PUCRS

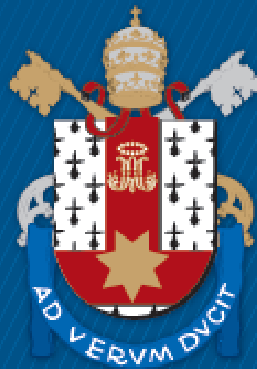
ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

ANGELO ELIAS DALZOTTO

PROPOSAL OF MANY-CORE CONTROL THROUGH A MANAGEMENT APPLICATION

Porto Alegre
2022

PÓS-GRADUAÇÃO - *STRICTO SENSU*



Pontifícia Universidade Católica do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

PROPOSAL OF MANY-CORE CONTROL THROUGH A MANAGEMENT APPLICATION

ANGELO ELIAS DALZOTTO

Master Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Fernando Gehm Moraes
Co-Advisor: Prof. Dr. Marcelo Ruaro

Porto Alegre
2022

Ficha Catalográfica

D153p Dalzotto, Angelo Elias

Proposal of many-core control through a management application / Angelo Elias Dalzotto. – 2022.
108 p.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Fernando Gehm Moraes.
Coorientador: Prof. Dr. Marcelo Ruaro.

1. Many-core. 2. Management application. 3. ODA. 4. Broadcast. 5. RISC-V. I. Moraes, Fernando Gehm. II. Ruaro, Marcelo. III. , . IV. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

ANGELO ELIAS DALZOTTO

PROPOSAL OF MANY-CORE CONTROL THROUGH A MANAGEMENT APPLICATION

This Master Thesis has been submitted in partial fulfillment of the requirements for the degree of Master in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on February 24, 2022.

COMMITTEE MEMBERS:

Prof. Dr. Everton Carara (PPGCC/UFSM)

Prof. Dr. Ney Laert Vilar Calazans (PPGCC/PUCRS)

Prof. Dr. Marcelo Ruaro (Université Bretagne Sud - Co-Advisor)

Prof. Dr. Fernando Gehm Moraes (PPGCC/PUCRS - Advisor)

AGRADECIMENTOS

Primeiramente, agradeço a Deus por me tornar capaz de superar os desafios da vida e dos estudos para atingir de forma satisfatória mais essa conquista.

Agradeço à minha família: meu pai, Everton; minha mãe, Silvana; e meu irmão, Marco Antônio, por todo o carinho e apoio incondicional propiciado em todo esse processo.

Agradeço à minha namorada, Íngridy, por todo o amor e paciência. Agradeço toda a sua dedicação a mim, por ter me mantido são durante esse período do mestrado.

Também agradeço meus avós, sogra, cunhada, tios, e primos, por sempre permanecermos unidos e por descontraírem meus fins de semana e, em alguns momentos, entenderem minha ausência necessária para trilhar essa jornada com o máximo de dedicação e empenho.

Agradeço ao professor Fernando Moraes por todos os ensinamentos e por ter aceitado o desafio de me orientar de forma completamente remota, proporcionando meu melhor desempenho nesse processo. Sem seu esforço este trabalho não teria sido possível.

Agradeço também ao Marcelo Ruaro, coorientador desta dissertação, pelo aprendizado a mim transmitido e pelo seu esforço, mesmo de muito longe.

Agradeço ao professor Ney Calazans, pelas inúmeras contribuições durante as avaliações desta dissertação.

Agradeço aos bolsistas de iniciação científica, Leonardo Vian Erthal, e Caroline da Silva Borges, pela dedicação às suas atribuições. Suas pesquisas acerca das heurísticas de mapeamento e de desfragmentação, respectivamente, foram de grande contribuição para esta dissertação.

Por fim, agradeço ao CNPq pelo suporte financeiro.

PROPOSTA DE CONTROLE DE MANY-CORES ATRAVÉS DE UMA APLICAÇÃO DE GERENCIAMENTO

RESUMO

A grande quantidade de *cores* em sistemas *many-core* introduziu desafios para gerenciá-los, incluindo escalabilidade, portabilidade e redução da interferência do gerenciamento sobre as aplicações em execução. Trabalhos disponíveis na literatura propõem um gerenciamento fortemente acoplado ao sistema operacional do *many-core*. Tal acoplamento implica em baixa flexibilidade para a modificação das organizações de gerenciamento, e baixa portabilidade. O estado-da-arte também mostra que poucos trabalhos propuseram organizações de gerenciamento, sendo que a maioria dos trabalhos se aproveitam de organizações presentes na literatura, como as baseadas em *clusters* e por-aplicação, para avaliar a qualidade de uma única meta proposta, como, por exemplo, potência ou temperatura. O presente trabalho propõe uma organização de gerenciamento, denominada de Aplicação de Gerenciamento (MA), que é fracamente acoplada à sua plataforma alvo. A MA propõe um gerenciamento como uma aplicação distribuída, permitindo que a mesma se beneficie do poder de processamento paralelo intrínseco aos many-cores. Comparado a uma organização baseada em *clusters*, os custos e os benefícios em gerenciar um *benchmark* com restrições de tempo-real usando a MA revelam menor ocupação de memória e maior vazão de gerenciamento devido à paralelização provida pela MA. Esse trabalho também propõe uma heurística de mapeamento que separa virtualmente o espaço de busca em *clusters* para reduzir o custo de execução, mantendo uma visão centralizada do sistema. Essa heurística também conta com um procedimento de desfragmentação embutido. Resultados são avaliados contra uma heurística do estado-da-arte em gerenciamento baseado em *clusters* e por-aplicação, revelando redução na distância média entre tarefas comunicantes e tempo de execução da heurística similar à abordagem baseada em *clusters*. A desfragmentação usa a migração de tarefas como meio de atuação, conseguindo reduzir a distância entre tarefas comunicantes usando poucas migrações. O arcabouço da MA é otimizado com a integração de uma rede intrachip baseada em *broadcast*, usada para troca de mensagens de gerenciamento, e com a adição de uma estrutura de monitoramento que se aproveita dessa rede e do mecanismo de acesso direto à memória para reduzir a sobrecarga de monitoramento. A rede reduz a interferência na comunicação das aplicações de usuário e melhora o tempo de execução, enquanto a estrutura de monitoramento permite menor latência de gerenciamento. Por fim, a organização MA é aplicada a uma plataforma equipada com um processador RISC-V, reduzindo o número de instruções executadas e o uso de memória. O resultado final é uma plataforma many-core que implementa a organização MA com um processador do estado-da-arte.

Palavras-Chave: many-core, Aplicação de Gerenciamento, ODA, broadcast, RISC-V.

PROPOSAL OF MANY-CORE CONTROL THROUGH A MANAGEMENT APPLICATION

ABSTRACT

The increasing core count in many-core systems introduced management challenges, including scalability, portability, and reducing the management overhead to user applications. Works available in the literature have their management tightly coupled to the many-core operating system. This coupling implies low flexibility for modification of the management organizations and reduced portability. The state-of-the-art also shows that few works proposed management organizations, being that most works exploit organizations present in the literature, such as cluster-based and per-application, to evaluate the quality of a single goal, such as power or temperature. The present work proposes a management organization called Management Application (MA), which is loosely coupled to its target platform. MA proposes a management as a distributed application, benefiting from the parallel processing power intrinsic to many-cores. Compared to a cluster-based organization, the costs and benefits to manage a benchmark with real-time constraints using the MA revealed improved memory footprint and higher management throughput due to the parallelization provided by the MA. This work also proposes a mapping heuristic that virtually separates the search space in clusters to reduce the execution cost, keeping a centralized view of the system. This heuristic also has a built-in defragmentation procedure. Results are evaluated against a state-of-the-art heuristic in clustered and per-application management, revealing reduced distance between communicating tasks and similar heuristic execution time to the clustered approach. Defragmentation uses task migration as actuation means, decreasing the distance between communicating tasks using few migrations. The MA framework is optimized by integrating a broadcast-based network-on-chip, used for exchanging management messages, and a monitoring structure that exploits this network and the direct memory access mechanism to reduce the monitoring overhead. The network reduces the interference in user applications and the execution time, while the monitoring structure allows smaller management latency. Lastly, the MA organization is applied to a platform equipped with a RISC-V processor, reducing the number of executed instructions and the memory footprint. The final result is a many-core platform that implements the MA organization with a state-of-the-art processor.

Keywords: many-core, Management Application, ODA, broadcast, RISC-V.

LIST OF FIGURES

1.1 Document organization, highlighting the main work contributions. 20
2.1 An adaptive resource management scheme. The workload and system variations trigger a goal switcher that changes the management objectives at runtime. The different goals can be conflicting, such as “System Throughput" and “Power and Energy". Source: [Rahmani et al., 2018b]. 22
2.2 Management organizations present in the literature. 23
2.3 How a typical system is partitioned. Source: [Tanenbaum and Bos, 2014]. . . 23
2.4 The Intel x86 Ring Architecture. Source: [Reid and Caelli, 2005]. 24
2.5 Monolithic (left) vs. microkernel (right) privileges. Source: [Biggs et al., 2018]. 25
2.6 Memphis many-core overview. Adapted from: [Ruaro et al., 2019a]. 26
2.7 Overview of Memphis kernels organization. Adapted from: [Ruaro et al., 2019a]. 26
2.8 Sequence diagram of the Application Injector protocol. Adapted from: [Ruaro et al., 2019a]. 27
3.1 Many-core taxonomy for the related work. 29
3.2 ARTE many-core structure. Source: [Mariani et al., 2013]. 32
3.3 RTM components in DRACON. **HMST** – Hardware Master, **HLSV** – Hardware Slave. Source: [Gregorek et al., 2019]. 33
3.4 Different PAM organizations. 34
4.1 MA organization. **O** – Observation, **D** – Decision, **A** – Actuation. 38
4.2 ODA Model used by MA paradigm. There is one LLM for each core. The AE is implemented by the OS running in each core. Arrows represent the communication between the entities. **MCSoC** – Many-core SoC. Source: [Ruaro et al., 2021]. 39
4.3 Sequence diagram of the new message-passing API for management communication. 42
4.4 Format of the communication address. **K** – Kernel message, **E** – Port to force exit at the NoC borders, enabling the communication with peripherals. 43
4.5 Application Injector protocol sequence diagram. 44
4.6 Management binaries size: CBM, MA, CBM kernel, MA kernel. 46
4.7 The MA pipeline model. The number of LLMs and AEs is one per PE. Observer tasks are defined at design time, while the number of Deciders and Actuators are a function of the management objectives. 49

4.8 Delay from monitoring message emission until task migration completion in CBM and MA. 50
5.1 BrNoC architecture. Source: [Wachter et al., 2017]. 52
5.2 BrLite message header (40 bits). The total flit length is 80 bits with the 40-bit payload. 54
5.3 Control fields in BrLite CAM. 54
5.4 Updated DMNI architecture. **LUT** – Lookup Table 55
5.5 Monitoring table line. 56
5.6 A single LUT line (*LL*). 56
5.7 Reacting times for Hermes-only and BrLite with monitoring framework scenarios. 62
5.8 Communication volume in each NoC for the both evaluated scenarios. A BrLite flit, 80 bits, is normalized to 32 bits, i.e., multiplied by 2.5. 63
5.9 Reacting times for Hermes-only and BrLite with monitoring framework equipped scenarios. 64
6.1 A virtual cluster, or window, in the many-core. $Wx = 3$, $Wy = 3$, and $Stride = 2$. 67
6.2 Virtual window sliding in a 8x7 many-core, $Wx = 3$, $Wy = 3$, and $stride = 2$. . 69
6.3 Application modeled as a CTG. 71
6.4 Execution of the mapping algorithm related to CTG presented in Figure 6.3, for $Wx = 3$ and $Wy = 3$. 74
6.5 Mapping validation on a 8x8 many-core, $Wx = 3$, $Wy = 3$, and $stride = 2$. . . . 75
6.6 Mapping validation on a 6x6 many-core with multitasking (2 tasks per PE), $Wx = 3$, $Wy = 3$, and $stride = 2$. 76
6.7 Mapping heuristic comparison between CBM, PAM, and MA. 79
6.8 Mapping heuristic latency for different sized applications in CBM, PAM and sliding window mapper. 80
6.9 Fragmentation in a 10x10 many-core. The application in orange is fragmented, with its bounding box, in red, disturbed by the pink, purple, and blue applications. 81
6.10 Defragmentation in a 10x10 many-core. The application in orange is defragmented, with its bounding box, in red, now smaller and not disturbed by the remaining applications. 84
6.11 Mapping states before defragmentation (a), after defragmentation (b,c), and with an extra application after defragmentation (d). 84

7.1 Privilege stack implemented by the RISC-V ISS. Adapted from: [Waterman et al., 2016b]. **ABI** – Application Binary Interface. **SBI** – Supervisor Binary Interface. 88
7.2 The `mrar` register. 88
7.3 Kernel bootloader assembly code. 90
7.4 Kernel trap handling assembly code. 91
7.5 Kernel interrupt handling assembly code. 92
7.6 Kernel environment call handling assembly code. 94
7.7 Kernel exception handling assembly code. 95
7.8 Application binary interface assembly code. 95
7.9 Total executed instructions of RISC-V vs. MIPS I for each application scenario. 96
7.10 Executed instruction breakdown for RISC-V and MIPS I. 97
7.11 Memory footprint for different applications in RISC-V and MIPS I. 98

LIST OF TABLES

3.1 Related work on management organizations. 30
4.1 Message exchange in CBM and MA. 47
4.2 Timestamps for the Dijkstra's application using CBM and MA (ms). 48
5.1 Services available in BrLite. 53
5.2 Functions available by the BrLite MA API. 58
5.3 Messages sent through BrLite. 59
5.4 Applications used to evaluate the monitoring framework. 63
6.1 Execution of the task mapping ordering algorithm, using Figure 6.3 as input. 72
6.2 Complexity of each phase of the heuristic. W – Window size in one dimension, N – number of application tasks, $|PE|$ – number of PEs. 76
6.3 Number of tasks of the applications used in the scenarios. 78

LIST OF ALGORITHMS

6.1 Window selection algorithm. 69
6.2 Task mapping ordering algorithm. 71
6.3 Task mapping algorithm. 73
6.4 Defragmentation algorithm. 83

LIST OF ACRONYMS

ABI – Application Binary Interface

ADAM – Agent-based Distributed Application Mapping

AE – Actuation Enforcer

AES – Advanced Encryption Standard

API – Application Programming Interface

ARTE – Application-specific Run-Time managEment

ASIC – Application-Specific Integrated Circuit

BRNOC – BRoadcast Network On Chip

CBM – Cluster Based Management

CAM – Content-Addressable Memory

CPU – Central Processing Unit

CSR – Control and Status Register

CSRR – Control and Status Register Read

CSRR – Control and Status Register Write

CTG – Communication Task Graph

DMA – Direct Memory Access

DMNI – Direct Memory Network Interface

DPM – Dynamic Power Management

DTW – Dynamic Time Warping

DVFS – Dynamic Voltage and Frequency Scaling

FIFO – First In, First Out

FPGA – Field-Programmable Gate Array

GCC – GNU C Compiler

GM – Global Manager

GP – Global Pointer

HAL – Hardware Abstraction Layer

HEMPS – Hermes Multiprocessor Systems

HMST – Hardware MaSTer

HSLV – Hardware SLaVe

I-FSM – Input Finite-State Machine

I/O – Input/Output

ID – Identification

ILP – Instruction Level Parallelism
IOT – Internet of Things
IPC – Inter-Process Communication
ISA – Instruction Set Architecture
ISS – Instruction Set Simulator
JPEG – Joint Photographic Experts Group
LL – LUT Line
LLM – Low-Level Monitor
LM – Local Manager
LRU – Least Recently Used
LUT – LookUp Table
MA – Management Application
MCSOC – Many-Core System-On-Chip
MEMPHIS – Many-core Modeling Platform for Heterogeneous SoCs
MEPC – Machine Exception Program Counter
MMR – Memory-Mapped Register
MPEG – Moving Picture Experts Group
MPI – Message-Passing Interface
MPSOC – Multiprocessor System On Chip
MRAR – Machine Relative Addressing Register
MRET – Machine RETurn
MSCRATCH – Machine SCRATCH
MTL – Monitoring Table Line
MWD – Multi-Window Display
NI – Network Interface
NOC – Network-On-Chip
O-FSM – Output Finite-State Machine
ODA – Observe, Decide, Act
OS – Operating System
PAM – Per Application Management
PE – Processing Element
PS – Packet-Switching
QOS – Quality of Service
RA – Return Address

