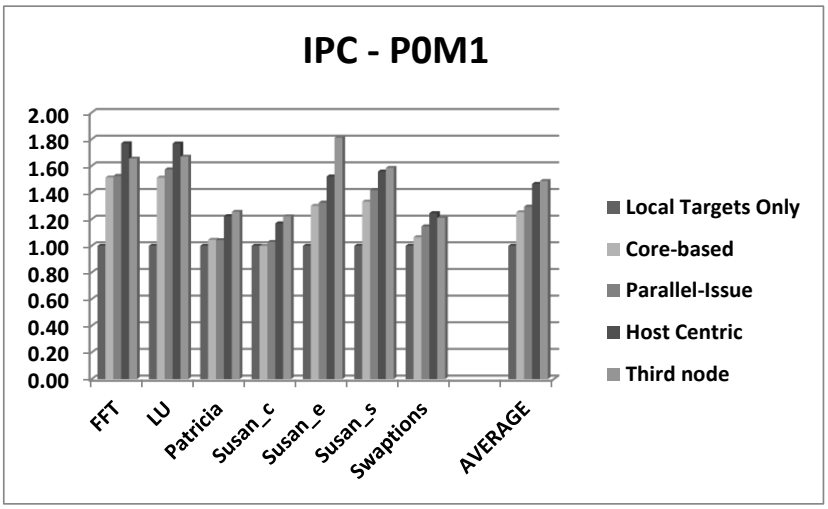
Source: BanaiyanMofrad (2014, p. 15).

5.4.2 Performance

Figure 5.11 represents the IPC results for different policies. It shows that, with small variations in different applications, the Third node policy has the best results on average. But, again, for smaller applications the Host Centric policy presents better results. Core-Based and Parallel-Issue policies have almost the same IPC. This outcome is expected because of the similar trends observed in network latency and average buffer occupancy results.

As shown in the network latency results, some applications benefit more from the fault tolerant method than others do. This happens because some memory accesses to faulty lines not necessarily generate L2 misses. Now some of this memory requests will generate L2 hits thanks to the remapping method. With that in mind, we can see that applications like FFT and LU have higher IPC, because they have lower network latency, as shown in Figure 5.9. Also, the amount of off-chip requests due to L2 misses harms the performance of the baseline more than in the proposed policies.

Figure 5.11. IPC of different policies normalized to the Baseline.

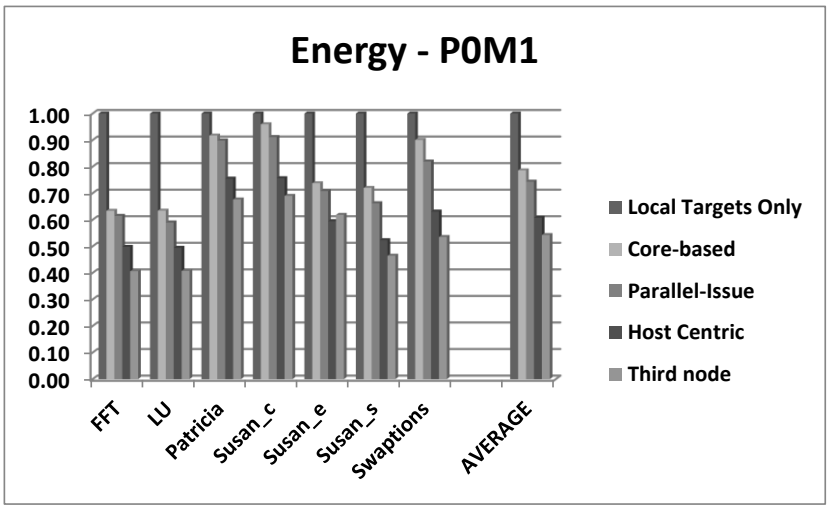


Source: BanaiyanMofrad (2014, p. 16).

5.4.3 Energy Results

Figure 5.12 shows the total LLC energy for different policies. As a direct consequence of less accesses to an off-chip memory, all policies present better energy results than the Baseline. In the case of Core-Based and Parallel-Issue policies, this advantage is smaller than the Host-Centric policy because of the network energy. The fact that the host and target blocks traverse through the network for more hops makes it more energy consuming in case of Core-Based and Parallel-Issue policies, especially for more activity in their router buffers. Also, due to reasons explained before, the network latency for these two policies is higher, which affects their energy consumption, as shown in Figure 5.12. For the same reasons, the energy consumption on the Third Node is much lower than in the Core-based and Parallel-Issue policies and therefore presents the best results in most cases.

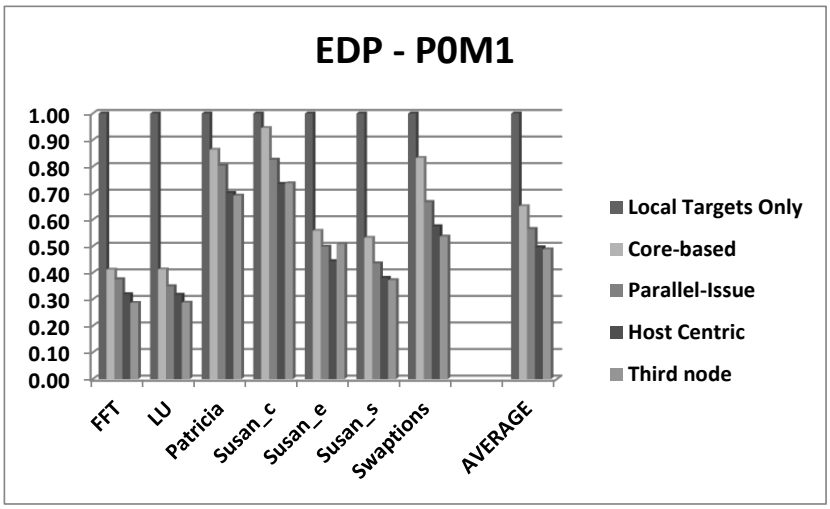
Figure 5.12. Energy of different policies normalized to the Baseline.



Source: BanaiyanMofrad (2014, p. 17).

Figure 5.13 shows the energy-delay product (EDP), which summarizes the results presented so far. The Third Node policy obtains the best EDP because of less amount of off-chip memory access, the lower amount of buffer occupancy, better performance, and lower energy in comparison with other policies. However, the Host Centric node also presents very similar results.

Figure 5.13. Energy-Delay Product of different polices normalized to the Baseline.



Source: BanaiyanMofrad (2014, p. 17).

5.4.4 Overhead Results

Since each policy requires some minor architectural changes, the solution presents some area and power overhead caused by the addition of extra components in the router, as explained in Section 5.2. Table 5.3 summarizes the area and power overheads of the proposed architecture for different policies. We consider the overhead of the RDB, the extra buffer, the multiplexing layer, and miscellaneous logic in the issue entities. The RDB is the major contributor to area and static power overhead. In addition, in order to protect the RDB and the tag arrays, we use a reliable 8T SRAM cell (CALHOUN, 2006), which has about 33% area overhead for these relatively small arrays in comparison with 6T SRAM data cells.

In the case of the Core-Based, Parallel-Issue and Third Node policies, each entity must have a full RDB (with entries for every line in the LLC). This is necessary since there is no limitation on which data request will arrive in the entity. Each core can access any line in any bank since they form the same address space. However, the Host-Centric policy is a different case. Since the issue entity is the router directly connected to the host bank, the data to be requested can only be one of the lines in that host bank. Therefore, the RDB in each router can have only the entries of the lines that the host bank holds. Hence, the overhead in all routers of the Host-Centric policy is equal to the size of one full RDB, while the overhead in the Core-Based and Parallel-Issue policies equals eight times (one per core) the size of a full RDB. Overall, the overhead of all policies in the proposed fault-tolerance solution is minimal (less than 3% area and less than 7% power overhead), demonstrating the viability of our approach.

Table 5.3. Power and Area overhead results.

Overhead	Core-Based	Parallel Issue	Host-Centric	Third Node
Area	2.46%	2.46%	0.41%	2.46%
Static power	5.53%	5.53%	0.81%	5.53%
Dynamic power	1.05%	1.05%	0.40%	1.05%

Source: BanaiyanMofrad (2014, p. 20).

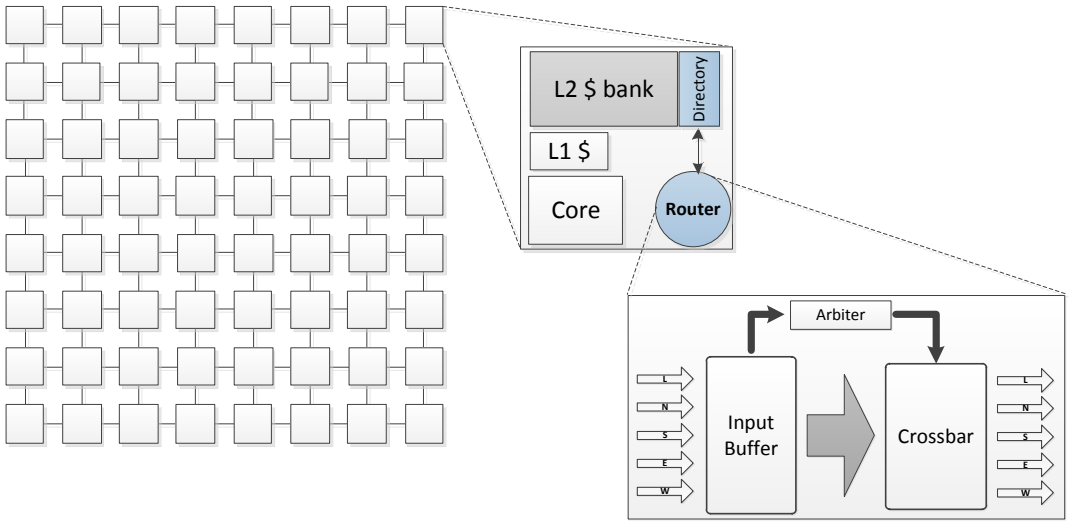
5.5 Reliability Clustering

Now we present the study on the Reliability Clustering itself. In this second study, we try to explore the process of clustering redundant memories in different patterns and experimenting distinct amounts of redundancy and fault rates. In order to do that, we continue to leverage on the NoC infrastructure in the same way presented in previous sections. Each router has a Remapping Data Block (RDB), but, this time, this mapping relates to redundant memories placed in the system in a cluster-based distribution.