RISC – Reduced Instruction Set Computer
RL – Reinforcement Learning
RTL – Register-Transfer Level
RTM – Run-Time Management
RT – Real Time
SESC – SupErSCalar Simulator
SCC – Single-Chip Cloud Computer
SCT – Supervisory Control Theory
SOC – System on Chip
SP – Stack Pointer
SPARC – Scalable Processor Architecture
TCB – Task Control Block
TDP – Thermal Design Power
TSP – Thermal Safe Power
VHDL – VHSIC Hardware Description Language
VHSIC – Very High Speed Integrated Circuit
VOPD – Video Object Plane Decoder

CONTENTS

1 INTRODUCTION . . . 17
1.1 MOTIVATION . . . 18
1.2 TARGET PLATFORM . . . 19
1.3 OBJECTIVES . . . 19
1.4 DOCUMENT ORGANIZATION . . . 20

2 BASIC CONCEPTS . . . 21
2.1 MANY-CORE MANAGEMENT GOALS . . . 21
2.2 MANY-CORE MANAGEMENT ORGANIZATION . . . 22
2.3 KERNEL DESIGNS . . . 23
2.4 MANY-CORE MODELING PLATFORM FOR HETEROGENEOUS SOCS (MEMPHIS) . . . 25

3 RELATED WORK . . . 29
3.1 CENTRALIZED APPROACHES . . . 32
3.2 CBM APPROACHES . . . 33
3.3 PAM APPROACHES . . . 34
3.4 HYBRID APPROACHES . . . 35
3.5 FINAL REMARKS . . . 36

4 MANAGEMENT APPLICATION . . . 37
4.1 THE MA PARADIGM . . . 37
4.2 PROOF-OF-CONCEPT IMPLEMENTATION . . . 40
4.2.1 KERNEL MODIFICATIONS AND PLATFORM IMPROVEMENTS . . . 40
4.2.2 COMMUNICATION API . . . 41
4.2.3 TASK INJECTION . . . 43
4.2.4 MIGRATION ODA . . . 44
4.3 RESULTS . . . 46
4.3.1 MA IMPLEMENTATION COST . . . 46
4.3.2 MA CASE STUDY . . . 47
4.3.3 MANAGEMENT THROUGHPUT . . . 49
4.4 FINAL REMARKS . . . 49

5 MANAGEMENT APPLICATION WITH A DEDICATED MONITORING FRAMEWORK . . . 51
5.1 FRAMEWORK IMPLEMENTATION . . . 52
5.1.1 BROADCAST NOC . . . 52
5.1.2 MONITORING FRAMEWORK . . . 55
5.2 MANAGEMENT ADAPTATION . . . 57
5.3 RESULTS . . . 60
5.3.1 CLUSTER EVALUATION . . . 61
5.3.2 SCALABILITY EVALUATION . . . 62
5.4 FINAL REMARKS . . . 65

6 MA MAPPING HEURISTIC . . . 66
6.1 WINDOW SELECTION ALGORITHM . . . 68
6.2 TASK MAPPING ORDERING ALGORITHM . . . 70
6.3 TASK MAPPING ALGORITHM . . . 72
6.4 MAPPING RESULTS . . . 75
6.4.1 FUNCTIONAL VALIDATION AND COMPUTATIONAL COMPLEXITY . . . 75
6.4.2 MAPPING QUALITY . . . 77
6.5 DEFRAGMENTATION . . . 80
6.5.1 DEFRAGMENTATION RESULTS . . . 84
6.6 FINAL REMARKS . . . 85

7 RISC-V INTEGRATION . . . 86
7.1 RISC-V HARDWARE SUPPORT . . . 87
7.2 RISC-V SOFTWARE SUPPORT . . . 88
7.2.1 KERNEL BOOTLOADING . . . 89
7.2.2 KERNEL TRAP HANDLING . . . 90
7.2.3 KERNEL INTERRUPT HANDLING . . . 91
7.2.4 KERNEL ENVIRONMENT CALL HANDLING . . . 93
7.2.5 KERNEL EXCEPTION HANDLING . . . 93
7.2.6 APPLICATION BINARY INTERFACE . . . 95
7.3 RESULTS . . . 96
7.4 FINAL REMARKS . . . 97

8 CONCLUSION AND FUTURE WORK . . . 99
8.1 FUTURE WORK . . . 101

1. INTRODUCTION

Many-cores emerged from massive hardware and software technology improvements, posing possible solutions for the power wall, memory wall, and Instruction Level Parallelism (ILP) wall [Manferdelli et al., 2008]. Woo and Lee [Woo and Lee, 2008] following the emergence of quad-core and octa-core processors, predicted the usage of more than 1000 cores in a single platform. Recently, Esperanto Technologies unveiled its new chip for machine learning, which contains a superscalar out-of-order quad-core processor, plus 1089 in-order multithreaded RISC-V cores in a single die [Peckham, 2020].

The increased number of cores results in complex resource allocation problems. These problems are addressed by the *many-core management*. In a many-core, the communication mechanism must serve all cores with reduced delays, and the performance must meet the requirements of tasks that are often real-time ones. Power dissipation must also be under the designed budget, being even more limited in domains such as the Internet of Things (IoT). Temperature should not exceed the physical limits while being appropriate to prevent excessive silicon wear. Using proper management structures, many-core systems are set to meet one or more management objectives that can be conflicting, such as power versus performance.

Following the complexity introduced by large core count and the resulting management challenges, there are also scalability issues. Many-core management scalability has been solved initially by separating the many-core system into clusters, reserving processors to manage these areas. Another approach is to reserve one manager processor for each application running in the system. Both methods present weaknesses. The main drawbacks of these approaches include the specialization of a set of processors for management, requiring a dedicated Operating System (OS), and the lack of modularity.

The many-core increasing requirements also demand the management to be multi-objective, to trade-off conflicting goals. These conflicting goals can vary over time and are hard to change in state-of-the-art platforms due to the management being tightly coupled (see Definition 1) to the OS and hardware. Fast hardware and software evolution demand modularity (see Definition 2) and portability (see Definition 3) so the management organizations can evolve with the platform.

The three concepts regarding the many-core management problems in the literature can be defined as:

Definition 1. Coupling – it is the degree of interaction between modules of a system. The software should have low, or loose, coupling [Pressman and Maxim, 2019]. In the context of operating systems, the coupling measures the dependency between kernel and nonkernel modules [Yu et al., 2004]. Many-cores, between management (nonkernel) and kernel, can be affected by the two highest degrees of coupling: *common coupling* and *content*

coupling. *Common coupling* occurs when two or more modules reference the same global structure, possibly leading to uncontrolled error propagation and to unforeseen side-effects when one of the commonly coupled module changes. *Content coupling* occurs when modules share the same code, violating information hiding [Pressman and Maxim, 2019], i.e., possibly requiring extensive modification if the design of one of the content coupled module changes. Tight coupling is connected to fault-proneness, reduced portability, and maintainability. These problems are largely caused by the phenomenon that dependencies within the code lead to regression faults [Yu et al., 2004].

Definition 2. Modularity – in this work, modularity is defined as the ability to add, modify, or remove management goals from the many-core, either at design time or runtime. Modularity at runtime can be achieved by enabling to allocate and deallocate the set of tasks responsible for managing a goal, allowing to switch the management decision priority based on the system state.

Definition 3. Portability – it is the ability to use a same management organization, goals, and heuristics between distinct many-cores while reusing code. Hardware- and OS-dependent organizations are not suitable for portability, leading to a slow development process and bad quality ad-hoc adaptations. One way to make the management agnostic of the platform is to make it work like an user application running in the many-core. Thus, it can be ported to other platforms by changing only the interactions with lower-level system services as long as a good programming interface between management and OS is defined.

State-of-the-art management organizations reveal few-to-none concerns with modularity or portability. In the literature, few works address a management organization explicitly. These works use consolidated management organizations to evaluate an algorithm quality, a communication strategy, or any other particularity of a platform. The present work does not follow the same objective and proposes a framework for the Management Application (MA) organization, which is loosely coupled to the hardware and OS, targeting modular goals.

Section 1.1 presents the motivations that guide the development of this work. Section 1.2 briefly presents the target platform of this work. Section 1.3 enumerates the goals of the work. Finally, Section 1.4 presents the organization of the remaining of this document.

1.1 Motivation

The motivation for this work is twofold:

1. The Author's previous experience in many-cores showed that system services implemented at the kernel level impose challenges in adding or modifying management

goals. These goals are tightly coupled to the platform services, resulting in a considerable effort to support different hardware architectures than the one targeted by the management, often leading to ad-hoc solutions.

2. The state-of-the-art also does not present portable and modular management organizations. Most of these works focus only on the management objectives quality, without proposing how the management structure is organized. Chapter 3 presents the state-of-the-art and discusses this problem in detail.

1.2 Target Platform

The Memphis platform [Ruaro et al., 2019a], described in Section 2.4, is used as the many-core baseline for the present work. This platform was modified during the MA development to achieve the strategic goal. Modifications were made in the hardware description, written in C++ with SystemC libraries, in the OS kernel code, written in C and assembly, and in the system build tools, written in Python.

1.3 Objectives

The *strategic* objective of this work is to provide a Management Application (MA) framework for many-core management employing the Observe, Decide, Act (ODA) paradigm [Hoffmann et al., 2013], loosely coupled (Definition 1) to the operating system and hardware.

To fulfill this strategic objective, it is necessary to meet the following *specific goals*:

1. Remove the many-core management tasks from the target platform OS, reducing the latter memory footprint and discarding dedicated OSs for management purposes;
2. Define the method to execute ODA tasks in userspace, making the MA modular (Definition 2) using a proper communication Application Programming Interface (API), system monitors, and actuation mechanisms;
3. Develop a mapping heuristic tailored to the MA;
4. Use the Quality of Service (QoS) management objective to evaluate the proof-of-concept MA with task migration;
5. Make the MA organization agnostic of the hardware, turning it possible to replace the processor, evidencing portability (Definition 3).

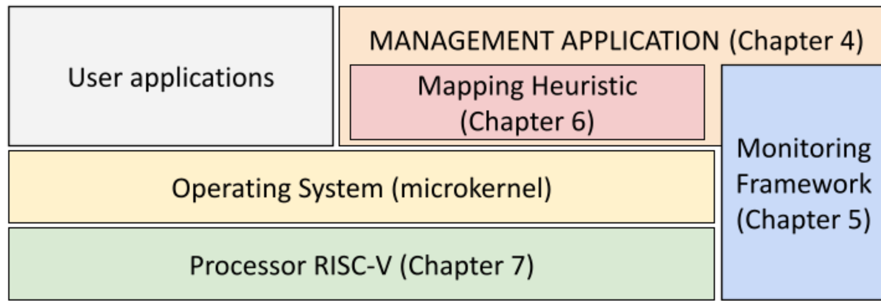


Figure 1.1: Document organization, highlighting the main work contributions.

1.4 Document Organization

This work is organized as follows. Chapter 2 describes basic concepts related to many-cores required in this work. Chapter 3 presents the state-of-the-art in many-core management organizations.

Chapter 4 to Chapter 7 present the original contributions of this work, according to Figure 1.1:

- Chapter 4 presents the MA framework [Dalzotto et al., 2021b] and a proof-of-concept implementation to evaluate a QoS management goal through task migration actuation.
- Chapter 5 presents the monitoring framework using a dedicated control Network-on-Chip (NoC) named BrNoC, to transmit management and monitoring messages, improving the MA performance.
- Chapter 6 details a task mapping heuristic to be used with MA [Dalzotto et al., 2021a]. Two reasons justify this proposal: (*i*) the change of the management organization in the target platform that removed its clustering, which was necessary for the former mapping heuristic, requiring a new one; (*ii*) the use of the QoS goal for the proof-of-concept implementation. This goal uses the task migration actuation, which needs a mapping heuristic.
- Chapter 7 presents the replacement of the original Memphis processor, Plasma, by the RISC-V processor, demonstrating the portability achieved by the MA approach.

Chapter 8 concludes this work and points-out directions for future work.

2. BASIC CONCEPTS

Many-cores are Systems-on-Chip (SoC) with simple processors replicated multiple times instead of few complex processors, granting benefits in power consumption and chip surface area [Asanovic et al., 2009]. Higher performance in many-cores is achieved with more explicit parallelism [Shalf et al., 2009].

This Chapter explains many-core concepts that guide this work. Section 2.1 introduces the goals that a many-core management must accomplish. Section 2.2 presents how a many-core management can be organized to provide those goals. Section 2.3 shows OS concepts from a many-core software point of view. Finally, Section 2.4 presents the Memphis many-core platform, used as the baseline architecture for this work.

2.1 Many-core Management Goals

To fully exploit the parallelism offered by many-cores it is necessary to execute several management tasks, such as: map tasks to cores aiming lower communication delay and energy; allows these tasks to migrate to avoid hot spots, solve mapping fragmentation, and meet deadlines; enable Dynamic Voltage and Frequency Scaling (DVFS) to keep the execution under power and temperature constraints or even target more reliability or lifetime.

According to Rahmani et al. [Rahmani et al., 2018b], developers usually configure a many-core environment to meet a few fixed objectives, while ignoring the dynamics of embedded systems goals. Figure 2.1 shows some conflicting goals that the Authors propose to switch at runtime. Switching goals at runtime based on the environment, the system changes, and the workload demand is called a *multi-objective* resource management.

One possible way to implement multi-objective resource management is to use the ODA control loop [Hoffmann et al., 2013]. This paradigm provides a modular way to organize the development of new management objectives through the division of roles into observation, decision, and actuation. It also allows better portability because it is only needed to update the tasks that interact with the system to port the management organization to another platform, eliminating redesign. The tasks that interact directly with the system are actuation and observation tasks. The ODA main advantage is the coordinated management of the components, which provides more efficiency and facilitates the multi-objective approach. The ODA control loop is divided into three phases:

- **Observe** - Extracts information about the system and task status, such as task period, deadline and execution time, temperature data, and communication latency.

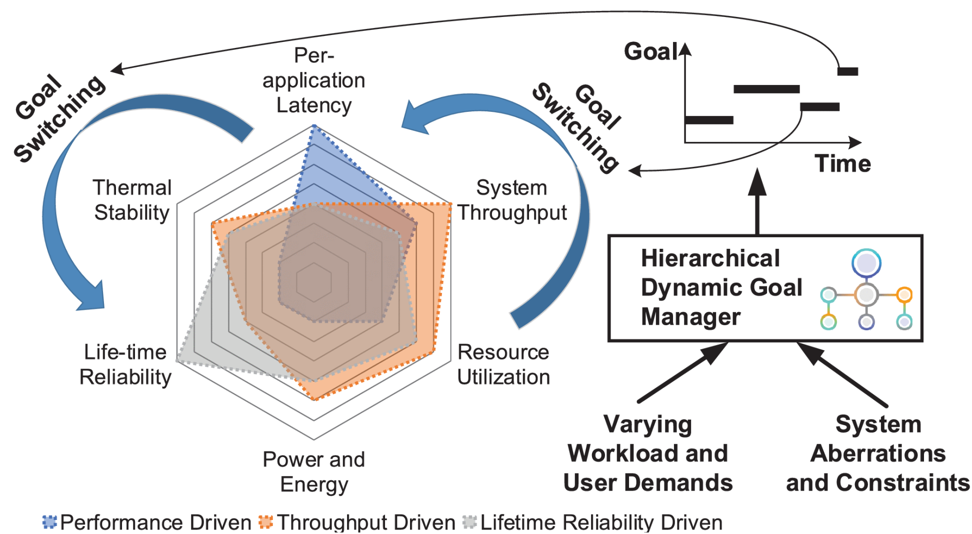


Figure 2.1: An adaptive resource management scheme. The workload and system variations trigger a goal switcher that changes the management objectives at runtime. The different goals can be conflicting, such as “System Throughput" and “Power and Energy". Source: [Rahmani et al., 2018b].

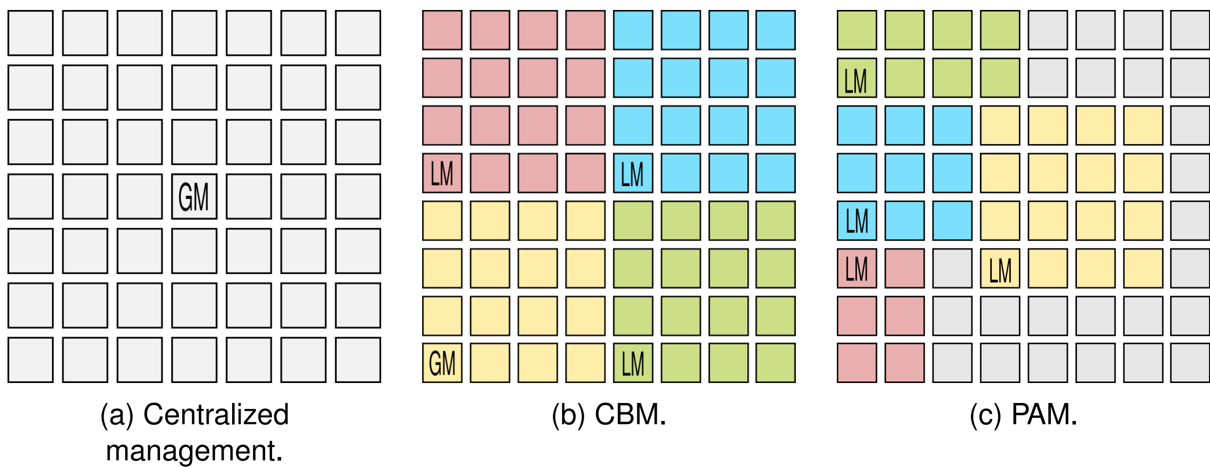
- **Decide** - Algorithms that decide the management action. Here enters the multi-objective approach, where the decision can be made from multiple different data from the observe phase.
- **Act** - Applies the decision made in the previous step, integrating the ODA loop to the hardware or the operating system through techniques such as DVFS, task migration, establishment of communication paths, and changes in priority of task scheduler.

2.2 Many-core Management Organization

The many-core management organization defines *where* the management is located within the many-core and *how* it is executed in a many-core. Figure 2.2 presents the three main organization classes: centralized management, Cluster-Based Management (CBM), and Per Application Management (PAM).

In centralized management, shown in Figure 2.2a, the designer allocates one Processing Element (PE) of the many-core, called Global Manager (GM), to be the controller of all management actions. Figure 2.2b shows a CBM organization, which besides using a GM to synchronize the whole many-core management, separates it into regions controlled by Local Managers (LM). PAM is another distributed way to manage a many-core, shown in Figure 2.2c, that dynamically assigns a manager for each running application.

The state-of-the art presented in Chapter 3 discusses works related to each organization, evaluating their pros and cons.



(a) Centralized management.

(b) CBM.

(c) PAM.

Figure 2.2: Management organizations present in the literature.

2.3 Kernel Designs

Tanenbaum and Bos [Tanenbaum and Bos, 2014] see an OS as a hardware extension, which implements device drivers and abstraction layers to support application programming. They also see the OS as a resource manager, multiplexing their use in time and space through task scheduling and memory allocation, among other techniques. Figure 2.3 shows that the OS runs its core software – called *kernel* – in a privileged Central Processing Unit (CPU) mode, the *kernel mode*, with access to all machine instructions and memory space. Simultaneously, applications run in *user mode*, having a subset of the CPU instructions available and only accessing the memory allocated to the application.

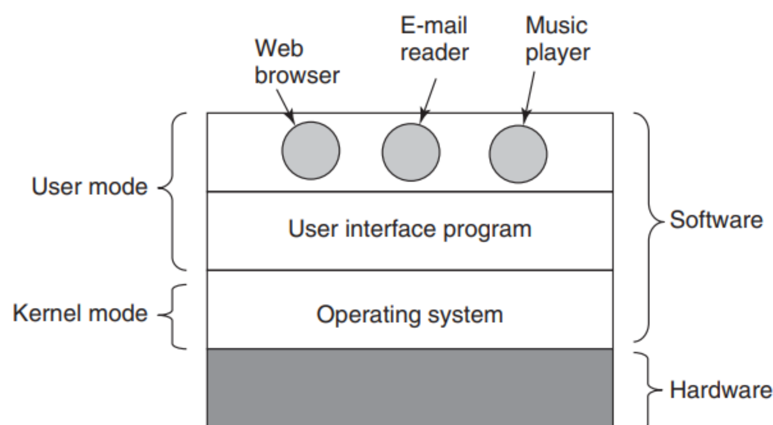


Figure 2.3: How a typical system is partitioned. Source: [Tanenbaum and Bos, 2014].

Kernel designs follow specific organization structures, but there are two that stand out: monolithic and microkernel. The monolithic design is the entire OS linked in a single binary. It can be very efficient when there are few functionalities and when the size of the OS remains moderate. In this case, there is a low overhead in calling functions from the kernel, and procedures can be shared between system services. Kernel functions available

to applications can be achieved by system calls, or syscalls. Syscalls are triggered by trap instructions that elevate the CPU privilege from user mode to kernel mode. Linked to a single binary, the entire OS is executed in kernel mode.

To Biggs et al. [Biggs et al., 2018], running the whole OS in kernel mode can be considered a flawed design because it increases the system trusted computing base, and due to the nature of all code containing bugs, it can lead to vulnerabilities. Tanenbaum and Bos [Tanenbaum and Bos, 2014] cite the same problem leading to full system crashes, reducing the overall reliability.

The microkernel approach achieves modularity, splitting the functionality of the OS into small programs. Only the microkernel itself runs in privileged kernel mode, while its modules run in user mode. It is essential not to confuse this concept with kernel modules in a monolithic context, where a module represents device drivers being attached to the kernel and running in kernel mode. The microkernel function is to execute only the minimum necessary for the system to work, like MINIX 3 [Tanenbaum and Bos, 2014], that only schedules processes, handles interrupts, and manages Inter-Process Communication (IPC).

Figure 2.4 shows a layered organization of privileges that microkernels can take advantage of, with each module running with a minimum of privilege level. It is possible for the drivers to run unprivileged, requiring a call to the microkernel to access Input and Output (I/O).

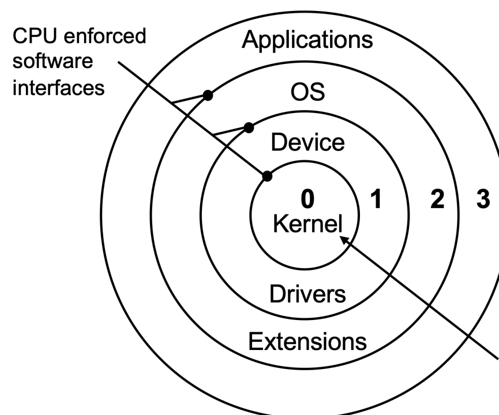


Figure 2.4: The Intel x86 Ring Architecture. Source: [Reid and Caelli, 2005].

Figure 2.5 shows a comparison between the partitioning of a microkernel and a monolithic approach, showing that instead of syscalls, microkernel uses IPC to communicate between services.

This work uses microkernel concepts to apply resource management, such as detaching management modules from the kernel and executing this modules in user mode, and using IPC through message-passing.

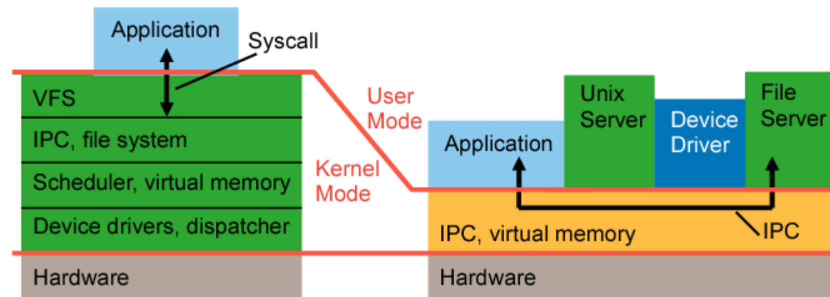


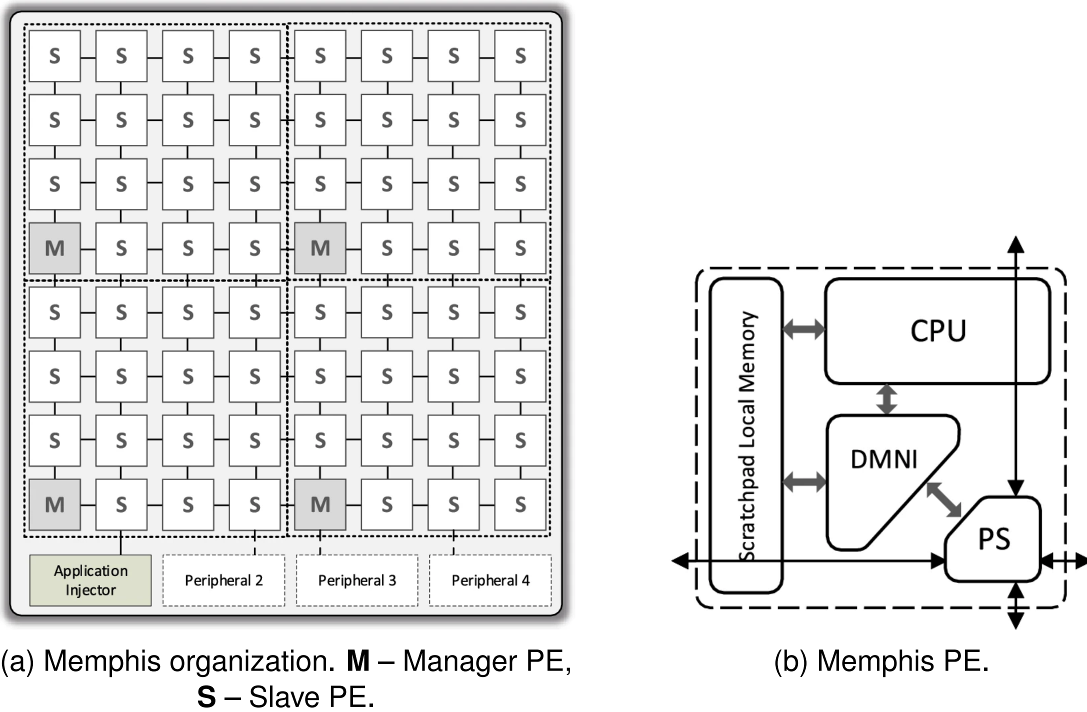
Figure 2.5: Monolithic (left) vs. microkernel (right) privileges. Source: [Biggs et al., 2018].

2.4 Many-core Modeling Platform for Heterogeneous SoCs (Memphis)

Memphis [Ruaro et al., 2019a] is a framework for automatic generation and validation of many-cores. It allows designing an NoC-based many-core surrounded by peripherals, with available debugging tools to simultaneously verify hardware and software. The Memphis architecture is an evolution of the Hermes Multiprocessor Systems (HeMPS) [Carara et al., 2009] Multiprocessor System on Chip (MPSoC), with support for external peripherals, and a new management kernel.

The Memphis model contains PEs interconnected by the Hermes 2D-mesh NoC [Moraes et al., 2004]. Hermes is a packet switching NoC, with XY routing and round-robin arbitration, input buffering, and credit-based control flow. Figure 2.6a shows the homogeneous region with PEs and peripherals connected to the many-core borders. Note that one of the Manager PEs is the GM, while the others are LMs. Figure 2.6b shows the components of each PE, including: (*i*) a Plasma **CPU** [Rhoads, 2001], extended with memory relocation support for multiprogramming. Plasma has an Instruction Set Architecture (ISA) similar to the MIPS I, but without unaligned load and store operations; (*ii*) a true dual-port scratchpad **memory** for instructions and data; (*iii*) a Direct Memory Network Interface (DMNI), integrating **Network Interface** (NI) and **Direct Memory Access** (DMA) modules; (*iv*) a Packet-Switching (PS) Hermes **router**.

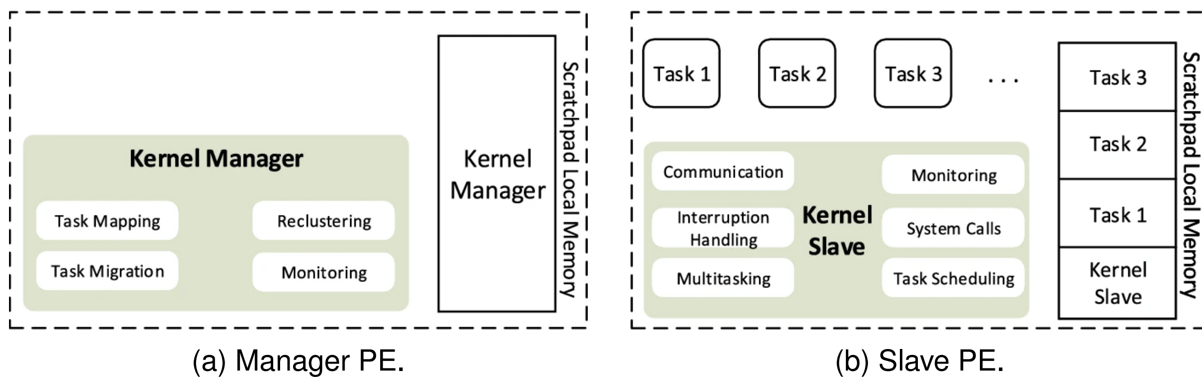
Memphis uses a CBM organization with reclustering support [Castilhos et al., 2013]. This approach uses a GM to control its clusters and to keep the synchronization of the whole system. Figure 2.7a shows the Kernel Manager, loaded into GM and LM PEs, which only runs management procedures and do not support user tasks. Figure 2.7b illustrates the Kernel Slave, loaded into all other PEs, which manages multitask scheduling with parameterizable pages, provides a message passing API, and monitors deadlines. Both kernels use monolithic designs, with all its functionalities, including management, linked to the same binary and running in the same privilege level. The primary Memphis management goals are providing QoS and meeting deadlines through task mapping and migration.