5.5.1 Example NoC Architecture

To illustrate our approach, we experiment on an example tiled NoC-based MPSoC, which is slightly different from the one presented in Figure 5.1. Here, each tile comprises a processor core, private L1 data and instruction caches, a shared L2 cache bank, and network router/switch. Tiles are interconnected as a 2-D mesh via a network-on-chip infrastructure. Figure 5.14 shows our baseline 64-core MPSoC with an 8x8 mesh NoC. Based on our cache organization, the L2 bank is a portion of the larger distributed shared last-level cache (LLC). The baseline design assumes a Non-Uniform Cache Architecture (NUCA) (KIM, 2002) for LLC. A directory-based protocol similar to the one described in Section 4.1 is implemented in order to maintain cache coherence. This is a larger MPSoC than the one presented in Section 5.1.1, and it is also composed of heterogeneous cores. We now have the same amount of cores as L2 caches, thus tending to stress the memory subsystem much more. Consequently, the failures in the L2 caches will be more impactful.

Figure 5.14. Baseline Architecture.



Source: BanaiyanMofrad (2013, p. 1606).

To model fault-tolerance and organize redundancy we divide the whole NoC into clusters comprised of multiple groups of tiles. Each cluster is a subsection of the base NoC with the same topology but with a smaller group of tiles. The clustering is used to partition redundancy sharing among the tiles inside the cluster. In each cluster, some specific tiles (e.g., center tiles) contain the redundancy used for the fault-tolerance of all tiles inside the cluster; these are labeled as redundancy nodes. These redundancy nodes can be used flexibly to accommodate different fault-tolerance schemes and varying forms of redundancy, such as redundant rows/columns/blocks; exploiting sections of available clear/faulty blocks as redundancy; and ECC codes. Furthermore, we leverage the available interconnect backbone to support implementation of a variety of modular, scalable, and efficient fault-tolerance schemes. For instance, in our example tiled NoC architecture, we modify the direct router connected to the redundancy nodes to support our selected fault-tolerant scheme.

Because this clustering organization is independent of a particular topological structure, various physical topologies can serve as fixed-silicon but dynamically reprogrammable reliable multicore clusters on top of NoC platforms. Meanwhile, the inherent reconfigurability allows customizing clusters according to not only to the clustered and varying fault rates but also to application communication patterns and resilience needs. Therefore, clusters can be defined statically or dynamically during runtime. For the sake of simplicity, here we consider static clustering.

Our mesh-based NoC is composed of N nodes, C clusters, with each cluster containing a dxd mesh of tiles (d = cluster dimension size). The size and number of clusters can affect the yield, overhead, and network latency. The cluster dimension size (d) would determine the upper bound for the latency of fault-tolerant LLC accesses. Depending on the shape and size of each cluster, number of clusters, amount of redundancy, and distribution of redundancy among clusters, we can explore different design strategies, which meet various design constraints.

5.6 Design Space Exploration

To illustrate the flexibility and utility of our exploration methodology, we outline a design space exploration study using the example NoC platform and present results of two exploration studies with redundancy sharing at the intra-cluster level and inter-cluster level configurations.

For these experiments we use almost the same setup configuration (presented in Table 5.1) and applications, as described in Section 5.4. The only difference is in the number of cores and L2 caches, which now are 64 each, as presented in Section 5.5.1.

5.6.1 Cluster-level (Intra-cluster)

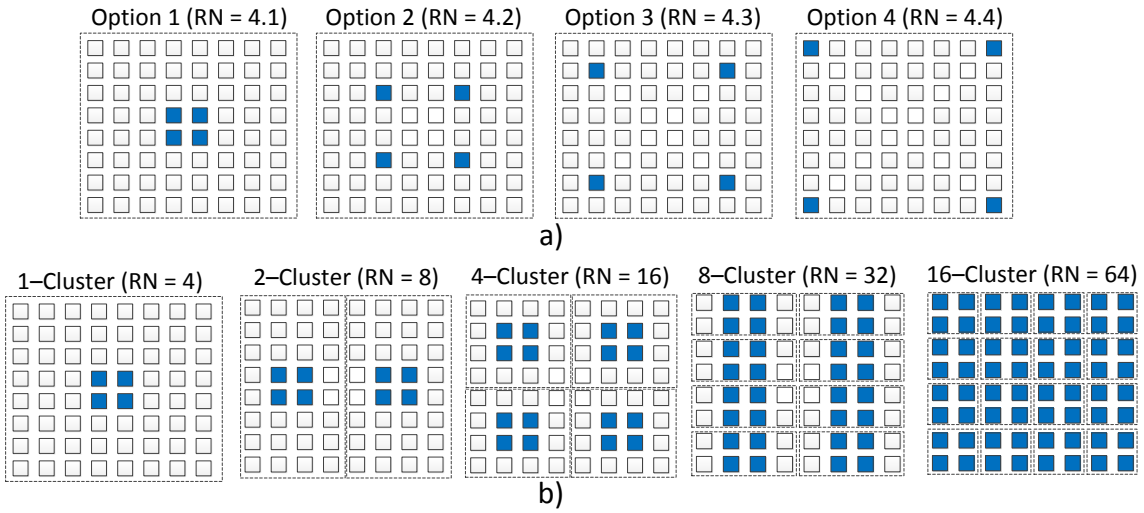
Here, we study the effect of redundancy distribution (number and location of redundancy nodes) on system design metrics. In this set of results, we consider the whole NoC as one cluster, fix either the amount of redundancy or fault rate, and change the redundancy distribution. In these experiments we explore the redundancy organization by changing either the number of nodes which contain redundancy (redundancy nodes) or their location. Redundancy node distribution can be studied in two directions, ranging from central nodes to all outward nodes or ranging from corner nodes to all inward nodes. Redundant elements are spread equally among all nodes that contain redundancy. Proposed distributions are selected based on a regular and scalable pattern, which is independent of the size of the cluster. Figure 5.15a presents some

possible distributions for our base architecture with four redundancy nodes. Here, for each redundancy distribution we have other variations of the distribution by spreading the nodes from the center (Option 1) towards the corners (Option 4). Notice that the label *N.X* means that the configuration has *N* nodes with redundancy, with *X* representing the distribution option. Higher values of *X* represent redundancy nodes that are closer to the corners.

5.6.2 System level (inter-cluster)

Here, we investigate the effect of node clustering and shared redundancy management among clusters on system design metrics. In this set of results, we change the number and size of clusters while fixing the total redundancy in the system. We select the size and number of clusters based on a regular and scalable pattern. Redundancy is spread equally among all clusters and all redundancy nodes inside each cluster. Here, we put the redundancy nodes in the center of each cluster. The intuition behind this approach is to guarantee that a certain number of cores always surround the redundancy nodes. This has the goal of not only minimizing the average distance between the cores and the redundancy nodes but also minimizing the variance in the average distance for each case. Figure 5.15b presents some possible clustering and distribution of redundancy for our base architecture with four central redundancy nodes per cluster. Here we illustrate sample distributions for 1, 2, 4, 8, and 16 clusters with 4, 8, 16, 32, and redundancy nodes, respectively.

Figure 5.15. Possible configuration patterns in a) cluster-level, b) system-level, with four redundancy nodes per cluster.



Source: BanaiyanMofrad (2013, p. 1607).

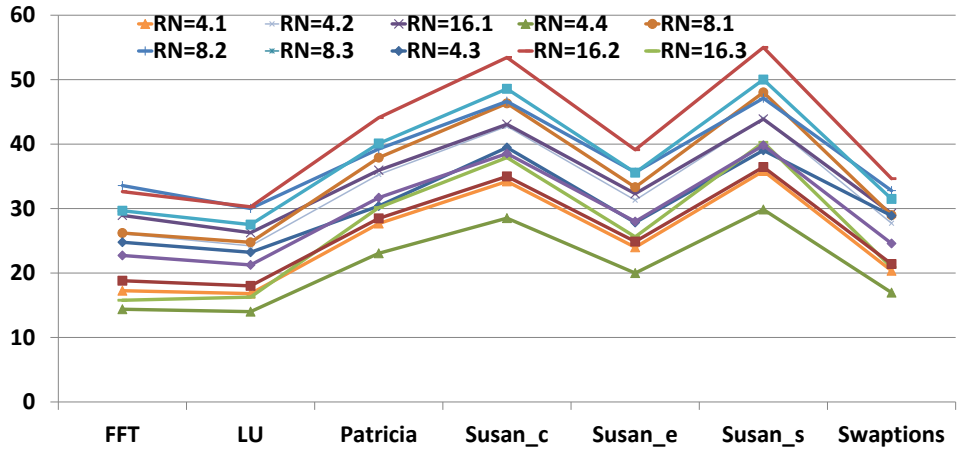
The results in this section represent the normalized performance/energy results of the block-redundancy scheme using the proposed clustering methodology, with respect to a baseline system without fault-tolerance support. This means that any access to a faulty memory address implies on off-chip memory access to retrieve the data.

5.6.3 Cluster-level Results

Figure 5.16 shows the gains of performance – the normalized execution time to the baseline – for each configuration shown in Figure 5.15a, with 3% of memory redundancy and also 3% of block fault rate in the system. Note that a 3% block fault

rate means that 3% of all memory blocks in the system are faulty. This figure shows the susceptibility of performance gains varying from one application to another. Some configurations present better results than others, depending on the application, but for most applications, in this particular case of 3% of block fault rate and memory redundancy, the configuration 16.2 seems to be the best option for this particular amount of redundancy and block fault rate.