(a) Memphis organization. **M** – Manager PE, **S** – Slave PE.

(b) Memphis PE.

Figure 2.6: Memphis many-core overview. Adapted from: [Ruaro et al., 2019a].



(a) Manager PE.

(b) Slave PE.

Figure 2.7: Overview of Memphis kernels organization. Adapted from: [Ruaro et al., 2019a].

Each applications in Memphis is modeled as a Communication Task Graph (CTG). The CTG is a model to represent functional parallelism, where an application is composed of parts that are independent of each other and thus are divided into tasks [Rauber and Rünger, 2013]. A graph node represents each task in a CTG, and the graph edges represent the communication between these tasks.

A parallel application is often structured in a pattern that is effective for many different applications [Rauber and Rünger, 2013]. These patterns provide a specific coordination structure for the application. The three main patterns used in Memphis applications are:

- **Fork-join**: a task *forks* the workload, splitting the computation between *worker* tasks. Another task awaits for the *workers* termination to *join* the results.

- **Master-slave**: a *master* task controls and distributes the workload to its *slaves*, or *worker* tasks. The *master* is also often responsible for executing the main part of the program and joining the results.
- **Pipeline**: data is forwarded from task to task to perform different processing steps in sequence. Parallelism is achieved by partitioning the data into streams that flow through the pipeline stages.

Memphis tasks communicate through *Send* and *Receive* primitives provided by the kernel API. A peripheral called *Application Injector* (see Figure 2.6a) is responsible for transmitting tasks to be loaded into the many-core. Figure 2.8 shows how this peripheral injects an application following a set of well-defined phases:

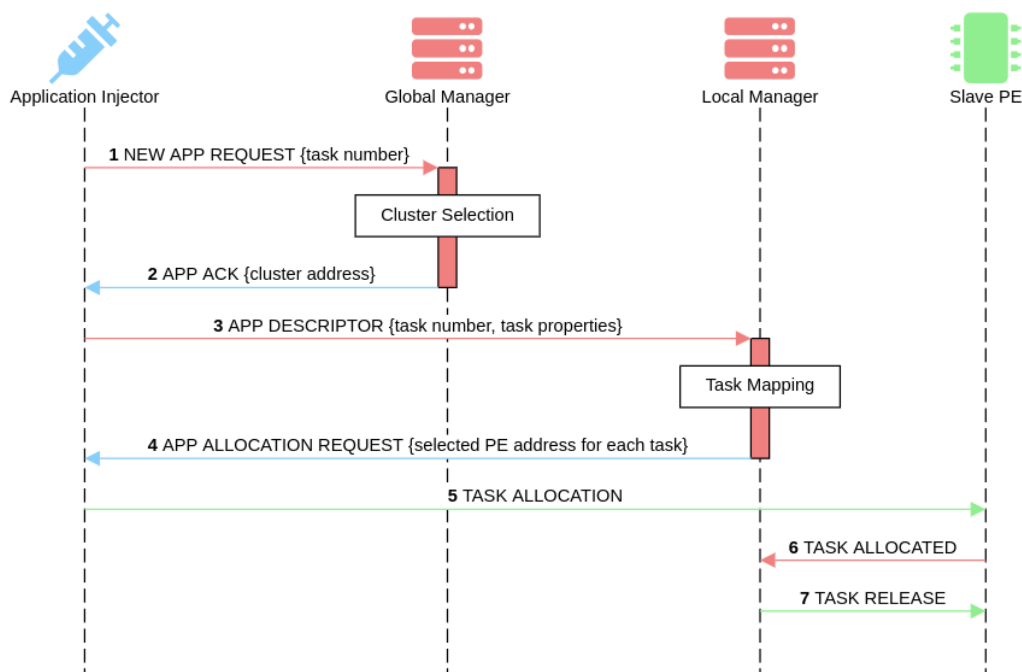


Figure 2.8: Sequence diagram of the Application Injector protocol.
Adapted from: [Ruaro et al., 2019a].

The protocol work as follows:

1. The Application Injector requests the GM a cluster for the application allocation;
2. The GM answers with the chosen LM address.
3. The Application Injector sends an application descriptor to the LM;
4. The LM runs the task mapping algorithm, sending back the selected PE for each task mapping;
5. The Application Injector sends the binary code of each task to each selected PE;
6. Slave kernels confirm the allocation to the LM;

7. When all tasks are allocated, the LM sends a synchronization message allowing the application to start.

The framework design flow is guided by files written in the YAML markup language. A *test case* is a file that describes the features of the platform, and a *scenario* is a file that lists the applications to evaluate. The configuration is interpreted by Python scripts that generate the hardware model and compile the kernel and applications. The *test case* supports parameters of page size, number of tasks per PE, number of PEs in the many-core, and the peripherals specification. The simulation of the platform can be achieved with Very-High-Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) or SystemC descriptions.

Memphis is an open-source framework with guides and video-tutorial available for download at https://www.inf.pucrs.br/hemps/memphis.html.

3. RELATED WORK

Figure 3.1 proposes a taxonomy to guide the many-core state-of-the-art evaluation. Four orthogononal criteria are adopted:

- **Management Organization:** defines how and where resource management is implemented, as explained in Section 2.2;
- **Architecture:** the managed system structure, including the communication infrastructure (NoC or bus), the memory organization, and the processor architecture;
- **Validation:** how the work is evaluated – emulation, simulation, or custom implementation in either Field-Programmable Gate Array (FPGA) or Application-Specific Integrated Circuit (ASIC);
- **Management Goals:** the controlled performance figures of the work, such as performance, energy, and QoS.

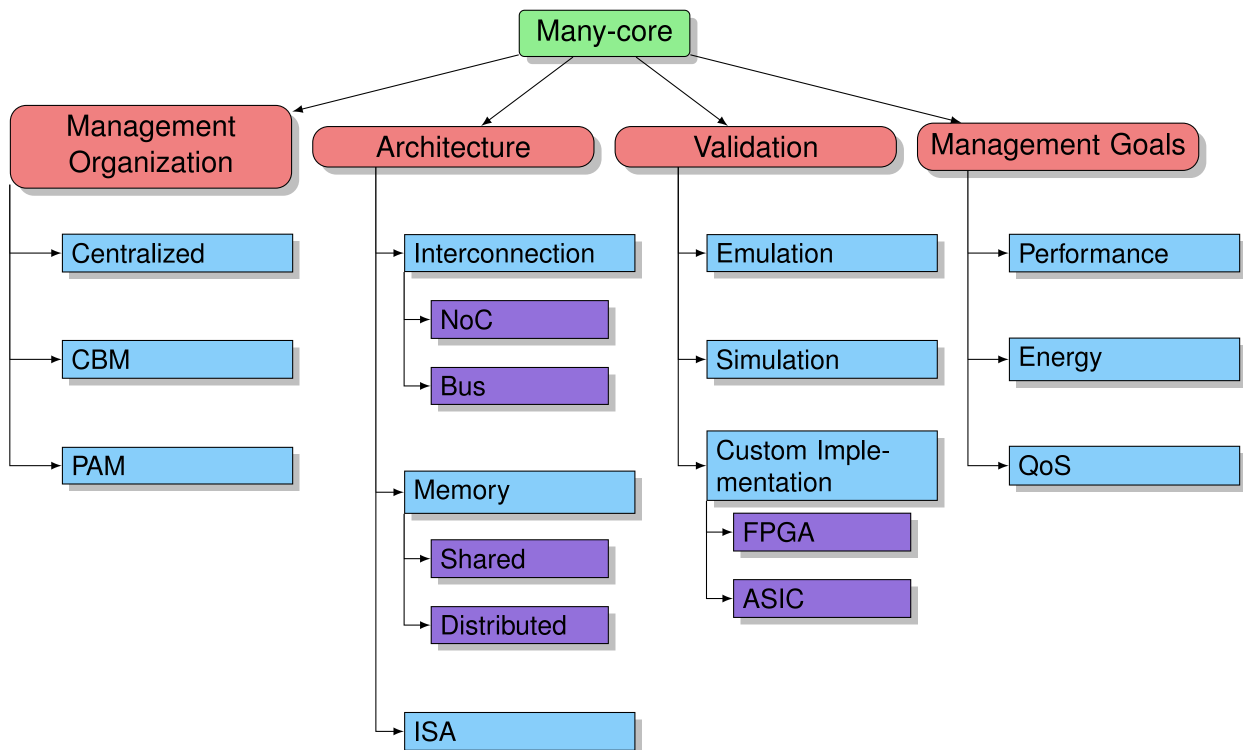


Figure 3.1: Many-core taxonomy for the related work.

Table 3.1 is an overview of the state-of-the-art in many-core literature. Each **row color** represents a management organization (first taxonomy criterion): red is centralized, green is CBM, blue is PAM, and violet is reserved to other approaches that are considered as "hybrid", not belonging to any other organization or belonging to more than one. The row in orange represents our management organization, the Management Application. The table **columns** follows the remaining taxonomy criteria:

- **First column:** the title of the work;
- **Second column:** the many-core architecture, highlighting its main features, such as number of cores, memory organization, and CPU type;
- **Third column:** the many-core validation method;
- **Fourth column:** the management goals the authors try to control. Cells in the fourth column marked with a **P** indicate that the work proposes a management paradigm.

Table 3.1: Related work on management organizations.

Work	Architecture	Validation Method	Goals/Paradigm	
ARTE: An Application-specific Run-Time managEment framework for multi-cores based on queuing models [Mariani et al., 2013]	Host processor and 16 MIPS-like processors with cache-coherent shared memory bus	Super ESCalar Simulator (SESC) cycle-accurate simulation tool [Renau et al., 2005]	Response time reduction	
Defragmentation of Tasks in Many-Core Architecture [Ng et al., 2016]	2D mesh NoC, up to 14x14 cores	Extended Noxim (SystemC simulator) [Catania et al., 2016]	Total execution time and energy reduction	
Dynamic Allocation/Reallocation of Dark Cores in Many-Core Systems for Improved System Performance [Huang et al., 2020]	Homogeneous 2D mesh NoC, up to 12x12 Alpha cores	Event-driven C++ network simulator, and cycle-accurate many-core simulator [Wang and Mak, 2013]	Reduce the impact of dark cores on communication latency and fragmentation	
Performance-Aware Resource Management of Multi-Threaded Applications on Many-Core Systems [Olsen and Anagnostopoulos, 2017]	16-core Nehalem	Sniper many-core simulator [Carlson et al., 2011]	Maximize performance keeping the number of hotspots low	
Reliability-Aware Runtime Power Management for Many-Core Systems in the Dark Silicon Era [Rahmani et al., 2017]	Intel Single-chip Cloud Computer (SCC)-like	SystemC system-level simulator based on Noxim [Rahmani et al., 2015]	Optimize performance while prolonging the lifetime of the system by avoiding stress and thermal hotspots	
Thermal-Cycling-aware Dynamic Reliability Management in Many-Core System-on-Chip [Haghbayan et al., 2020]	Scalable Processor Architecture (SPARC) processors with NoC-based shared memory	Noculator many-core simulator [Ausavarungnirun et al., 2014]	Reduce thermal cycling effects	
ADAM: Run-time agent-based distributed application mapping for on-chip communication [Faruque et al., 2008]	Up to 32x32 PEs mesh NoC	FPGA prototype	Scheme for a run-time application mapping in a distributed manner	**P**
Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes [Castilhos et al., 2013]	Up to 12x12 MIPS-like PEs in a 2D mesh NoC (HeMPS [Carara et al., 2009])	Register-Transfer Level (RTL) cycle-accurate SystemC simulation	Distributed resource management with dynamic cluster sizes (reclustering)	**P**
DRACON: A Dedicated Hardware Infrastructure for Scalable Run-Time Management on Many-Core Systems [Gregorek et al., 2019]	Intel SCC-like baseline with dedicated structures for hardware managers	Transaction-level Agamid simulation framework [Gregorek and Garcia-Ortiz, 2018]	Elimination of user interference in management and mitigation of the management overhead	**P**

Continued on next page

Table 3.1 – continued from previous page

Work	Architecture	Modeling	Goals/Paradigm	
Hierarchical adaptive Multi-objective resource management for many-core systems [Martins et al., 2019]	12x12 MIPS-like PEs in a 2D mesh NoC (HeMPS [Carara et al., 2009])	RTL cycle-accurate SystemC simulation	Dynamically adapt the running applications according to peaks and valleys of workload inherent to real systems while guaranteeing the power cap	
Hierarchical dynamic thermal management method for high-performance many-core microprocessors [Wang et al., 2016]	Up to 25x25 Alpha processors	Simulation in MATLAB	Reduce performance degradation and improve the thermal reliability	
Self-adaptive QoS management of computation and communication resources in many-core SoCs [Ruaro et al., 2019b]	64 MIPS-like processors connected via NoC (Memphis [Ruaro et al., 2019a])	RTL simulations (SystemC and VHDL)	Dynamic profiling and self-adaptive QoS management for soft real-time applications	
Application-Arrival Rate Aware Distributed Run-Time Resource Management for Many-Core Computing Platforms [Tsoutsouras et al., 2018]	Intel SCC: 24 tiles of 2 cores each connected via NoC [Howard et al., 2010]	Physical implementation	Regulate application admission without neglecting its distributed nature	
A Scalable Strategy for Run-time Resource Management on NoC based Manycore Systems [Liao and Srikanthan, 2011]	5x6 PowerPC NoC	NoCsim-UNISIM Cycle-accurate simulator [Liao et al., 2011]	Scalable hierarchical strategy for runtime resource management on embedded manycore NoCs	**P**
DistRM: Distributed resource management for on-chip many-core systems [Kobbe et al., 2011]	Up to 32x32 cores NoC	System-level simulation	Less management effort through agent-based distributed scheme	**P**
Distributed run-time resource management for malleable applications on many-core platforms [Anagnostopoulos et al., 2013]	Intel SCC: 24 tiles of 2 cores each connected via NoC [Howard et al., 2010]	Physical implementation	Present a distributed management framework for malleable applications	**P**
System Software for Resource Arbitration on Future Many-* Architectures [Schmaus et al., 2020]	Shared memory within tiles and message passing through NoC	FPGA prototype	Allow micro-paralellism, reduce OS interference and implement strong monitoring	
Self-Aware Cyber-Physical Systems-on-Chip [Dutt et al., 2015]	Communication NoC, Sensor NoC and Introspective Sentient Units	N/A	Define a model of self-awareness and cyber-physical systems for autonomous adaptive management	
SPECTR: Formal Supervisory Control and Coordination for Many-core Systems Resource Management [Rahmani et al., 2018a]	Exynos big.LITTLE ARM octa-core processor	Physical implementation	Efficiently manage complex system with multiple goals	
This work	Memphis-V: a RISC-V many-core based on Memphis (see Chapter 4, Chapter 5, and Chapter 7)	SystemC simulation	Propose a modular management framework, loosely coupled to the OS, and portable to other architectures	**P**

The next Sections discuss the main related work. Section 3.1 discusses the use of centralized management organization. Section 3.2 describes some CBM works. Section 3.3 lists different PAM organizations. Section 3.4 presents two works that are classified as "hy-

brid". Finally, Section 3.5 shows the state-of-the-art final remarks and how the present work fulfills the observed gaps in the reviewed proposals.