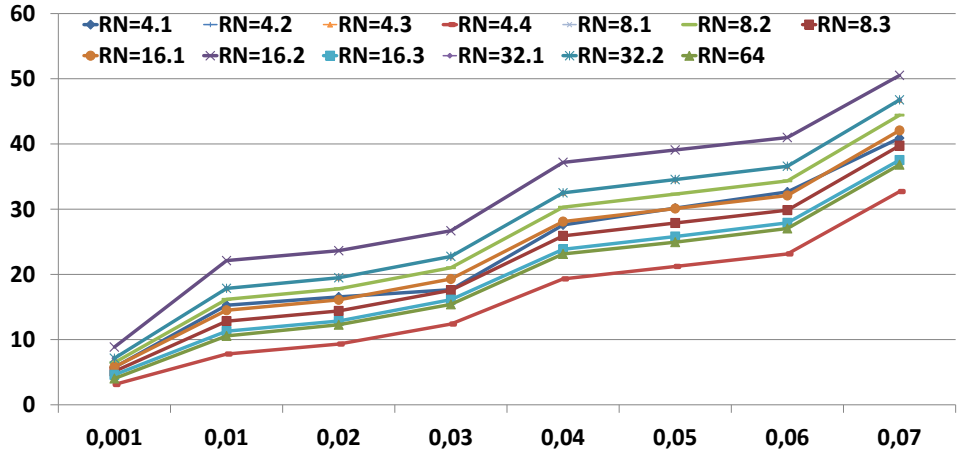
Figure 5.16. Performance improvement at cluster-level across different benchmarks.



Source: BanaiyanMofrad (2013, p. 1607).

Figure 5.17 presents the average performance results over all benchmarks when using 3% of memory redundancy and changing the block fault rate in the system. We note that configurations using too many redundancy nodes do not present good results. This may suggest that the best distribution must not have a small amount of redundant memory per node, because this will spread the redundant area too much, creating a higher average NoC distance between the nodes.

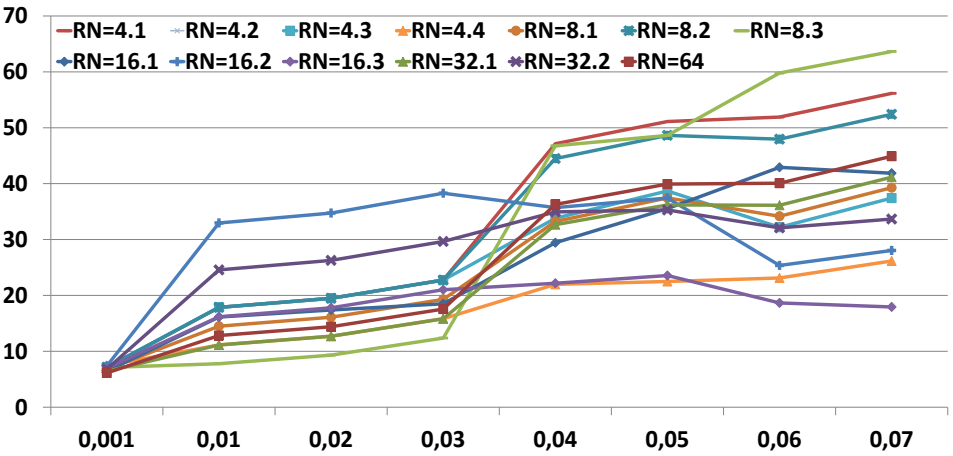
Figure 5.17. Performance improvement at cluster-level across different fault rates.



For a third experiment we evaluate the effect of different amounts of redundancy for cluster level (Figure 5.18). In this experiment, we fix the block fault rate to 3%. It is important to clarify that, even with an amount of redundancy higher than the block fault rate, the location of the faults is not equally distributed and some portions of the system can, in the worst case, contain all the faulty blocks. Hence, there is still the occasional

necessity to reach for redundancy memory on other nodes and, eventually, nodes that are far way. In Figure 5.18 we see an inversion from 4% of redundancy memory: configurations that were inferior in the previous experiments start to present better performance, and those that were previously superior now appear to become worse. This suggests that at some point, when the total of redundancy memory is higher than the amount of faulty blocks in the system, it is better to have the redundancy nodes in the corners. This could be due to the fact that, as the XY routing tends to concentrate the load in the center of the NoC (DEHYADGARI, 2005), these packets for redundancy memory requests (which now occur less frequently) may generate less contention if they avoid the middle of the NoC.

Figure 5.18. Performance improvement at cluster-level across different amounts of redundancy.

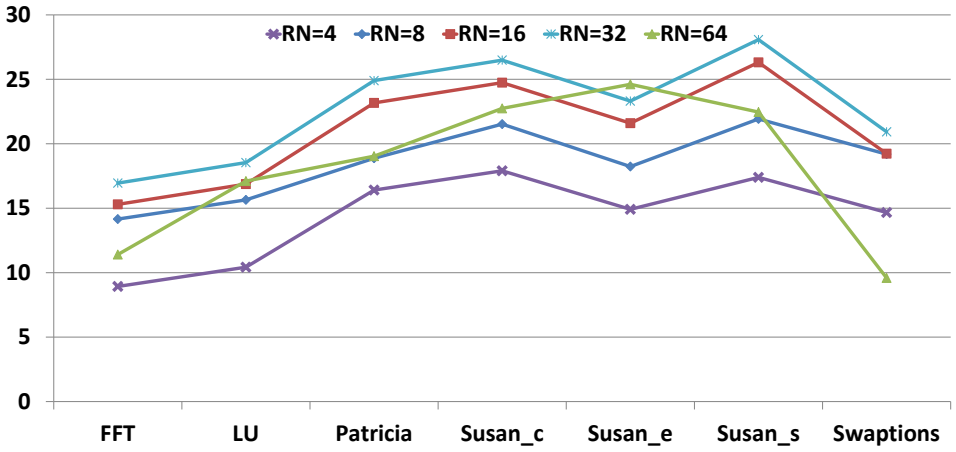


Source: BanaiyanMofrad (2013, p. 1607).

5.6.4 System-level Results

For the inter-cluster case we put the redundancy nodes in the center of the cluster and change the size and number of clusters. Similarly to Figure 5.16, Figure 5.19 presents the results at the inter-cluster level in terms of performance gains when fixing the redundancy memory at 3% and the block fault rate also at 3%. In this particular case there is no much variation from one application to another, but it is possible to see a lot of variation for the 64 configuration. This is a configuration that spreads the redundancy data equally throughout all the nodes, and, therefore, it is more susceptible to different memory access rates. For less memory accesses it works well, because there are few redundancy memories per node across all nodes. For higher memory accesses it may not work well since it has a higher chance to have a situation where the redundancy node may be too far away.

Figure 5.19. Performance improvement at system-level across different benchmarks.



Source: BanaiyanMofrad (2013, p. 1608).

Figure 5.20 presents performance results for these configurations, with a total of 3% of redundancy memory, while changing the block fault rate, ranging from 0.1% to 7%. The figure shows that the 8-cluster configuration has the better results. This is due to the fact that this configuration presents a low average distance between cores and redundancy area and also because this average distance does not vary too much considering all cores.

Figure 5.20. Performance improvement at system-level across different fault rates.

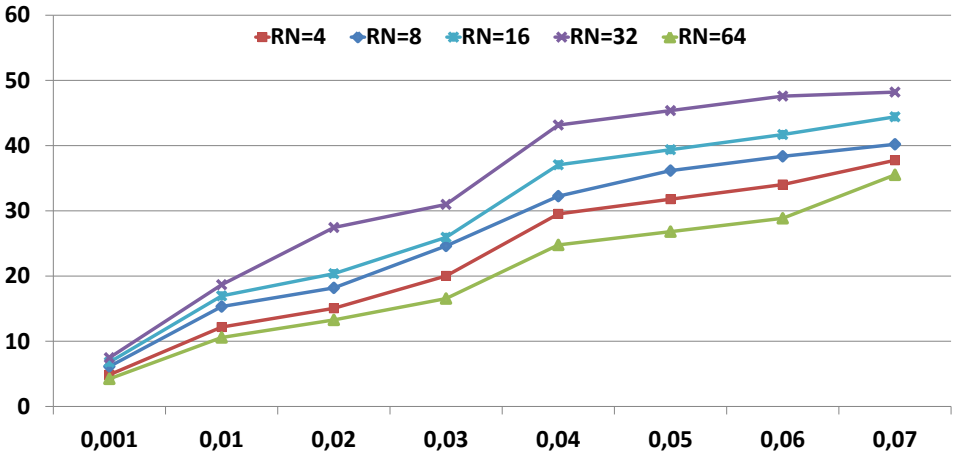
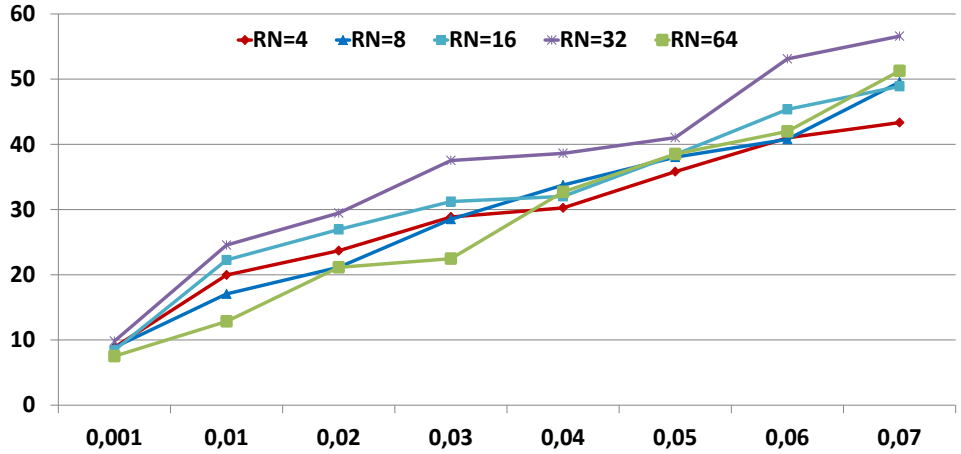


Figure 5.21 presents results regarding the same exploration presented in Figure 5.18 but this time with this set of system level configurations. Similarly, we observe that a few changes occur after the point of 3% of redundancy memory, but, differently from the case of cluster level configurations, there is no clear change of scenario, i.e. the better result remain unchanged. This could be due to the fact that in this set of configurations there is no configuration that places the redundancy nodes in the edges of the NoC.