3.1 Centralized Approaches

Centralized is the most straightforward organization architecture for a many-core. This Section only discusses two works from Table 3.1 which present a different type of centralized management, where the global resource manager is located outside of the many-core fabric.

Rahmani et al. [Rahmani et al., 2017] connect a many-core similar to the Intel Single-Chip Cloud Computer (SCC) to a host machine controlling runtime application mapping and Dynamic Power Management (DPM). The many-core tiles are responsible for scheduling tasks, managing communications, and monitoring sensors. Their work focus on a multi-objective DPM that also considers performance fulfillment, reliability, and system lifetime. Authors claim to have achieved enhanced throughput, Thermal Design Power (TDP) or Thermal Safe Power (TSP) constraints, and overall lifetime compared to the state-of-the-art management policies.

Figure 3.2 shows the Application-specific Run-Time managEment (ARTE) [Mariani et al., 2013], a framework that uses a general-purpose host processor called "fabric controller" connected to a many-core fabric composed by 16 MIPS-like processors. It uses queuing models to achieve less response time and meet QoS under power constraints. Results showed up to 68.5% more performance than traditional approaches and similar behavior with reduced overhead compared to Reinforcement Learning (RL) techniques.

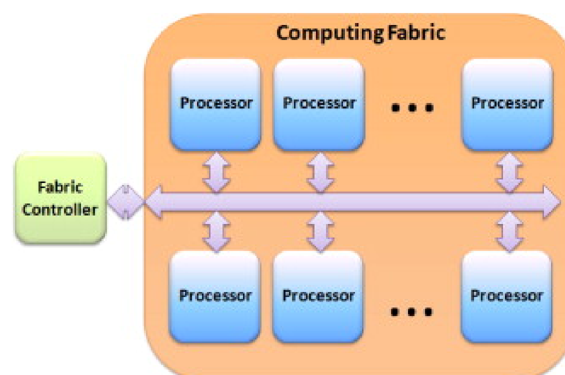


Figure 3.2: ARTE many-core structure. Source: [Mariani et al., 2013].

According to Castilhos et al. [Castilhos et al., 2013], centralized management can be overloaded when answering requests from numerous PEs, also generating a considerable amount of traffic around it. Therefore, it is only suited for lower core counts due to its weak scalability.

3.2 CBM Approaches

This Section discusses the three works of Table 3.1 which propose a CBM organization. The remaining CBM works present in the Table only use a previous defined organization to evaluate management goals.

Agent-based Distributed Application Mapping (ADAM) [Faruque et al., 2008] was the first hierarchical CBM that follows a centralized manager called a "global agent" that can borrow resources from other clusters in a "reclustering" technique. This work focus is a distributed runtime application mapping, which showed more than seven times less computational effort than a simple centralized mapper. The PEs monitoring also showed more than ten times lower traffic compared to a centralized organization. HeMPS [Carara et al., 2009], a platform that later became Memphis, uses these same CBM concepts.

Figure 3.3 shows DRACON [Gregorek et al., 2019]: a dedicated infrastructure for RunTime Management (RTM) using CBM organization. It has an RTM slave hardware connected to each core that operates as an OS serving system calls and scheduling tasks and communicates with each other through dedicated management interconnect. Each cluster has an RTM master hardware, which also communicates with other masters through another dedicated interconnect. DRACON achieves performance improvements from 6.12% up to 15.21% depending on the workload compared to software-based management and increases the chip area by 3.01%.

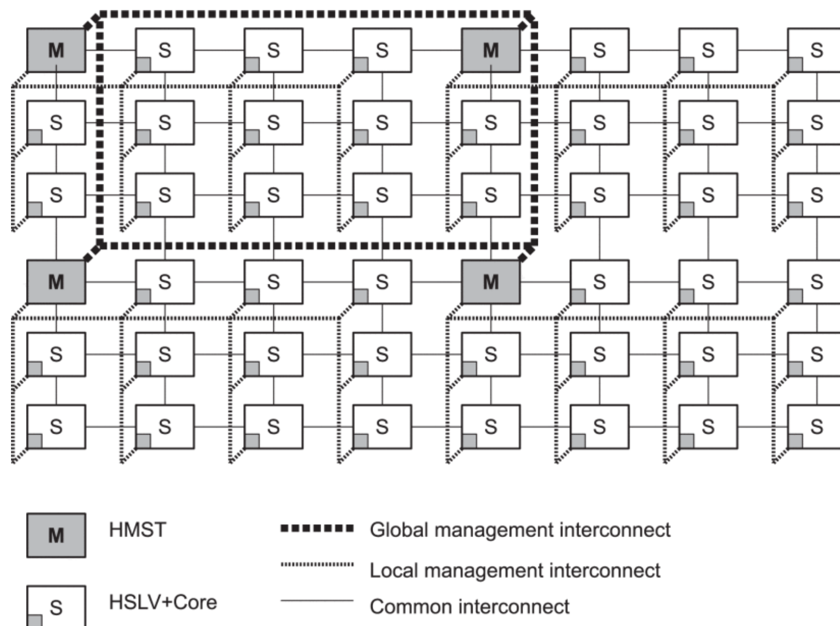


Figure 3.3: RTM components in DRACON. **HMST** – Hardware Master, **HLSV** – Hardware Slave. Source: [Gregorek et al., 2019].

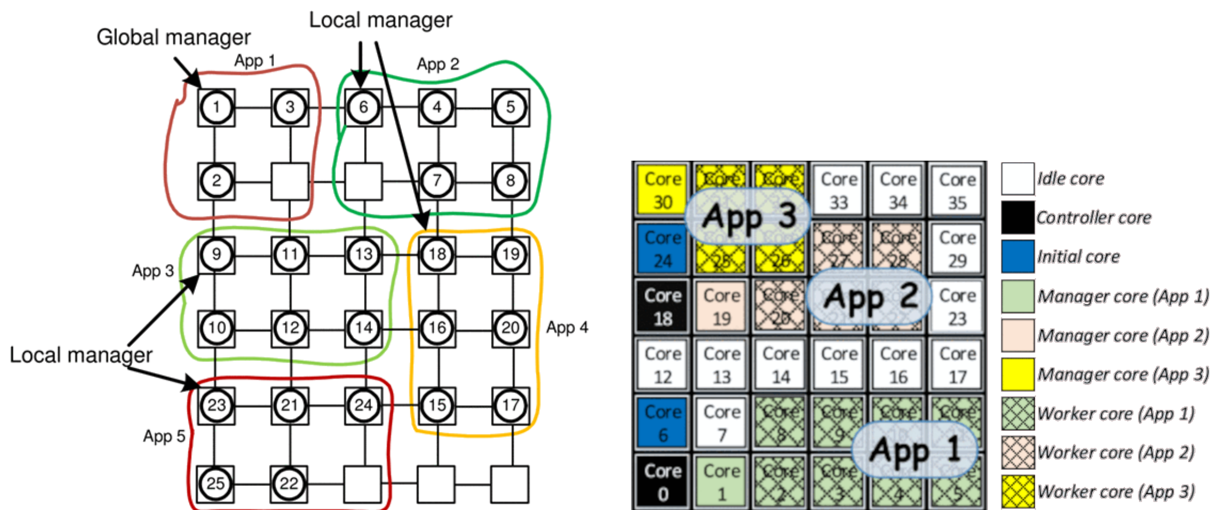
The CBM main advantage is that it is easily scalable to a large number of cores. However, the disadvantages are: (*i*) the resulting overhead of lost PEs used only for man-

agement purposes, mainly occurring on huge many-cores with smaller cluster sizes; and (*ii*) the cluster size is essentially static, with its size and location being decided at design time, impairing the system dynamics.

3.3 PAM Approaches

This Section discusses the three works of Table 3.1 which propose a PAM organization.

Figure 3.4a shows the PAM approach proposed by Liao and Srikanthan [Liao and Srikanthan, 2011] that uses a global manager to create rectangular sub-meshes and assigns one core of this area to be a local manager that will control the application hierarchically. They justify the internal fragmentation of the sub-meshes as acceptable due to the utilization wall. This PAM architecture achieved lower resource allocation times and up to 63% reduced management communication energy compared to a centralized approach.



(a) PAM with global manager. Source: [Liao and Srikanthan, 2011].

(b) PAM with three levels of hierarchy. Adapted from: [Anagnostopoulos et al., 2013; Tsoutsouras et al., 2018].

Figure 3.4: Different PAM organizations.

DistRM [Kobbe et al., 2011] uses the same PAM organization but is fully decentralized without any global synchronization by assigning each manager called "agent" randomly at application arrival. These managers are based on the concept of multi-agent systems, distributing the control among agents that have local information about the system and act independently of each other aiming to solve difficult or impossible problems for a central management. It achieves 84% of the optimal centralized scheme mapping quality that considers free task migrations while reducing the computational complexity to less than 1% of the centralized in a 1024 core system with 12.75% of the required network bandwidth.

Figure 3.4b shows the proposal of Anagnostopoulos et al. [Anagnostopoulos et al., 2013], which uses PAM in a more deep hierarchical way. They first separate the many-core into clusters based on "controller cores", using "initial cores" for temporary resource management at application arrival, and then finally creating "manager cores" for each application management and resource exchange. Compared to DistRM, this work shows reduced communication overhead with 70% fewer messages and 64% less message size while gaining up to 20% speed-up.

PAM addresses the CBM problem of lacking runtime mutability by creating and killing managers at runtime. However, two main trade-offs exist: (*i*) a considerable overhead for applications with a small number of tasks; and (*ii*) focusing mainly on applications QoS instead of achieving the whole system goals. DistRM and Anagnostopoulos et al. address the first issue by using malleable applications, which can adapt their degree of parallelism using more cores when possible, thus preventing a low number of tasks that would result in high management overhead.

3.4 Hybrid Approaches

SPECTR [Rahmani et al., 2018a] aims at system autonomy to provide adaptive management for multi-objective goals. The authors use Supervisory Control Theory (SCT) to achieve this coordinated autonomy, which breaks the management problems into small-scale sub-problems solvable by control loops. The platform runs Linux and invokes the controllers every 50 ms and the supervisor every 100 ms. They benchmark the proposed solution to show that it could manage conflicting goals adapting to the system changes and achieving efficient QoS within the power budget. At the same time, state-of-the-art controllers could not do so. This particular work is scalable by breaking the problems into simple controllers coordinated by a high-level supervisor but do not evaluate a distributed memory environment with a large number of cores.

Dutt et al. [Dutt et al., 2015] also aim at autonomy by the awareness of the system state (self-awareness) and its environment (context-awareness). They call "Cyber-Physical systems" those combining a rich set of sensors and actuators using a self-aware computing-communication-control paradigm with adaptive and reflexive middleware to control the effects of computation on the chip and the interacting environment. Their proposed platform contains sensing and actuation, self-aware adaptations, cross-layer interactions and interventions, and predictive modeling and learning. The work focus on defining a model for self-awareness and cyber-physical systems without enforcing a specific implementation.

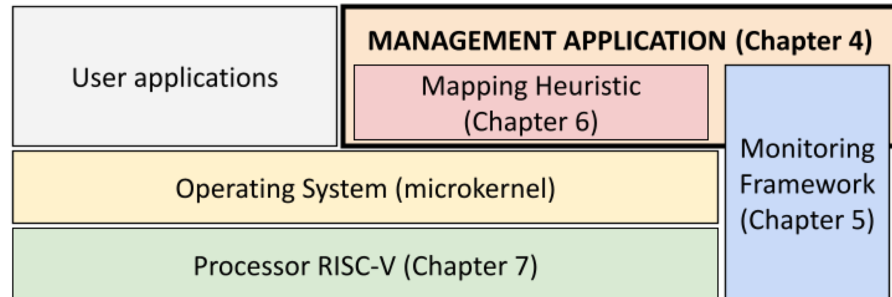
3.5 Final Remarks

Most of the evaluated works in this research [Mariani et al., 2013; Ng et al., 2016; Huang et al., 2020; Olsen and Anagnostopoulos, 2017; Rahmani et al., 2017; Haghbayan et al., 2020; Martins et al., 2019; Wang et al., 2016; Ruaro et al., 2019b; Schmaus et al., 2020; Dutt et al., 2015; Rahmani et al., 2018a; Tsoutsouras et al., 2018] make use of an organization to evaluate an algorithm, hardware, or framework. This fact resulted in six centralized resource management works. Adopting this approach is justifiable because of its implementation simplicity that uses a PE of the many-core to execute the management procedures. But these proposals present poor scalability to a small number of cores.

The three studied organizations (centralized, CBM, and PAM) rely on dedicated cores for management, which require specific support from the OS kernel, imposing challenges to reuse the management procedures in other systems and making it difficult to add new goals. CBM organization, some reviewed centralized organizations [Mariani et al., 2013; Rahmani et al., 2017], and DRACON [Gregorek et al., 2019] are tightly coupled to the hardware platform, making it impractical to use an already verified management organization on different platforms. Despite being scalable, PAM and CBM result in an overhead of reserved resources for management, reducing the maximum allocated tasks, implying in a potential loss of parallelism.

Chapter 4 proposes a way to address these limitations, suggesting a new management organization that provides a modular method to add new management goals, loosely coupled to the OS, and independent of the hardware organization.

4. MANAGEMENT APPLICATION



This Chapter presents the original first contribution of this work: the MA paradigm to implement a many-core organization supporting multi-objective goals with the ODA loop. Ruaro et al. [Ruaro et al., 2021] originally proposed the MA paradigm, but without an actual implementation to demonstrate its benefits. Section 4.1 details the MA paradigm, and Section 4.2 the development of a MA proof-of-concept using application mapping and task migration. Section 4.3 shows the results obtained by the MA. Results presented in this Chapter were published in [Dalzotto et al., 2021b].

4.1 The MA Paradigm

Figure 4.1 presents an example of a many-core managed by the MA approach. Each gray tile in the Figure is a PE either free or occupied by user tasks. The blue, red, and green tiles represent Observation, Decision, and Actuation tasks, respectively. Observation tasks gather and abstract raw monitoring data. Decision tasks choose an actuation based on the observed scenario using heuristics. Actuation tasks manage protocols and underlying hardware using APIs to apply decisions such as controlling DVFS or migrating a task.

The MA tasks can be mapped at different system positions, and their number can also be defined at runtime according to the workload requirements. Note that the mapping of the MA tasks presented in Figure 4.1 is just an example. Tasks can be mapped close to each other, they can be allocated close to strategic regions of the chip where the monitoring load is higher, or allocated in cores with more computing power, specially for decision tasks that need to run heuristics at runtime.

The main advantages of the MA paradigm that motivated this work are [Ruaro et al., 2021]:

- No need for dedicated cores for management execution as in centralized management, CBM, and PAM. **Management tasks can share processors with user tasks**.

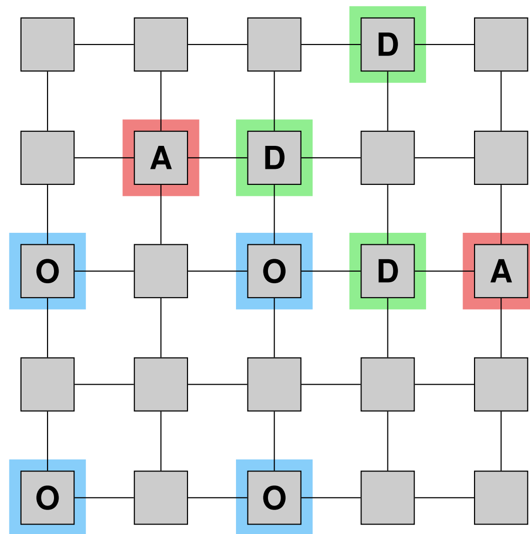


Figure 4.1: MA organization. **O** – Observation, **D** – Decision, **A** – Actuation.