Figure 5.21. Performance improvement at system-level across different amounts of redundancy.



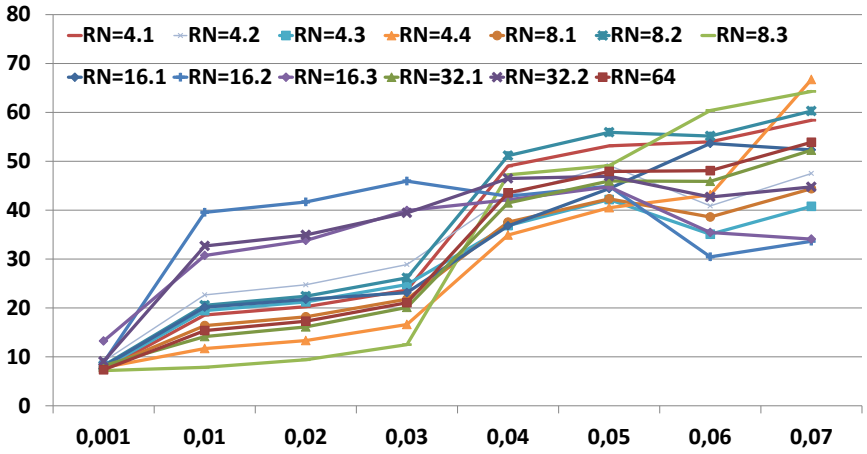
Overall, by looking at the results on both sets of experiments, it is possible to see that a good decision on where to place the redundancy nodes in the NoC can lead to a performance improvement of 20% in some of the cases. This is a relevant difference, since it does not incur in an increase in overhead and only requires changing the redundancy organization.

5.6.5 Energy Results

Figure 5.22 presents the energy results (normalized to baseline) for both sets of experiments fixing a fault-rate of 3% and varying the redundancy from 0.1% to 7%. The trends here are quite similar to the ones presented in Figure 5.18 and Figure 5.21. The difference is that, since off-chip memory accesses consume more energy than regular on-chip memory accesses, the penalty for not having a redundancy memory hits the baseline the hardest. The results presented here show the energy savings of the redundancy cluster approaches compared to this baseline. The energy saved reaches 47% in the cluster-level approach and 51% in the system-level approach.

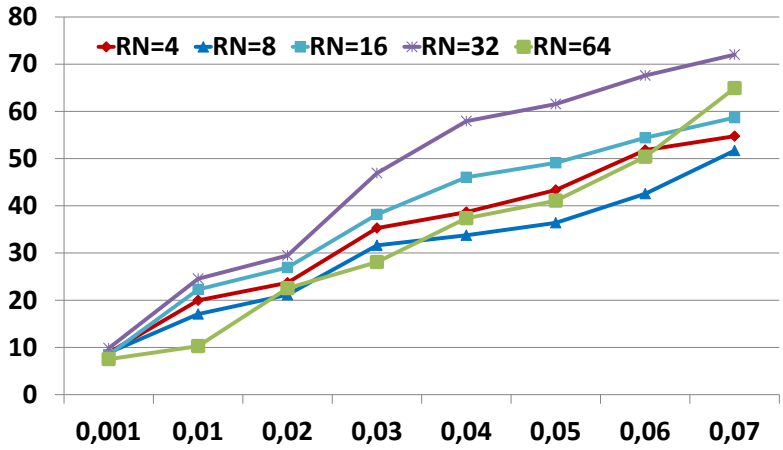
In Figure 5.22 we present the results for the intra-cluster level approach. The energy consumption in this experiment for the most part follows the same trend as the performance results. Some situations can present better results depending on the average distance between the cores and the redundancy nodes (like RN=16.2). If better distributed, the energy consumption to reach the redundancy nodes will also be smaller. After the point where the amount of redundancy surpasses the block fault rate, some configurations that place the redundancy blocks far away from the center of the NoC present better results (like RN=8.3). This could be because of the tendency of XY routing to create more contention in the center of the NoC.

Figure 5.22. Energy results across different amount of redundancy for the cluster-level configurations.



In Figure 5.23 the inter-cluster (system-level) approach energy savings are presented, and the trend is similar to the one presented in Figure 5.21. The differences between the configurations with different numbers of clusters seem to be increased and, therefore, the 8-cluster configuration presents the best results due to an even distribution of redundant blocks in the NoC without reducing the amount of redundancy per node too much. The 16-cluster results present a better result at 7% redundancy rate. This is due to the fact that, in this system-level approach, if the redundancy rate is much higher than the block fault rate, the amount of redundancy per node reaches a point where fewer and closer redundancy nodes are enough to cover the faults. With a lower distance between the nodes, a lower NoC energy is consumed.

Figure 5.23. Energy results across different amount of redundancy for the system-level configurations.



6 CONCLUSIONS AND FUTURE WORKS

This thesis investigates solutions for improving the overall performance on NoC-based MPSoC systems. These solutions are all based on the same resource-aware paradigm, which has been the focus of several works in the literature. The efficient use of resources in future MPSoCs is crucial in order to meet ever-increasing applications' requirements. These requirements come in the form of not only computational resources but also memory subsystem adaptability and organizational fitness. With the advance of embedded software platforms, one system can be executing applications from distinct developers, which, in turn, may have used different parallel programming models.

The first proposal is the Processor Clustering approach, in which tasks from the same application and their current resources are treated as a cluster. As processors become more available (when tasks finish their execution), a distributed scheduler chooses other more charged processors to migrate tasks from. In addition, to support distinct programming models, the hardware platform presented in this work is adaptable in the sense of providing different memory organizations (distributed or shared). Hence, we are able to provide native support for Message Passing and Shared Variables programming models. In the context of this work, four resource-aware task migration policies were proposed. These policies try to take advantage of intrinsic characteristics from the software or hardware platform in order to minimize the task migration overhead, thus improving the overall system performance. Results show that different aspects can be considered regarding the task migration in order to effectively distribute resources. For instance, aspects like task migration overhead cost for different parallel programming models, the workload of the applications, and the average distance between nodes running tasks from the same application have impact on performance. Experimental results using policies that consider these aspects show that resource management of tasks in an MPSoC can be improved up to 22.5% in average. Furthermore, these policies result in different gains (or losses), depending on the combination of parallel programming models used by distinct, concurrent applications.

The use of clustering techniques can improve performance by using available processing resources and spreading the load amongst cores in the system. However, some applications may have memory requirements that are more important to meet than processing requirements. Therefore, in this proposal a Memory Clustering approach is presented. In this approach the memory subsystem is treated as a set of available resources (in the form of distributed shared L2 cache banks) that can be logically reorganized to become exclusive resources of an application in the system. In order to find the best aggregation of resources two policies are presented. In one of these

policies the resources are rearranged at runtime after a statistical comparison based on cache misses. Results show different behaviors from the applications, in which some of these applications loose performance while others can gain. Performance and energy results of this policy are highly dependent on the moment where the decision to rearrange the memory resources is made. Overall the results presented a possibility of up to 18% of performance gains and 20% of energy savings.

Another study presented in this thesis aggregates some of these clustering techniques in one experiment. For instance, we can use Processor Clustering and Memory Clustering at the same time. In this case, as applications need more processing resources, they also need memory resources, therefore the task scheduler can continue to act as resource manager for the cores in the system while the external memory controllers can redistribute L2 cache banks accordingly. This creates the possibility of slightly different migration policies based on processor clustering policies. Also, additional memory distribution policies are based on the fact that an application cluster can be spread in the system and distant from its respective memory resources. In this case, the resource manager must be careful when assigning L2 cache banks to certain clusters. Results presented here show that this aggregation of clustering techniques produces an improvement of up to 31% of performance gains and 28% of energy savings.

Managing processors and memories as resources in a cluster-based environment can lead to investigations trying to improve other features like fault tolerance or reliability. With that goal in mind, this thesis also presented investigations regarding reliability solutions that use the notions of clusters. In particular, a cluster-based approach is presented in order to provide reliability at the memory level. To construct reliable memory architectures in emerging multi/many-core NoC platforms, designers must consider the interconnect, distribute and utilize redundancy efficiently to alleviate the high cost of fault-tolerance schemes, employ a hierarchical set of fault-tolerance strategies, and create novel design paradigms that consider system-level issues. The final cluster-based approach tries to tackle this issue by introducing Reliability Clustering. In this case, the concept of clustering is used to model allocation and placement of redundant memories in an NoC-based MPSoC. In this study, we leverage on the NoC to implement a fault-tolerant scheme for L2 caches using small redundant memories spread in the system. The goal here is to efficiently find redundant memories in order to avoid cache misses. Several cluster-based patterns of distribution of these redundant memories are explored, showing a set of positive results with no obvious winner. Thus, the main contributions of this work are:

- Proposal of a hardware support for execution of applications with distinct parallel programming models;
- Resource management of processing cores using a cluster-based task migration approach;
- Exploration of different resource management policies based on intrinsic characteristics of both hardware and software of the platform;
- Proposal of a resource management for the memory subsystem based on applications' requirements;
- Fault-tolerant scheme for memories using NoC support;
- Design space exploration on cluster-based placement of redundant memories.

6.1 Future Works

This thesis presented resource-aware cluster-based techniques in order to improve performance on NoC-based MPSoCs. Investigations presented here were focused on three main areas in a chip: processors and applications, memory subsystem, and communication infra-structure. These approaches could be used isolated or slightly combined (as presented in Section 4.6). There are some extensions of this work that could be investigated.

6.1.1 New policies for Processor and Memory Clustering

As mentioned in Chapters 3 and 4, most experiments with policies were performed in an isolated fashion in order to investigate the impact of that key idea in the overall migration mechanism. However, there is no evidence that these policies cannot be combined in some way to produce a more robust decision making algorithm. In addition, other characteristics of the application can be taken into account, such as:

- Size of an application (or task): the size of a task can be helpful in order to decide whether or not to migrate it. Some policies could take into account this information depending on the programming model (and memory hierarchy used).
- Complexity of the application: many times in this thesis we have dealt with objective features of an application, like the number of tasks. However, tthis is not an accurate metric to determine the complexity of an application. Obviously, to know whether or not some application is computationally more complex than other is a valid information.
- Lifetime of an application (or task): it would be interesting to have knowledge on how long the application or task is going to run. This information is very desirable, since it seems a waste of time to migrate and give more resources to an application that is about to finish.

6.1.2 Communication Clusters

The same concept of cluster-based resource-awareness used for the processors and memory subsystem can be applied to the interconnection mechanism. In this particular case, the Network-on-Chip, can be seen as a set of resources (the routers), and a particular application running on processors connected to those routers may have a performance improvement if all routers were isolated from the other routers in the chip. This would bring lower communication latency, since no traffic from other clusters would be allowed inside the cluster.

This approach would be very dependent on the processor clustering mechanism. This would happen because the communication resources (routers) would have to be the same ones connected to the processors. In the case of memory clustering, the situation is more complicated, since the mechanism allows the usage of memories that reside in nodes connected to other clusters. This could be solved by using exceptions regarding the traffic allowed inside the cluster.

6.1.3 Holistic Approaches

As debated, the clustering mechanisms can be used in distinct layers in the system. However, the use of these resources could be improved if one considers a holistic approach. The key idea would be to exchange relevant information between layers in

order to help the decision of redistributing resources. Below we list three possible situations where this information exchange could be useful:

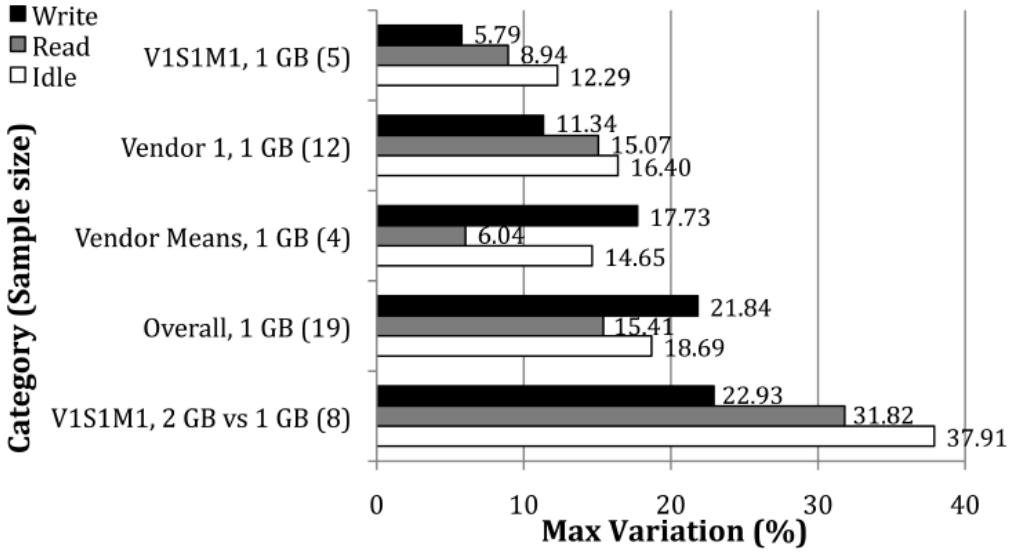
- Application layer: Fault-aware mechanisms could be used in order to provide information regarding the status of resources. For instance, a set of routers could be defective. This information would be important when allocating critical applications.
- Memory layer: Generally speaking, all information regarding memory access patterns could be useful if extracted beforehand. Also, critical tasks and number of tasks per node would have impact during the decision of redistribution.
- Communication layer: As mentioned in the previous section, the information of application allocation (processor clustering) and memory usage (memory clustering) could be useful regarding router allocation. This information would be even more important when considering a heterogeneous environment (caused by router failure, for instance).

6.1.4 Variability

Another goal would be to explore the effects of variability in an NoC-based MPSoC which would make it a heterogeneous environment.

Variability is already present on current system-on-chip, and the transistor scaling process tends to aggravate this scenario. Figure 6.1 presents the variation of power of DDR3 DIMM memories fabricated using current technology. Power variations occur even when the device is on standby (leakage power) and reach 12% for the same model of a single vendor and 16% for different models of the same vendor. In addition, the dynamic power is also affected, since there is a variation up to 21% amongst tested memories (DUTT, 2013).

Figure 6.1. Variability impact on DDR memories depending on models and vendors.



Source: Dutt (2013, p. 127).

The variability phenomenon can manifest itself as a function of four causes. These causes impact performance, power consumption, and hardware reliability. They differ in the sense that they can be classified at different points inside the design flow in which they manifest themselves.

- Manufacturing: The ITRS (International Technology Roadmap for Semiconductors) highlights power/performance variability and reliability management in the next decade as a red brick (i.e., a problem with no known solutions) for design of computing hardware. As for memory/storage devices, recent results show variation between 27% and 57% on energy consumption and up to 50% on fault rate amongst flash memory devices that are supposed to be identical (same manufacturing process).
- Environment: The ITRS projects Vdd variation to be 10% while the operating temperature can vary from -30oC to 175oC (e.g., in the automotive context) resulting in over an order of magnitude sleep power variation and several tens of percent performance change.
- Aging/Wear-out: Wires and transistors in integrated circuits suffer substantial wear-out leading to power and performance changes over time of usage. Physical mechanisms leading to circuit aging include bias temperature instability, hot carrier injection, and electromigration.
- Vendor: Parts with almost identical specifications can have substantially different power, performance or reliability characteristics. This variability is a concern as single vendor sourcing is difficult for large-volume systems.

As the variability extends to almost every chip currently manufactured, the devices affected by it become different regarding physical features (power and energy consumption) and this may also affect response times like memory latency. Therefore, chips with a large number of homogeneous cores and memory modules, in reality, will behave as heterogeneous systems.

However, we can take advantage of this forced heterogeneity by identifying different components and using them appropriately. In the context of resource-aware approaches, in each layer, resources can be assigned to clusters depending on their performance and energy cost. High-priority applications would be assigned to better resources.

6.1.5 Dark Silicon

Another work that could derive from this thesis is to explore a tendency that researchers on the multicore field have called Dark Silicon (ESMAEILZADEH, 2011). Taking advantage of transistor scaling tendencies, processor manufacturers have been releasing in the market processors with an ever higher amount of cores. This tendency has led to expectations of a massive number of cores per processor in a near future. Some expect the manufacturing of hundreds or even thousands of cores per processor. However, the transistor scaling process has not been followed by voltage and capacitance scaling (known as Dennard scaling). This means that even though the size of transistor is scaling (making the chip area a manageable problem) the voltage needs are not diminishing in the same rate. Soon, the power and energy consumption levels

will force systems-on-chip to keep most of their components turned off as an attempt to meet the constraints. This phenomenon is called Dark Silicon.

Figure 6.2 presents conservative and optimistic estimations regarding the area, power and frequency scaling on the transition from 45nm to 8nm technologies. These numbers show that the transistor area is greatly reduced, but power does not drop by the same factor.

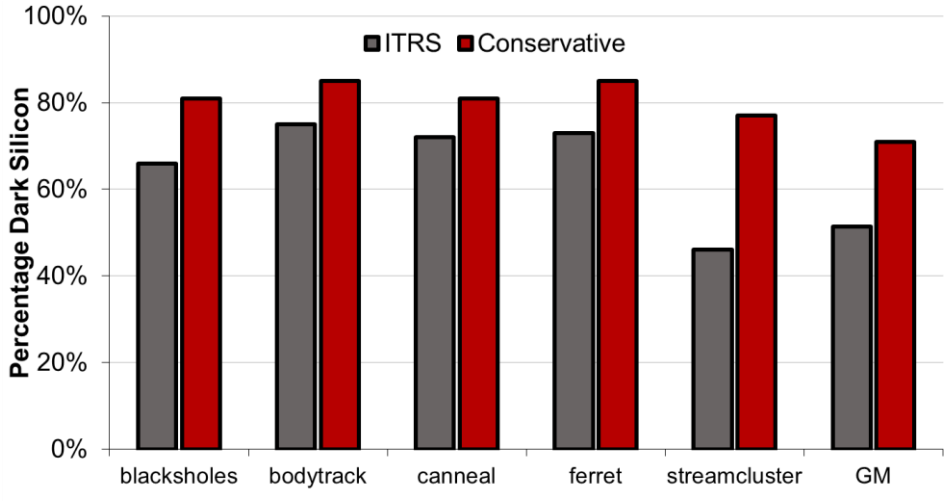
Figure 6.2. Estimations regarding area, power and frequency scaling.

	Conservative	Optimistic
Area	32x ↓	32x ↓
Power	4.5x ↓	8.3x ↓
Frequency	1.3x ↑	3.9x ↑

Source: Esmaeilzadeh (2011, p. 366).

Following the same reasoning, Figure 6.3 presents the percentage of transistors that must remain turned off in order to meet power requirements on chips built with future technologies. These numbers are estimated considering the use of an 8nm technology and, in this case, the number of transistors turned off can reach 80%.

Figure 6.3. Percentage of Dark Silicon area in a chip.



Source: Esmaeilzadeh (2011, p. 367).

Considering this scenario, resource-aware systems become even more attractive. The dark silicon phenomenon simply cut the number of resources available in the system. Therefore, an efficient resource-aware mechanism can help reduce the impact on users' applications. Also, one can consider that some resources can be turned on but only for a brief amount of time (because too much time could lead to problems such as heating and higher energy consumption). In that case, the resource-aware approach may

have policies to activate some part of the chip in strategic moments. The system now has resources that may be available or unavailable depending on the current conditions.

In addition, the dark silicon problem and the variability can be investigated in the same scenario. Heterogeneity (even when forced, in the case of variability) can be welcomed in an environment where resources are scarce.

6.2 List of publications

As a result of the development of this work, many papers were published. Table 6.1 lists these papers and highlight the direct contributions of each of them to the goals of this thesis.