- **MA tasks** are not bound to specific locations as in other paradigms and **can be migrated**. This possibility brings **additional reliability** since a task migration can move a management task from a core that is faulty or violating thermal constraints.
- OS with a **smaller memory footprint** and **easier to maintain** since it is not overloaded or modified with the insertion of new resource management modules. This assumption follows the microkernel concepts presented in Section 2.3, as the management tasks also run in user mode.

Identified limitations of the MA are [Ruaro et al., 2021]:

- MA tasks have the maximum scheduling priority over other user tasks and, consequently, **cannot share a CPU with Real-Time (RT) tasks**. However, MA tasks can be reactive, staying on a waiting state until an external event triggers its execution, reducing the overhead on cores shared with user tasks;
- The system designer needs to carefully choose when two or more MA tasks can share the same CPU, since **conflicting tasks can reduce the overall management performance**.

The hardware is independent to this proposal, i.e., the Management Application is hardware-agnostic. Besides being hardware-agnostic, the Management Application should also be OS-agnostic because the management is loosely coupled to the OS. Despite being OS-agnostic, the OS must still provides the following services to support MA:

- **Low-Level Monitors (LLM)**: periodically pulls raw data from hardware and redirects to Observation tasks without executing complex computation. LLM examples include task execution profiling (whether it is computation- or communication-intensive), RT constraints, power, thermal, communication latency, and core utilization.

- **Actuation Enforcer (AE)**: implements drivers and provides APIs to physically apply the requests from Actuation tasks. Examples are the DVFS driver and the task migration API, which require privileged memory access.
- **Communication API**: the OS must provide a secure communication method for MA tasks, ensuring that user tasks are not allowed to tamper with the system.

Figure 4.2 depicts the MA framework model. In summary, the LLM running at the OS of *each* core generates messages periodically. Observation tasks related to task performance, system budgets, and user commands (to force actuations directly), receive and abstract the monitored data to achieve the awareness about the system status. Observation tasks know the system *goals* and can convert raw **Monitoring data** into **Objectives**. Objectives are sent to the Decision task that converts them into **Actions** by using algorithms that detect when and what resource needs adaptation. If necessary, the Decision task triggers an Actuation task, which implements the protocols to dynamically change the resources by interacting with the AE at the OS level.

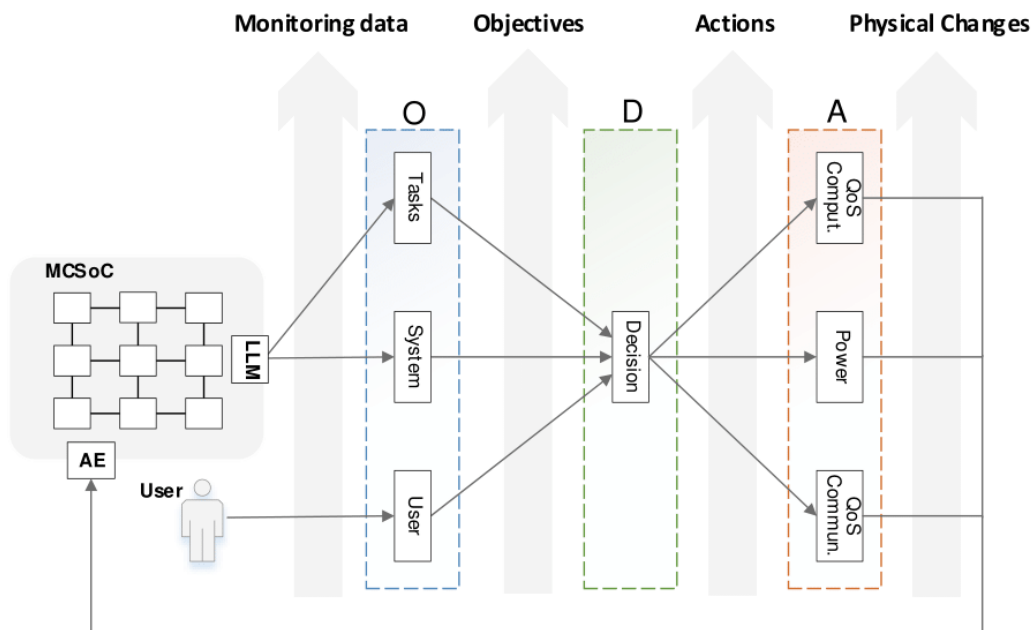


Figure 4.2: ODA Model used by MA paradigm. There is one LLM for each core. The AE is implemented by the OS running in each core. Arrows represent the communication between the entities. **MCSoC** – Many-core SoC. Source: [Ruaro et al., 2021].

Note that a goal is a high-level loosely defined plan of the system [Rahmani et al., 2018b], such as a demand for more bandwidth or less power. Objectives are the concrete action to meet such demand, such as establishing a dedicated network path or reducing the system frequency and voltage. The conversion of goals into an objective is not straightforward since different goals can result in one single objective [Shamsa et al., 2019]. Therefore, in a multi-objective management, the conversion tries to reach a balanced decision among those goals to apply conflicting objectives.

4.2 Proof-of-concept Implementation

The proof-of-concept is achieved by modifying the Memphis platform described in Section 2.4 to use the MA paradigm. The proposal focuses on providing an MA-based task mapping with a complete set of ODA tasks of the MA for migrating RT tasks at runtime when deadline violations are detected by the MA system. Section 4.2.1 details the modifications made to the Memphis kernel to support MA and improvements added to the platform. Section 4.2.2 details the new message-passing API implemented to support MA communication. In Section 4.2.3, the new application injection protocol is described alongside the new MA Injector. Finally, Section 4.2.4 describes the set of the MA ODA tasks used to provide task migration to RT tasks that violate deadlines.

4.2.1 Kernel modifications and platform improvements

The Memphis platform originally uses CBM, with different kernels in management cores and cores for user tasks. The first step to support MA is to remove the dedicated management kernel, creating a many-core running the same kernel in all cores. The many-core kernel is refactored, leaving only essential functions, such as task scheduling, message passing API, and task management for allocation and migration. The following data structures changed:

- **Task Location**: contains the location of tasks of the same application for all tasks running in the PE. This structure is changed, so that each task in the PE stores the location of the other tasks of its application locally, enhancing security and easing task migration. Security is enhanced because there is no more a global structure with all tasks identifiers (ID), avoiding access to a forbidden ID by a malicious task. Task migration is simplified since this structure contains only its own task data and can be easily transferred to the PE that will receive the task.
- **Message Request Repository**: this structure keeps the incoming message requests from consumer tasks destined to all producer tasks running in the PE. This message request is used by the handshake of the platform message-passing protocol. It changed to store the requests per task, also easing migration and enhancing security.
- **Message Pipe**: is a buffer with a parameterizable number of entries that stores the pending output messages of all tasks running in the PE. This buffer changed, so each task in the PE has its own buffer supporting a single message, adding the possibility to migrate tasks with a pending output message, and reducing the kernel size.

- **Migrated Tasks**: is a circular buffer added to store information about the tasks that migrated from the current PE and where they are migrated to. This buffer has a limited size since it serves the consumers of migrated task messages when they request a message to the PE where the migrated task was first mapped. Producer tasks are updated when the migrated task issues a message request. Additionally, the circular buffer size should be enough to avoid overwriting during the application lifespan.

Additionally to the kernel refactoring, the Memphis platform build system is rewritten to unify platform and scenario generation, application compilation, simulation, and debugging in a single executable while also adding the steps to build the MA.

Support for versions up to 11 of the GNU C Compiler (GCC) for the MIPS architecture is added to replace the previous version 4 limitation. This is made possible by modifying the linker configuration and updating the inlining declarations. This update benefits a possible support for different ISAs, such as RISC-V that has GCC support starting in version 7, standardizing compiler versions for all targets.

The kernel low-level functions, such as context switching, interrupt handling, and system bootloader are refactored to be detached from the high-level kernel code written in C, also aiming to facilitate the support for different ISAs. This detachment is done by implementing these functions as a Hardware Abstraction Layer (HAL) so that another ISA HAL can be swapped into the platform.

An initial step to standardize Memphis software libraries is developed with the adequacy of *stdlib* and *string* functions arguments and return values, and with the inclusion of the *stdio* library, providing a total of 12 standard C functions in the platform. The Memphis APIs library is also refactored with a new set of helper functions for ODA tasks.

4.2.2 Communication API

The management tasks of the MA can be reactive. The triggers for these reactions can come from numerous sources. However, the Memphis message API only supports receiving messages from a known producer due to the platform applications being modeled as a CTG. Thus, there is a need to provide a new communication API to MA.

Memphis provides two classes of communication messages: *request* and *delivery*. A producer task generates a *delivery* only when it has received a *request*, indicating that the consumer has allocated buffers and is ready to receive the message. The problem is that each *request* is emitted for a single producer defined at design time. The consumer is blocked until the message is delivered, thus preventing it from receiving messages from other sources than the designed.

Figure 4.3 shows the new messaging API diagram for management purposes using a new initial handshake step. When a producer wants to send a message, a *data available* packet is generated containing the producer task identifier (ID) and location. This packet is sent to the consumer to be stored in a structure inside the Task Control Block (TCB). When the consumer calls the receive message function, it will check the First In, First Out (FIFO) data available structure for the producers trying to send messages. The consumer then sends a *message request* to the producer, knowing its task ID and location from the *data available* message. Finally, the producer kernel dispatches the stored message inside a *message delivery* packet.

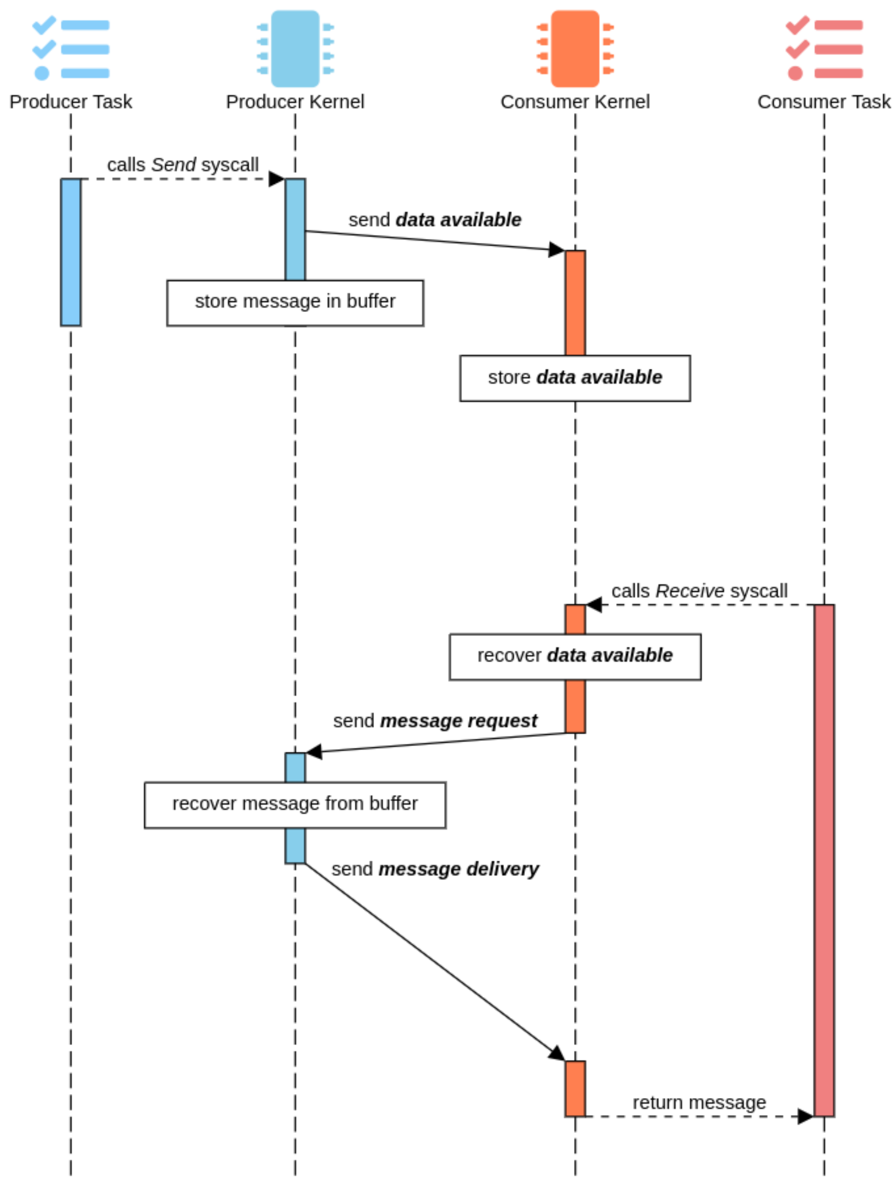


Figure 4.3: Sequence diagram of the new message-passing API for management communication.

This mechanism is similar to the `MPI_ANY_SOURCE` directive of the Message Passing Interface (MPI) used in task-to-task communication. The main difference of the new protocol from the `MPI_ANY_SOURCE` is that it also supports sending and receiving messages both

ways from kernel-to-task, peripheral-to-task, and peripheral-to-kernel. This communication results in a message-passing IPC. Figure 4.4 depicts the 32-bit flit used for the communication addressing. The lower bits still represent the X and Y addresses of the target used in Memphis. A new bit (K) indicating kernel (1) or task (0) destination is added. The upper bits indicating a forced exit port (E) at the address destination for peripherals has priority over the K bit to keep compatibility with the former messaging protocol.

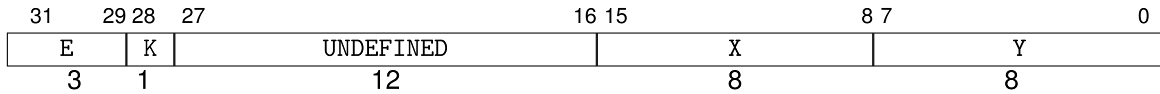


Figure 4.4: Format of the communication address. **K** – Kernel message, **E** – Port to force exit at the NoC borders, enabling the communication with peripherals.

4.2.3 Task injection

The Application Injector of Memphis is adapted in this work to address the new communication API and the removal of clustering. Figure 4.5 shows the diagram of the new Application Injector protocol. The first injection step occurs when the peripheral detects a new application and reaches its injection time configured at design time. The Injector sends a message to the MA Mapper Task containing a description of the application tasks to be used by the mapping algorithm. After a successful mapping, the Mapper Task answers the peripheral with an array containing the mapped task IDs with their mapped locations. Using the same communication API, the Application Injector sends the tasks binaries one by one to each mapped PE. The target PEs inform the task mapper MA that the task is allocated. When all tasks are allocated, the task mapper is responsible for sending a *task release* message containing all the application tasks locations back to each allocated PE kernel.

The Application Injector can be seen as an external communication interface receiving applications to run in the platform. For security reasons, it is necessary to separate the deployment of management tasks from user tasks. Thus, a second peripheral is added to the Memphis platform, called *MA Injector*. This new injector behavior is similar to the Application Injector, using the same protocol shown in Figure 4.5.

The MA Injector main functional difference from the Application Injector is that it injects the MA Mapper Task first, without negotiation, sending it to a PE defined at design time. The MA Injector only sends the remaining MA tasks after that, but this time negotiating with the Mapper Task. At system startup, all external interfaces of the many-core are disabled, except the MA Injector interface. Thus, this new peripheral can be seen as a flash memory containing a trusted boot code. Once all MA tasks are loaded in the system, the mapper task releases the external interfaces, e.g. the interface used by the Application Injector.

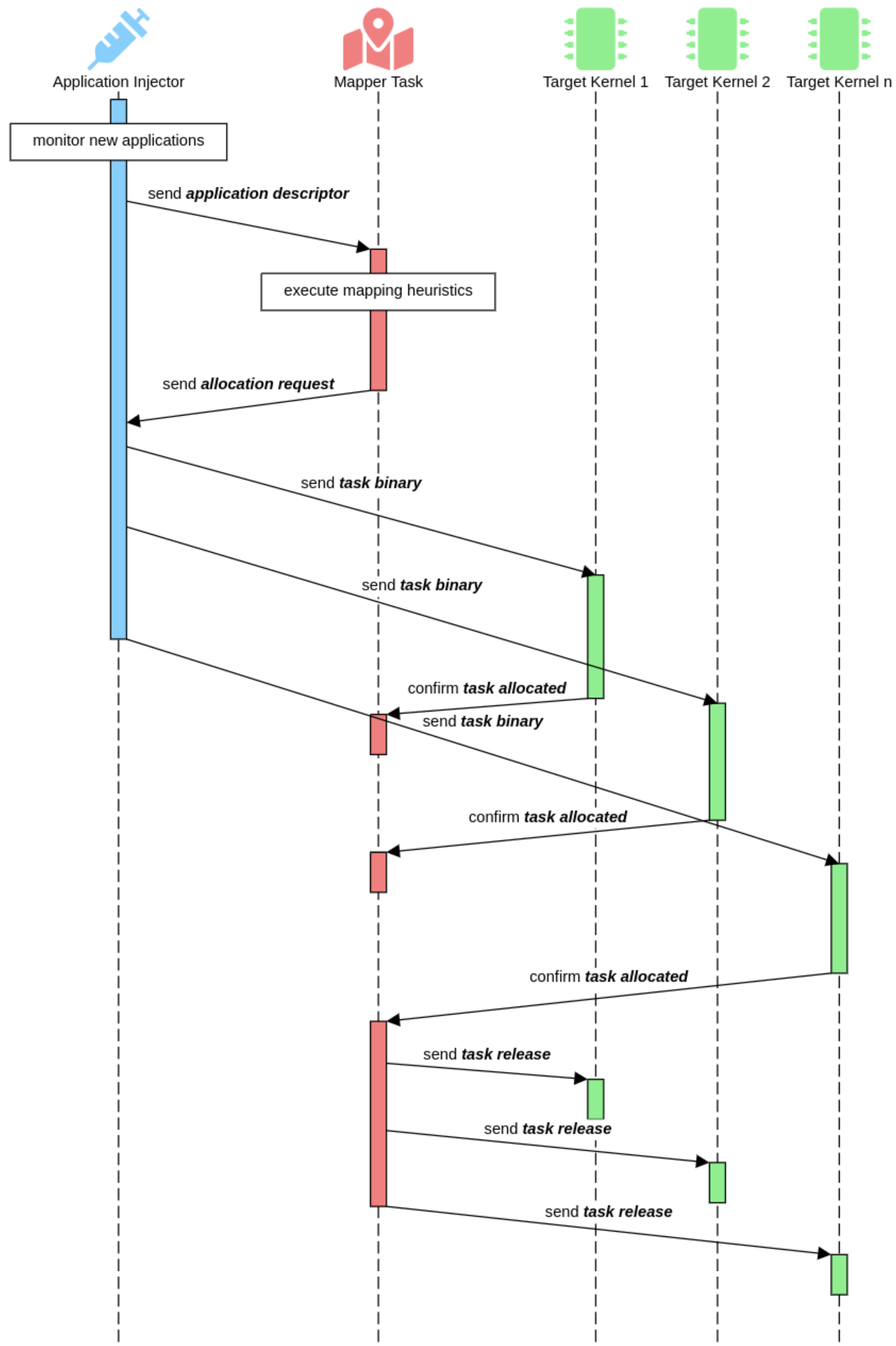


Figure 4.5: Application Injector protocol sequence diagram.

4.2.4 Migration ODA

Section 4.1 presented the three components required by the platform to support the MA framework (LLM, AE, and Communication API). The Communication API is detailed

in Section 4.2.2. The existing OS migration support in Memphis is used as the AE, triggered by messages issued by the Actuator. The QoS LLM is implemented in the task scheduler of each PE. Every time the task scheduler is called by the OS, it updates the timers of the RT tasks. After updating, it checks if the PE reached the monitoring period configured at design time to send the RT timers to the Observer task of the application.

The OS knows whom to send LLM messages to, by coupling an Observer task to each monitored user task when the mapper releases it to run, notifying its nearest Observer task. The mapper does this by checking a tag inserted into each task binary file, indicating whether the task is Observer, Decider, Actuator or user task, and its Observing, Deciding, and Actuating capabilities, such as QoS, migration, DVFS, or power management. This feature also allows the mapper to answer service discovery messages issued by the Observers and Deciders tasks that request the task that serves each following step in the ODA loop.

The proof-of-concept MA contains a set of ODA tasks working together to guarantee the QoS of an RT application, triggering task migration when deadline violations occur. The proof-of-concept MA is composed of the following ODA tasks:

- **Observer: RT task monitor** - Checks for deadline misses for each monitored task, and in case of occurrence of deadline miss, sends a message to the QoS Decider.
- **Decider: QoS** - Stores in a buffer with a Least Recently Used (LRU) replacement policy the latest monitored tasks by the Observer with a deadline violation counter. After a parameterizable number of deadline misses, it asks for a task migration.
- **Actuator: task migration** - Runs a mapping heuristic for a chosen task to migrate and uses the system API to apply the decision.

Note that the RT task monitor Observer could also send information to the QoS Decider about tasks meeting deadlines with sufficient slack time and processor usage to execute a DVFS Actuator when supported by the platform, which could lower the frequency of the processors running those tasks, aiming to improve power efficiency.

In the ODA set described above, the task migration actuator is also the mapper task. This choice is due to the need for the migration to run a task mapping algorithm to migrate. Therefore, the mapping procedures and structures can be reused, reducing also the management memory overhead. The mapper task maintains a global view of the system resources, but only for mapping purposes. Other management goals can be achieved in a distributed way by other MA tasks. The mapper MA task implements Decision and Actuation in the same task by running the mapping heuristic and requesting the allocation to the kernel and injectors. The mapping heuristic is detailed in Chapter 6.

4.3 Results

With the platform adapted as described in Section 4.2, the MA is evaluated by three criteria: implementation cost, maximum throughput, and its impact in an actual benchmark. Results are organized into three subsections. Section 4.3.1 evaluates the cost of adopting MA in terms of memory footprint and the number of management packets and discusses the MA modularity. Section 4.3.2 compares the effects of CBM and MA in an actual benchmark with QoS constraints. Section 4.3.3 also compares both approaches, but in terms of their performance with relation to the throughput of management actions.

4.3.1 MA implementation cost

Figure 4.6 compares the CBM and MA management binaries sizes. CBM has all management tasks inside its kernel, while the MA has the management split into RT task Observer, QoS Decider, and migration Actuator, which is also a mapper task. The MA tasks combined are 78.9% smaller than the CBM while keeping the same functionality. This reduction occurs because the MA has the advantage of no clusters to manage, which needs more structures in memory and more code for the reclustering procedures. The size difference between the kernels running in PEs executing user tasks is negligible, even with the new communication API required by the MA paradigm.

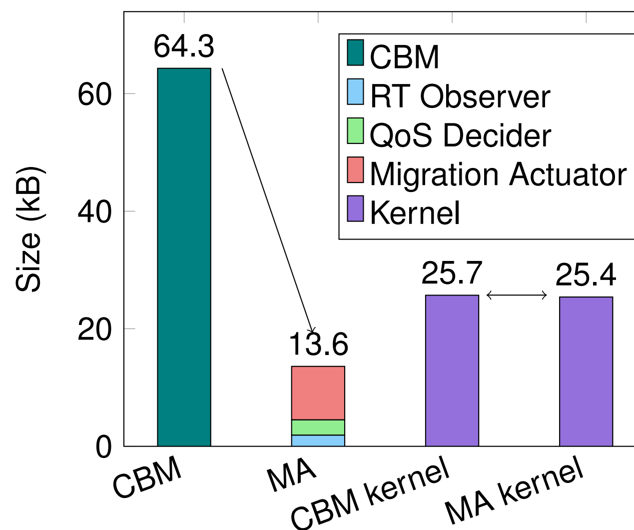


Figure 4.6: Management binaries size: CBM, MA, CBM kernel, MA kernel.

CBM requires dedicated PEs for management, with a 64.3 kB management kernel at each of these processors. Any PE can run user tasks in the MA approach, with a kernel having roughly the same size as the CBM kernel for PEs that run user tasks. The proof-of-

concept MA application requires 13.6 kB, being distributed in several PEs. This result shows that the MA makes better use of system resources without requiring more memory.

Table 4.1 shows the number of messages exchanged for CBM and MA related to Section 4.3.2 experiment. Despite resulting in about 87% more exchanged messages, MA only increased the total volume of flits by about 20% because the added monitoring and management messages are small compared to messages exchanged by user tasks. Note that this case study is simple and contains only one application and one set of MA tasks. Real scenarios execute several applications and may use many ODA tasks. Such scenarios reduce the overhead of the MA messages since the distance, in hops, between user tasks and management tasks reduces by using the MA approach since the MA tasks can be mapped to strategic regions.

Table 4.1: Message exchange in CBM and MA.

	CBM	MA
Number of messages	1,443	2,698
Number of flits	77,257	92,830

Another advantage of the MA is modularity. While the CBM approach can only vary the size of its clusters at design time, changing the number of managers, the MA platform can vary the number of ODA tasks and change the management goals at runtime by adding a new set of ODA tasks. Besides modularity, another advantage is portability, which allows the reuse of ODA tasks in other platforms by updating the LLM, AE, and communication calls to the target platform API.

4.3.2 MA case study

Experiments in this Section adopt a 3x3 system to verify the MA feasibility. This small many-core corresponds to one cluster in the CBM and is scalable to larger systems by increasing the number of clusters. The benchmark is Dijkstra's shortest path algorithm, partitioned into seven tasks. In the MA approach, PEs 0x0, 0x1, and 0x2 receive the task mapping, QoS Decider, and RT task monitor tasks. The CBM manager is mapped at PE 0x0. Both management approaches allow up to four 32 kB tasks per PE.

The RT constraints of all Dijkstra's tasks have the same deadline and execution time, and each task is configured to load 25% of the processor by changing the RT period. The initial task mapping of this application is one PE running four tasks while the remaining three tasks are split in one other PE each. The reason to adopt this mapping is to induce deadline misses in the PE loaded with four RT tasks with 25% load each, saturating the PE

with 100% load. The adopted migration heuristic and the monitoring window are the same for both management approaches, to make a fair comparison.

Table 4.2 evaluates the time required for the management approaches to detect and react to deadline misses and the application execution time. The first column details the events where the time is measured. The second and third columns detail the measured timestamp for CBM and MA, respectively.

Table 4.2: Timestamps for the Dijkstra's application using CBM and MA (ms).

Event	CBM	MA
1st migration request	4.99	5.29
end of 1st migration	5.16	5.48
2nd migration request	9.22	5.37
end of 2nd migration	9.39	5.58
end of application execution	11.80	11.72

CBM reacts quicker than MA for the first acting condition (configured to 3 deadline misses), firing the migration before MA. This happens due to the MA pipeline structure, with messages sent from the LLM to the Observer tasks, then from this task to the Decider that triggers the migration Actuator. *However, this pipeline behavior is the MA strength*. Observe the second acting condition. CBM misses this event because it is still finishing computing the previous decision and actuation procedures (remember that CBM executes all ODA actions in the same processor). Thus, CBM acts only in a third acting condition. As the MA has the ODA tasks split into several processors, it can almost simultaneously detect violations from different tasks.

Even with the increased number of messages due to the separated management tasks and the increased complexity of the communication API, the application executed faster using MA (last Table row). The reason behind the application speed up is the faster MA detection and actuation, which makes Dijkstra's application benefit from the migration sooner.

Figure 4.7 depicts this parallelization on a scenario where many LLMs (one for each PE) send QoS monitoring messages to the nearest Observer task. Observer tasks gather these data and pack them to the correct Decider task that sends messages based on the chosen action. Finally, the Actuation tasks apply the decisions via the AE of a chosen target PE. Note that each entity is running in parallel, showing how the ODA loop introduces parallelism to the management processes with a pipeline model that truly exploits the many-core parallel computing power.

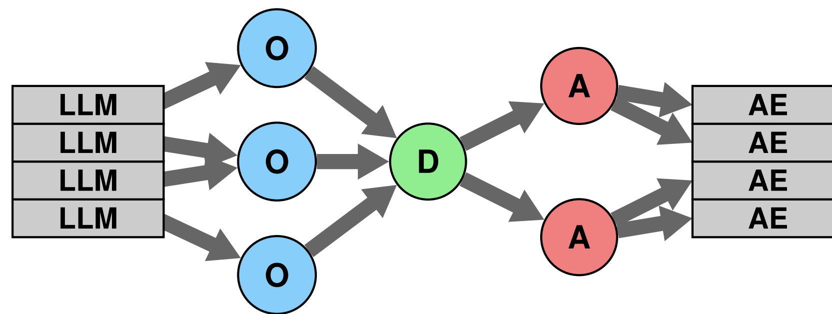


Figure 4.7: The MA pipeline model. The number of LLMs and AEs is one per PE. Observer tasks are defined at design time, while the number of Deciders and Actuators are a function of the management objectives.

4.3.3 Management throughput

The third experiment aims to saturate the management infrastructure, using the previous experimental setup, but with a synthetic task instead of Dijkstra's application. This synthetic task generates bursts of deadline miss messages, and for each message, there is an actuation (task migration).

Figure 4.8 shows on the x-axis the sequential number of the monitoring messages generated and on the y-axis the time for the management technique to handle an event (time to execute the observation, decision, and migration). This chart has three regions. The first region (messages 1 to 5) corresponds to the MA warm-up, i.e., fill the MA pipeline. In this first region, CBM acts quickly, as observed in Table 4.2. For a short period (messages 5 to 9), the CBM processor can still process the observation messages and execute the decision. However, from the ninth message onwards, the system reaches a steady state, with the throughput adapted to the processing capacity of each management method. MA is faster than CBM due to its parallel nature.

Although small (11.7%), the observed gain is obtained in a system with one application and one cluster or ODA set, seeking to illustrate the behavior of the management techniques. Such gain will increase in real-world scenarios, where multiple tasks generate monitoring data (QoS, temperature, faults), leading to several decisions.