Table 6.1. List of publications

Year		Paper
2009	**Title**	"Performance and energy evaluation of memory hierarchies in NoC-based MPSoCs under latency"
	Venue	International Conference on Very Large Scale Integration (VLSI-SoC)
	Contribution	Development of SIMPLE platform and experiments with distinct memory architectures and cache coherence mechanisms. Initial knowledge on the relation between parallel programming models and memory subsystem.
2011	**Title**	"Performance and Energy Evaluation of Memory Organizations in NoC-based MPSoCs under Latency and Task Migration"
	Venue	Book chapter in VLSI-SoC: Technologies for Systems Integration.
	Contribution	Initial implementations of a task migration mechanism on SIMPLE.
	Title	"Hardware and Software Support for Parallel Programming Models on NoCs"
	Venue	DAC Workshop on Diagnostic Services in Networks-on-Chip (DSNoC)
	Contribution	Implementation of a hardware support for distinct programming models.
	Title	"Dynamic Clustering for Distinct Parallel Programming Models on NoC-based MPSoCs"
	Venue	International Workshop on Network-on-Chip Architectures (NoCArc)
	Contribution	Use of a first resource-aware cluster-based approach (Processor Clustering).
2012	**Title**	"A novel NoC-based design for fault-tolerance of last-level caches in CMPs"
	Venue	International Conference on Hardware/Software Codesign And System Synthesis (CODES+ISSS)
	Contribution	First implementation and experiments on LLC fault-tolerance mechanism using NoC routers.
2013	**Title**	"Exploring resource mapping policies for dynamic clustering on

		NoC-based MPSoCs”
	Venue	Design, Automation and Test in Europe (DATE)
	Contribution	Development of policies for resource-aware approaches.
	Title	“Modeling and analysis of fault-tolerant distributed memories for Networks-on-Chip”
	Venue	Design, Automation and Test in Europe (DATE)
	Contribution	Experiments on LLC fault-tolerance mechanism using redundant memories distributed in cluster-based patterns (Reliability Clustering).
2014	**Title**	“NoC–Based Fault-Tolerant Cache Design in Chip Multiprocessors”
	Venue	Transactions on Embedded Computing Systems
	Contribution	Extensive design space exploration on LLC fault-tolerance using NoCs (new router architectures and routing algorithms)

REFERENCES

ABOUSAMRA, A.; JONES, A.K.; MELHEM, R. Codesign of NoC and Cache Organization for Reducing Access Latency in Chip Multiprocessors. **IEEE Transactions on Parallel and Distributed Systems,** Los Alamitos, v. 23, n. 6, p.1038-1046, jun. 2012.

AGARWAL, A. et al. A process-tolerant cache architecture for improved yield in nanoscale technologies. **IEEE Transaction on VLSI Systems,** Los Alamitos, v. 13, n. 1, p.27–38, jan. 2005.

AGGARWAL, N. ET AL. Configurable isolation: building high availability systems with commodity multi-core processors. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 34, 2007. **Proceedings…** New York, NY: ACM, 2007. p. 470-481.

AL FARUQUE, M.A.; KRIST, R.; HENKEL, J. ADAM: Run-time Agent-based Distributed Application Mapping for on-chip Communication. In: DESIGN AUTOMATION CONFERENCE, 45, 2008. **Proceedings…** Los Alamitos, CA: IEEE, 2008. p. 760-765.

AMDAHL, G.M.. Validity of single-processor approach to achieving large-scale computing capability. In: AFIPS CONFERENCE, 1, 1967. **Proceedings…** Reston, VA:ACM, 1967 p. 483-485.

ANAGNOSTOPOULOS, I. et al. A divide and conquer based distributed run-time mapping methodology for many-core platforms. In: DESIGN AUTOMATION AND TEST IN EUROPE, 15, 2012, **Proceedings…** New York, NY: IEEE Press, 2012. p. 111-116.

ANGIOLINI, F. et al. Reliability Support for On-Chip Memories Using Networks-on-Chip. In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN, 25, 2007. **Proceedings…** Los Alamitos: IEEE press, 2007. p. 389-396.

BAK, S. et al. Memory-Aware Scheduling of Multicore Task Sets for Real-Time Systems. In: INTERNATIONAL CONFERENCE ON EMBEDDED AND REAL-TIME COMPUTING SYSTEMS AND APPLICATIONS, 18, 2012, **Proceedings…** Los Alamitos: IEEE press, 2012. p.300-309.

BANAIYANMOFRAD, A.; HOMAYOUN, H.; DUTT, N. FFT-Cache: A Flexible Fault-Tolerant Cache architecture for ultra-low voltage operation. In: INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURES AND SYNTHESIS FOR EMBEDDED SYSTEMS, 6, 2011. **Proceedings…** New York, NY: ACM, 2011. p.95-104.

BANAIYANMOFRAD, A.; GIRÃO, G.; DUTT, N. A novel NoC-based design for fault-tolerance of last-level caches in CMPs. In: INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS, 7, 2012. **Proceedings…** New York, NY: ACM, 2012. p. 63-72.

BANAIYANMOFRAD, A.; GIRÃO, G.; DUTT, N. Modeling and analysis of fault-tolerant distributed memories for Networks-on-Chip. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 16, 2013. **Proceedings…** Los Alamitos: IEEE press, 2013. p.1605-1608.

BANERJEE, A.; MULLINS, R.; MOORE, S., A Power and Energy Exploration of Network-on-Chip Architectures. In: INTERNATIONAL SYMPOSIUM ON NETWORKS-ON-CHIP, 1, 2007. **Proceedings…** Los Alamitos: IEEE press, 2007 p.7-9.

BARCELOS, D.; BRIÃO, E. W.; WAGNER, F. R. A Hybrid Memory Organization to Enhance Task Migration and Dynamic Task Allocation in NoC-based MPSoCs. In: Symposium on Integrated Circuits and Systems Design, 20, 2007. **Proceedings…** New York, NY: ACM, 2007. p. 282-287.

BATHEN, L.A.D.; DUTT, N.D. E-RoC: Embedded RAIDs-on-Chip for low power distributed dynamically managed reliable memories. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 14, 2011. **Proceedings…** Los Alamitos: IEEE press, 2011. p.1-6.

BECKER, J. et al. Hardware prototyping of novel invasive multicore architectures. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, 17, 2012, 2012. **Proceedings…** Los Alamitos: IEEE press, 2012. p.201-206.

BECKMANN, B. M.; WOOD, D. A. Managing wire delay in large chip-multiprocessor caches. In: INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 37, 2004. **Proceedings…** Washington, DC : IEEE Computer Society, 2004. p. 319-330.

BENINI, L.; DE MICHELI G. Networks on Chips: a New SoC Paradigm, **IEEE Computer**, Los Alamitos, v. 35, n. 1, p. 70-78, jan. 2002.

BIENIA, C. et al. The PARSEC benchmark suite: characterization and architectural implications. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 17, 2008. **Proceedings…** New York, NY: ACM, 2008. p.72-81.

BOGDAN, P.; DUMITRAS, T.; MARCULESCU, R. Stochastic communication: A new paradigm for fault-tolerant networks-on-chip. In: ANNUAL SYMPOSIUM ON VLSI, 2, 2003. **Proceedings…** Washington, DC: IEEE Computer society, 2003. p. 8-12.

BORKAR, S. Thousand core chips: a technology perspective. In: DESIGN AUTOMATION CONFERENCE, 44, 2007. **Proceedings…** Los Alamitos: IEEE press, 2007. p. 746-749.

BRANDENBURG, B.; ANDERSON, J. Optimality results for multiprocessorreal-time locking. In: REAL-TIME SYSTEMS SYMPOSIUM, 31, 2010. **Proceedings…** Los Alamitos: IEEE press, 2010. p. 49-60.

BRIÈRE, M. et al. Heterogeneous modeling of an optical network-on-chip with SystemC. In: INTERNATIONAL WORKSHOP ON RAPID SYSTEM

PROTOTYPING, 16, 2005. **Proceedings…** Washington, DC : IEEE Computer Society, 2005. p. 10-16.

BRUSCHI, F.; MIELE, A.; RANA, V. On-chip network resource management design and validation. In: INTERNATIONAL CONFERENCE ON EMBEDDED COMPUTER SYSTEM, 13, 2011. **Proceedings…** Los Alamitos: IEEE press, 2011. p. 249–254.

CALHOUN, B.; CHANDRAKASAN, A. A 256 kb sub-threshold sram in 65nm cmos. In: INTERNATIONAL SOLID-STATE CIRCUITS CONFERENCE, 41, 2006. **Proceedings…** Washington, DC: IEEE Computer society, 2006. p. 2592-2601.

CARARA, E. A.; ET AL. HeMPS - A Framework for Noc-Based MPSoC Generation. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, 41, 2009. **Proceedings…** Los Alamitos: IEEE press, 2009. p. 1345-1348.

CARVALHO, E. et al. Evaluation of static and dynamic task mapping algorithms in NoC-based MPSoCs. In: INTERNATIONAL SYMPOSIUM ON SYSTEM-ON-CHIP, SOC, 11, 2009. **Proceedings…** Los Alamitos: IEEE press, 2009. p.87-90.

CHEN, B.; NEDELCHEV, I. Power compiler: a gate-level power optimization and synthesis system. In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN: VLSI IN COMPUTERS AND PROCESSORS, 14, 1997. **Proceedings…** Los Alamitos: IEEE press, 1997. p.74-79.

CHOU, C-L.; MARCULESCU, R. FARM: Fault-aware resource management in NoC-based multiprocessor platforms. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 14, 2011. **Proceedings…** Los Alamitos: IEEE press, 2011. p. 14-18.

CONCATTO, C. ET AL. NoC Power Optimization Using a Reconfigurable Router, 2009. In: ANNUAL SYMPOSIUM ON VLSI, 8, 2009. **Proceedings…** Washington, DC: IEEE Computer society, 2009 p.213-215.

CUI, Y; ZHANG, W; YU, H. Decentralized Agent Based ReClustering for Task Mapping of Tera-Scale Network-on-Chip System. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, 44, 2012. **Proceedings…** Los Alamitos: IEEE press, 2012. p. 2437-2440.

DEHYADEGARI, M. et al., Evaluation of Pseudo Adaptive XY Routing Using an Object Oriented Model for NOC. In: INTERNATIONAL CONFERENCE ON MICROELECTRONICS, 17, 2005. **Proceedings…** Los Alamitos: IEEE press, 2005. p. 204-206.

DEHYADEGARI, M.; et al. A tightly-coupled multi-core cluster with shared-memory HW accelerators. In: INTERNATIONAL CONFERENCE ON EMBEDDED COMPUTER SYSTEMS, 14, 2012. **Proceedings…** Los Alamitos: IEEE press, 2012. p.96,103.

DUMMLER, J.; RAUBER, T.; RUNGER, G., Mapping Algorithms for Multiprocessor Tasks on Multi-Core Clusters. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 37, 2008. **Proceedings…**, New York, NY: ACM 2008. p.9-12.

DUTT Research Group. Available at: <https://duttgroup.ics.uci.edu/>. Accessed on: 25 mar. 2013.

DUTT, N. et al. Variability-Aware Memory Management for Nanoscale Computing. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE (ASP-DAC), 18, 2013. **Proceedings…** Los Alamitos: IEEE press, 2013. pp 125-132.

ESMAEILZADEH, H. et al. Dark Silicon and the End of Multicore Scaling. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 38, 2011. **Proceedings…** New York, NY: ACM, 2011. p. 365-376.

FLYNN, M., Very high-speed computing systems. **Proceedings of the IEEE**, New York, v. 54, n. 12, p.1901-1909, Dec. 1966.

FORSELL, M. A Scalable High-Performance Computing Solution for Networks on Chips. **IEEE Micro**, v. 5, n. 22, p. 46-55, Sep. 2002.

GAISLER, J. The Leon Processor's User Manual. 2001. Available at: <http://www.gaisler.com>. Accessed on: 30 mar. 2014.

GIRÃO, G. et al. 2007. Cache coherency communication cost in a NoC-based MPSoC platform. In: CONFERENCE ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 20, 2007. **Proceedings…** New York, NY: ACM, 2007. p. 288-293.

GIRAO, G.; BARCELOS, D.; WAGNER, F.R., Performance and energy evaluation of memory hierarchies in NoC-based MPSoCs under latency. In: INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, 17, 2009. **Proceedings…** Los Alamitos: IEEE press, 2009, p. 12-14.

GIRÃO, G.; BARCELOS, D.; WAGNER, F.R. Performance and Energy Evaluation of Memory Organizations in NoC-based MPSoCs under Latency and Task Migration. In: J.BECKER, J.; JOHANN, M.; REIS, R. (ed.), **VLSI-SoC: Technologies for Systems Integration**. 1 ed. New York, NY: Springer, 2011. p. 56-80.

GIRÃO, G.; WAGNER, F.R. Hardware and Software Support for Parallel Programming Models on NoCs. In: DAC WORKSHOP ON DIAGNOSTIC SERVICES IN NETWORKS-ON-CHIP, Available at: <http://www.dsnoc.org/files/DSNoC2011_digest_final.pdf>. Accessed on: 30 mar. 2014

GIRÃO, G.; SANTINI, T.C.; WAGNER, F.R.. Dynamic Clustering for Distinct Parallel Programming Models on NoC-based MPSoCs. In: INTERNATIONAL WORKSHOP ON NETWORK ON CHIP ARCHITECTURES, 4, 2011. **Proceedings…** New York, NY: ACM, 2011. p. 63-68.

GIRÃO, G.; SANTINI, T.; WAGNER, F. R. Exploring resource mapping policies for dynamic clustering on NoC-based MPSoCs. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 16, 2013. **Proceedings…** Los Alamitos: IEEE press, 2013. p.681-684.

GUTHAUS, M.R. et al. MiBench: A free, commercially representative embedded benchmark suite. In: INTERNATIONAL WORKSHOP ON WORKLOAD CHARACTERIZATION, 4, 2001. **Proceedings…** Washington, DC: IEEE press, 2001. p.3-14.

HANNIG, F.; DUTTA, H.; TEICH, J. Mapping a Class of Dependence Algorithms to Coarse-grained Reconfigurable Arrays: Architectural Parameters and Methodology. **International Journal of Embedded Systems**, Geneva, v. 2, n. 1, p. 114–127, jul. 2006.

HEISSWOLF, J. et al. Hardware-assisted Decentralized Resource Management for Networks on Chip with QoS. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM WORKSHOPS & PHD FORUM, 28, 2012. **Proceedings…** Los Alamitos: IEEE press, 2012. p.21-25.

HELD, J. Single-chip cloud computer: an IA tera-scale research processor. In: CONFERENCE ON PARALLEL PROCESSING, 16, 2010. **Proceedings…** Berlin: Springer-Verlag, 2010. p.85-85.

HOMAYOUN, H. et al. RELOCATE: register file local access pattern redistribution mechanism for power and thermal management in out-of-order embedded processor. In: INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE EMBEDDED ARCHITECTURES AND COMPILERS, 5, 2010. **Proceedings…** Berlin: Springer-Verlag, 2010. p. 216-231.

HOSKOTE, Y.. Teraflop Prototype Processor with 80 Cores. **SYMPOSIUM ON HIGH PERFORMANCE CHIPS**. Available at: <http://www.hotchips.org/wp-content/uploads/hc_archives/hc19/2_Mon/HC19.03/HC19.03.02.pdf >. Accessed on: 3 Apr. 2014.

HU, J.; MARCULESCU, R. Energy- and Performance-Aware Mapping for Regular NoC Architectures. **IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems,** Los Alamitos, v. 24, n. 4, p. 551-562, Apr. 2005.

INTEL, The Intel Xeon Phi™ Coprocessor. Available at: <http://www.intel.com/content/www/us/en/high-performancecomputing/ high-performance-xeon-phi-coprocessor-brief.html>. Accessed on: 3 Apr. 2014.

JALEEL, A. Memory characterization of workloads using instrumentation-driven simulation - a pin-based memory characterization of the spec cpu2000 and spec cpu2006 benchmark suites. VSSAD, Technical Report, 2007. Available at: <http://http://www.glue.umd.edu/~ajaleel/workload/>. Accessed on: 30 mar. 2014.

JIN, X.; SONG, Y.; ZHANG, D. FPGA prototype design of the computation nodes in a cluster based MPSoC. In: INTERNATIONAL CONFERENCE ON ANTI-COUNTERFEITING SECURITY AND IDENTIFICATION IN COMMUNICATION, 4, 2010. **Proceedings…** New York, NY: IEEE press, 2010. p. 18-20.

KANG, K.; BENINI, L.; MICHELI, G.D., A high-throughput and low-latency interconnection network for multi-core Clusters with 3-D stacked L2 tightly-coupled data memory. In: INTERNATIONAL CONFERENCE ON VLSI AND SYSTEM-ON-CHIP, 20, 2012. **Proceedings…** Los Alamitos: IEEE press, 2012. p.7-10.

KAHNG, A.B. et al. ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 12, 2009. **Proceedings…** Los Alamitos: IEEE press, 2009. p.423-428.

KIM, C.; BURGER, D.; KECKLER, S. W. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 10, 2002. **Proceedings…** New York, NY: ACM, 2002. p. 211-222.

KIM, J.; NICOPOULOS, C.; PARK, D. A Gracefully Degrading and Energy-Efficient Modular Router Architecture for On-Chip Networks. In: INTERNATIONAL

SYMPOSIUM ON COMPUTER ARCHITECTURE, 33, 2006. **Proceedings…** New York, NY: ACM, 2006. p. 4-15.

KISSLER, D. A Highly Parameterizable Parallel Processor Array Architecture. In: INTERNATIONAL CONFERENCE ON FIELD PROGRAMMABLE TECHNOLOGY, 5, 2006. **Proceedings…** Washington, DC : IEEE Computer Society, 2006. p. 105–112.

KOBBE, S. et al. DistRM: Distributed Resource Management for On-Chip Many-Core Systems. In: INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS, 6, 2011. **Proceedings…** New York, NY: ACM, 2011, p. 119-128.

KUMAR, R. et al, Bidirectional interconnect design for low latency high bandwidth NoC. In: INTERNATIONAL CONFERENCE ON IC DESIGN & TECHNOLOGY, 6, 2013. **Proceedings…** Los Alamitos: IEEE press, 2013, p. 29-31.

KWON, S. et al. A Retargetable Parallel-Programming Framework for MPSoC. **ACM Transactions on Design Automation of Electronic Systems**, New York, v. 13, n. 3, p. 38-66, jul 2008.

LATIF, K.; et al. Resource-aware task allocation and scheduling for segbus platform. In: INTERNATIONAL CONFERENCE ON ELECTRONICS, CIRCUITS, AND SYSTEMS (ICECS), 17, 2010. **Proceedings…** Los Alamitos: IEEE press 2010 p. 12-15.

LEE, H.G.; et al. On-Chip Communication Architecture Exploration: a Quantitative Exploration of Point-to-Point, Bus and Network-on-chip Architectures. **ACM Transactions on Design Automation of Electronic Systems**, New York, v. 12, n. 3, p. 21-40, aug. 2007.

LIU YAN; L. et al., Performance evaluation of the memory hierarchy design on CMP prototype using FPGA. In: INTERNATIONAL CONFERENCE ON ASIC, 8, 2009. **Proceedings…** Los Alamitos: IEEE press, 2009, p.20-23.

MAGNUSSON, P.S. et al. Simics: A Full System Simulation Platform, **IEEE Computer**, v. 35, n. 2, p. 50-58, feb. 2002.

MANDELLI, M.; CASTILHOS, G.M.; MORAES, F.G. Enhancing performance of MPSoCs through distributed resource management. In: INTERNATIONAL CONFERENCE ON ELECTRONICS, CIRCUITS AND SYSTEMS, 19, 2012. **Proceedings…** Los Alamitos: IEEE press, 2012. p. 9-12.

MANOLACHE, S.; et al. Fault and Energy-Aware Communication Mapping with Guaranteed Latency for Applications Implemented on NoC. In: DESIGN AUTOMATION CONFERENCE, 42, 2005. **Proceedings…** Los Alamitos: IEEE press, 2005. p. 266-269.

MARCULESCU, R. ET AL. Outstanding Research Problems in NoC Design: System. Microarchitecture, and Circuit Perspectives. **IEEE Transactions on CAD**. v. 28, n. 1, p. 3–21, jan. 2009

MARESCAUX, T.; Brockmeyer, E.; Corporaal, H. The impact of higher communication Layers on NoC supported MPSoCs. In: INTERNATIONAL SYMPOSIUM ON NETWORKS-ON-CHIP, 1, 2007. **Proceedings…** Los Alamitos: IEEE Press, 2007. p. 107-116.

MARWEDEL, P. **Embedded System Design**. 1st ed. Kluwer Academic Publishers, 2003.

MASKO, L.; TUDRUJ, M. Task Scheduling for SoC-Based Dynamic SMP Clusters with Communication on the Fly. In: INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED COMPUTING, 22, 2008. **Proceedings…** Los Alamitos: IEEE press, 2008. p. 99-106.

MESSAGE Passing Interface Forum. MPI: A Message Passing Interface Standard. Available at: <http://www.mpi-forum.org>. Accessed on: 30 mar. 2014.

Monchiero, M. Exploration of Distributed Shared Memory Architectures for NoC-based Multiprocessors. In: INTERNATIONAL CONFERENCE ON EMBEDDED COMPUTER SYSTEMS: ARCHITECTURES, MODELING AND SIMULATION, 9, 2006. **Proceedings…** Los Alamitos: IEEE press, 2006, p. 144-151.

MORRIS, M.F. Kiviat graphs: conventions and figures of merit. **ACM SIGMETRICS Performance Evaluation Review**. New York, v.3, n. 3, p. 2-8, oct. 1974.

MURALI, S.; et al. A methodology for mapping multiple use-cases onto networks on chips. In: DESIGN AUTOMATION AND TEST IN EUROPE, 9, 2006. **Proceedings…** Washington, DC : IEEE Computer Society, 2006. p. 118-123.

NASSIF, S. R.; MEHTA, N.; CAO, Y. A resilience roadmap. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 13, 2010. **Proceedings…** Los Alamitos: IEEE press, 2010. p. 1011-1016.

Nollet, V. et al. Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles. In: DESIGN AUTOMATION AND TEST IN EUROPE, 8, 2005. **Proceedings…** Washington, DC : IEEE Computer Society, 2005. p. 234-239.

OPENMP Architecture Review Board, OpenMP C and C++ Application Program Interface, Version 2.0, March, 2002. Available at: <http://www.openmp.org>. Accessed on: 30 mar. 2014

OGRAS, U.Y. et al, Voltage-Frequency Island Partitioning for GALS-based Networks-on-Chip. In: DESIGN AUTOMATION CONFERENCE, 44, 2007. **Proceedings…** Los Alamitos: IEEE press, 2007 p. 4-8.

ORSILA, H.; et al. Automated Memory-Aware Application Distribution for Multi-Processor System-On-Chips. **Journal of Systems Architecture**, Amsterdam, v. 53, n. 11, p. 795-815, nov. 2007

PARK, D. ET AL. Exploring Fault-Tolerant Network-on-Chip Architectures. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, 7, 2006. **Proceedings…** Washington, DC : IEEE Computer Society, 2006. p. 93-104.

PASTRNAK, M.; DE WITH, P.H.N.; VAN MEERBERGEN, J. Realization of qos management using negotiation algorithms for multiprocessor noc. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, 39, 2006, **Proceedings…** Los Alamitos: IEEE press, 2006. p. 1912–1915.

PIRRETTI, M. et al. Fault tolerant algorithms for network-on-chip interconnect. In: ANNUAL SYMPOSIUM ON VLSI, 3, 2004. **Proceedings…** Washington, DC: IEEE Computer society, 2004. p. 46-51.

PUENTE, V. et al. Immunet: A cheap and robust fault-tolerant packet routing mechanism. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 31, 2004. **Proceedings…** New York, NY: ACM, 2004. p. 198-209.

QIAN, Z. ET AL. A traffic-aware adaptive routing algorithm on a highly reconfigurable network-on-chip architecture. In: INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS, 7, 2012. **Proceedings…** New York, NY, USA: ACM 2012, p.161-170.

RABANI, Y.; SINCLAIR, A.; WANKA, R. Local divergence of Markov chains and the analysis of iterative load-balancing schemes. In: SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, 39, 1998. **Proceedings…** Washington, DC: IEEE Computer society, 1998. p. 694–703.

SANDERS, P. Randomized priority queues for fast parallel access. **Journal Parallel and Distributed Computing, Special Issue on Parallel and Distributed Data Structures**, Amsterdam, v. 49, n. 1, p. 86–97, feb. 1998.

SHABBIR, A. et al. Distributed resource management for concurrent execution of multimedia applications on MPSoC platforms. In: INTERNATIONAL CONFERENCE ON EMBEDDED COMPUTER SYSTEMS, 13, 2011. **Proceedings…** Los Alamitos: IEEE press, 2011. p.132-139.

SILVA JR, E.T. et al. An MPSoC Virtual Platform for Real-Time Embedded Systems. In: INTERNATIONAL WORKSHOP ON JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS, 6, 2008. **Proceedings…** New York, NY: ACM, 2008. p.31-37.

SYSTEMC. SystemC Language Reference Manual. **IEEE Std 1666**. 2006

TAJIK, H.; HOMAYOUN, H.; DUTT, N. VAWOM: Temperature and process variation aware WearOut Management in 3D multicore architecture. In: DESIGN AUTOMATION CONFERENCE, 50, 2013. **Proceedings…** Los Alamitos: IEEE press, 2013. p.1-8.

TAN, J.; et al. A predictive and parametrized architecture for image analysis algorithm implementations on FPGA adapted to multispectral imaging. In: INTERNATIONAL WORKSHOP ON IMAGE PROCESSING THEORY, TOOLS AND APPLICATIONS, 1, 2008. **Proceedings…** Los Alamitos: IEEE press, 2008. pp 1-8.

TEICH, J. et al. Invasive Computing: An Overview. In: HUBNER, M.; BECKER. J. (Ed.) **Multiprocessor System-on-Chip**. 1st ed. New York, NY: Springer, 2011. p. 241-268.

TER BRAAK, T.D. et al. Run-time spatial resource management for real-time applications on heterogeneous MPSoCs. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 14, 2011. **Proceedings…** Los Alamitos: IEEE press, 2011. p. 357-362.

THOMAS, A., BECKER, J. New adaptive multi-grained hardware architecture for processing of dynamic function patterns. **IT - Information Technology**. Berlin, v. 49, n. 3, p.165–173, may 2007.

TOTA, S. et al. MEDEA: a hybrid Shared Memory/Message Passing Multiprocessor NoC-based Architecture. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 13, 2010. **Proceedings…** Los Alamitos: IEEE press, 2010. p. 45-50.

TSAI, W. et al. A fault-tolerant NoC scheme using bidirectional channel. In: DESIGN AUTOMATION CONFERENCE, 48, 2011. **Proceedings…** Los Alamitos: IEEE press, 2011. p. 918-923.

UREÑA, I.A.C. Invasive MPI on intel's single-chip cloud computer. In: INTERNATIONAL CONFERENCE ON ARCHITECTURE OF COMPUTING SYSTEMS, 25, 2012. **Proceedings…** Berlin: Springer-Verlag, 2012. p.74-85.

WANG, Y. et al. Address Remapping for Static NUCA in NoC-Based Degradable Chip-Multiprocessors. In: PACIFIC RIM INTERNATIONAL SYMPOSIUM ON DEPENDABLE COMPUTING, 16, 2010. **Proceedings…** Los Alamitos: IEEE press, 2010. p. 70-76.

WEICHSLGARTNER, A.; WILDERMANN, S.; TEICH J. Dynamic decentralized mapping of tree-structured applications on NoC architectures. In: INTERNATIONAL SYMPOSIUM ON NETWORKS-ON-CHIP, 5, 2011. Proceedings… Los Alamitos: IEEE press, 2011. p. 201-208.

WILTON, S.; JOUPPI, N. Cacti: An enhanced cache access and cycle time model, **IEEE Journal of Solid State Circuits**, Washington, DC, v. 31, n. 5, p.677-688, may 1996.

WINTER, M.; FETTWEIS, G. Guaranteed service virtual channel allocation in nocs for run-time task scheduling. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 14, 2011. **Proceedings…** Los Alamitos: IEEE press, 2011. p. 1–6.

WOO, S.C. et al. The SPLASH-2 programs: characterization and methodological considerations. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 22, 1995. **Proceedings…** New York, NY: ACM, 1995. p. 24-36.

XU, B.; ALBONESI, D.H. Runtime Reconfiguration Techniques for Efficient General-Purpose Computation. **IEEE Design & Test,** Los Alamitos, v. 1, n. 17, p.42-52, jan 2000.

ZEFERINO, C.A.; SUSIN, A.A. SoCIN: A Parametric and Scalable Network-on-Chip. In: CONFERENCE ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 16, 2003. **Proceedings…** Washington, DC:ACM, 2003. p. 169-174.

ZHANG, D-L.; XIA, Y.; DU, G-M.; HOU, N, A study of communication performance in the infrastructure of cluster-based MPSoC. In: INTERNATIONAL CONFERENCE ON ANTI-COUNTERFEITING, SECURITY AND IDENTIFICATION, 6, 2012. **Proceedings…** New York, NY: IEEE press, 2012. p.1-5.ZHANG, X.; RABAH, H.; WEBER, S., Cluster-Based Hybrid Reconfigurable Architecture for Auto-adaptive SoC. In: INTERNATIONAL CONFERENCE ON ELECTRONICS, CIRCUITS AND SYSTEMS, 14, 2007. **Proceedings…** Los Alamitos: IEEE press, 2007. p. 11-14.