4.4 Final Remarks

This Chapter presented a new method to manage many-core systems. The Management Application uses the ODA control loop to parallelize the management, running it as an user application, being loosely coupled to the platform (Definition 1), resulting in additional modularity (Definition 2) and portability (Definition 3). Additional reliability is also resulted by the loose coupling. Previously, if a management task entered an infinite loop

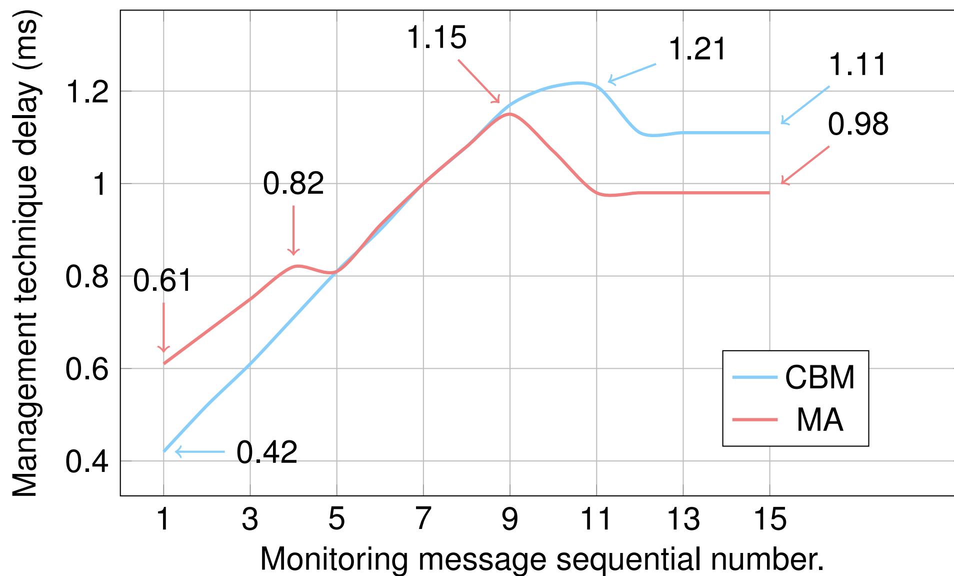


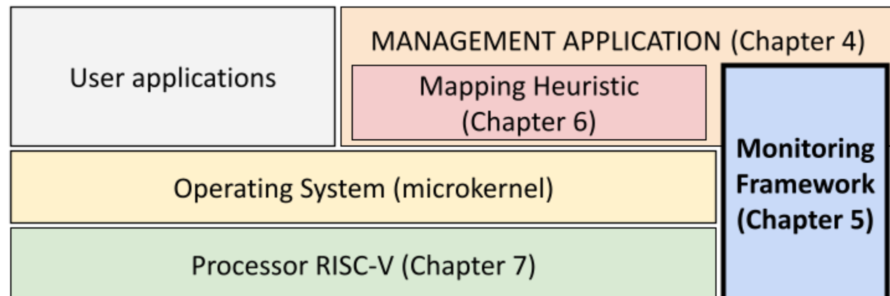
Figure 4.8: Delay from monitoring message emission until task migration completion in CBM and MA.

state, it would completely stall the OS. By separating the management tasks from the OS, if the former stalls, the latter still operates normally and schedules other tasks to execute.

A proof-of-concept is implemented in the Memphis platform. This platform originally used the CBM organization. The platform is modified to run a homogeneous kernel with essential drivers and services, with a new communication API for management tasks to replace CBM with MA.

The experiments compared the Memphis platform with CBM and MA organizations, using a set of ODA tasks to manage RT applications through task migration actuation. The following advantages compared to the state-of-the-art were identified: (*i*) there is less computational resources reservation for management; (*ii*) implementation of the management method as a distributed application. As a consequence of adopting this method, it is observed that MA improves the reactivity to missed deadlines and enhances the execution time of applications.

5. MANAGEMENT APPLICATION WITH A DEDICATED MONITORING FRAMEWORK



This Chapter presents a dedicated monitoring framework, which improves the MA performance by using the BrNoC [Wachter et al., 2017].

Chapter 4 showed that the Management Application presents an overhead concerning the number of exchanged messages for management purposes. This overhead is not caused directly by the management messages, which are small. The more significant number of messages is due to the management communication API, which has an additional handshake step, and the additional monitoring messages emitted by the LLM.

The minimum packet size transmitted by the Hermes [Moraes et al., 2004] data NoC is 13 flits in Memphis. These flits identify the service embedded in the packet, the source-target identifiers, among other functions. The management messages can be smaller since they carry simple control data, like monitoring or actuation triggers. Thus, when used by management, the standard packet structure also has an overhead that increases the communication volume and disturbs the data traffic of user applications.

The proposed solution to these issues is to use a dedicated control NoC, based on a state-of-the-art broadcast NoC, called BrNoC. BrNoC is used to transmit small management and monitoring messages. This network has low latency and enhanced fault tolerance due to the broadcast transmission that results in a flooding behavior, being well suited for management purposes.

The increased monitoring message volume of the proposed MA also causes an increase in the number of generated interrupts to processors executing Observer tasks. The BrNoC is suitable for integrating to the DMNI, due to the BrNoC characteristic of having messages composed by a single flit. Thus the hardware necessary to connect the BrNoC to the DMNI is reduced due to the absence of logic to control the packet flow. Exploiting the DMNI allows direct memory access, eliminating interrupts generated by monitoring messages.

This Chapter is organized as follows. Section 5.1 presents the broadcast network used in this work and the monitoring framework implementation. Section 5.2 details the management messages sent through the broadcast network. Section 5.3 presents a comparative

analysis of the platform with the broadcast network and monitoring framework against the platform used to evaluate the Management Application in Chapter 4. Finally, Section 5.4 concludes this Chapter with final remarks.

5.1 Framework Implementation

5.1.1 Broadcast NoC

Figure 5.1 details the BrNoC architecture used as base for this implementation. Its topology follows the same 2D-mesh, used by the data NoC, with North, South, East, West, and Local ports. The BrNoC modules are: (*i*) an Input Arbiter and Input Finite-State Machine (I-FSM); (*ii*) a central Content-Addressable Memory (CAM); and *(iii)* an Output Arbiter and Output Finite-State Machine (O-FSM).

The round-robin Input Arbiter selects the port with data to write into the CAM. To write a message to the CAM, the I-FSM must assert that the data is not in the memory and it has available space, marked by the *used* field.

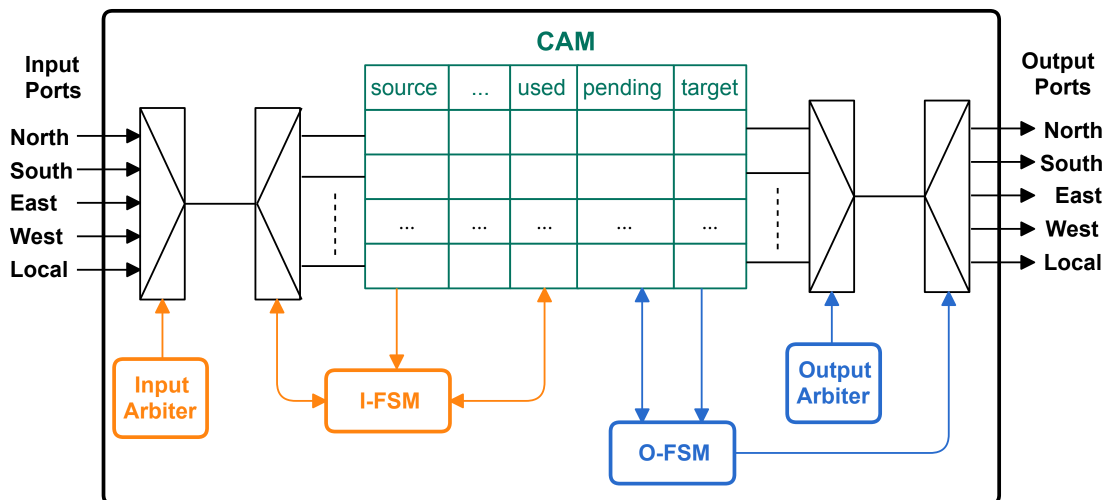


Figure 5.1: BrNoC architecture. Source: [Wachter et al., 2017].

The BrNoC Input and Output logic are independent. A round-robin Output Arbiter selects a CAM line to propagate to the outputs (broadcast). The O-FSM searches the *pending* field for messages that need to be sent. The data is propagated to all ports, except to the one where it came from.

The most relevant BrNoC feature is that all messages fit in one flit. The payload size is parameterizable, according to the constraints of the design. The advantages of 1-flit messages are: (*i*) no buffers on local ports; (*ii*) simplified switching mode, which enables

the broadcast; (*iii*) smaller router silicon area, corresponding to 50% of a Hermes router[1] [Wachter et al., 2017].

The BrNoC has four distinct services: (*i*) broadcasting to **all** PEs, which broadcasts a message to all processors; (*ii*) broadcasting to a **target**, which also broadcasts a message, but the only processor that receives the message is a defined *target*; (*iii*) broadcasting **without a target**, to send internal BrNoC control messages; and (*iv*) **unicast**, implemented through a backtracking mechanism.

Note that the O-FSM showed in Figure 5.1 only propagates a message to the local output port when it is a broadcast to **all** or when a broadcast to the **target** arrives at its destination. In this last case, the message is only sent to the local port and is not propagated to the remaining ports.

A broadcast **without a target** sent by the source of each propagated message erases each CAM line. This message releases the CAM line to receive a new broadcast.

The present work implements the *BrLite*, a simpler version of the BrNoC, without the **unicast** service. The **unicast** service is typically used to obtain a fault-free routing path between two PEs, which is not exploited by the Memphis platform in this work. Removing the **unicast** service enables a smaller CAM by discarding the need to save the broadcast path and reducing the router logic complexity.

Therefore, BrLite has only three services, listed in Table 5.1: (*i*) `ALL`; (*ii*) `TARGET`; and (*iii*) `CLEAR`, which is the only case of the broadcast **without a target** of this implementation. The `CLEAR` message is emitted by the message source 180 clock cycles after starting a given service. This value was tuned by simulation, avoiding losing and retransmitting messages in scenarios with up to 256 PEs. The advantage of releasing CAM lines automatically in hardware is the software simplification, which does not need to include the execution of the message propagation completion in its services.

Table 5.1: Services available in BrLite.

Service	**Description**
`ALL`	Send a message to all processors
`TARGET`	Broadcast a message and transmit only to the target processor
`CLEAR`	Clear a CAM line

The BrNoC is *generic*, enabling its usage parallel to other NoCs and targeting messages with any format as long as it fits in the payload field. This feature of the BrNoC is exploited in Memphis by modifying the message format to comply with BrNoC constraints.

Figure 5.2 shows the message header transmitted by the BrLite. The `ID` is a sequential identifier generated by the source processor to discard replicated messages received in the broadcast process. The second field, `SVC`, defines the service. It has 3 bits

[1] BrNoC: 8-line CAM, 84 bits per line; Hermes: 8-flit buffer depth per port, 32 bits flit width, 5 ports.

to differentiate the three services provided by the BrLite and up to five monitoring classes described in Section 5.1.2, which are treated as a `TARGET` message by the router. The `SRC` and `TGT` fields represent the source processor of a message and destination in the case of a `TARGET` message. The `PROD` field is a requirement of the Memphis platform to indicate the message producer task identifier. Each message in the BrLite carries a 40-bit `PAYLOAD`, resulting in 80-bit messages.

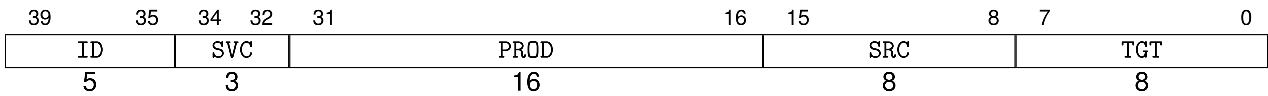


Figure 5.2: BrLite message header (40 bits). The total flit length is 80 bits with the 40-bit payload.

Additionally, each CAM line has fields dedicated to internal control, as shown in Figure 5.3. The field `USED` signalizes if the buffer entry is used by a message, and it is cleared when a `CLEAR` message arrives to its entry. Field `PEND` indicates if the message still needs to be propagated. Finally, field `ORIGIN` keeps the port whose message entered the table to avoid repropagating to the input port. These fields result in only 4 bits for internal control.

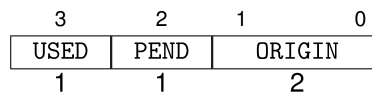


Figure 5.3: Control fields in BrLite CAM.

To summarize, the BrLite CAM has eight 84-bit lines corresponding to 672 bits. For comparison purposes, a 32-bit flit Hermes with an 8-flit buffer at each port requires 1,280 bits for temporary flit storage.

The treatment of messages received from the BrLite to the processor may occur in two ways:

1. Monitoring messages: the DMNI computes the message writing address in the observation task memory space and writes this data directly into the memory without interrupting the processor. This process is further detailed in Section 5.1.2;

2. Control messages (`ALL` and `TARGET`): stored in a buffer. The buffer is required to avoid the BrLite stall due to the time spent interrupting the processor and executing the message reception. This buffer has a parameterizable size. This implementation has eight entries, storing the 40-bit payload, the message source address, and the source task identifier, resulting in a 64-byte buffer per PE. The DMNI raises an interruption to the OS when the buffer is not empty. The services `ALL` and `TARGET` are further detailed in Section 5.2.

5.1.2 Monitoring framework

Monitoring can be costly to the PEs holding Observer tasks in the MA due to the constant reception of packets coming from the NoC and interrupting the processor to handle them. Therefore, a solution is implemented by exploiting the DMNI to receive monitoring packets directly to the Observer tasks memory space without interrupting the processor. The DMNI [Ruaro et al., 2016] joins the DMA and NI functions in a unified design to optimize the packet reception and transmission performance.

Figure 5.4 shows the original DMNI structure, with the added modules: arbiter and logic to receive messages from the BrLite. The arbiter gives priority to received broadcasts. This higher priority is due to the broadcast transmission behavior. A broadcast advances in the BrLite when all propagated ports acknowledge the reception of a given message. Thus, the reception of a given message should occur with the minimum delay to avoid performance loss in BrLite. This priority does not disturb the reception of messages coming from Hermes because broadcast messages contain a single flit, and the Hermes reception is buffered.

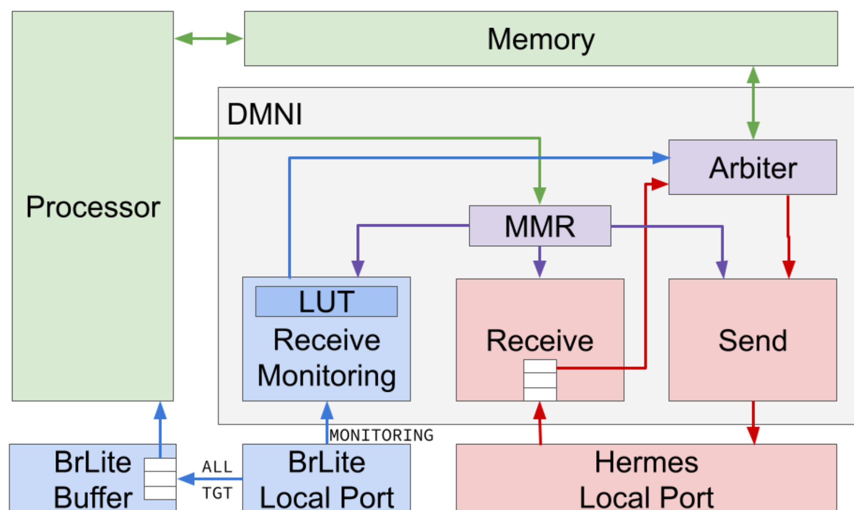


Figure 5.4: Updated DMNI architecture. **LUT** – Lookup Table

The processor configures receiving and sending through Hermes using MMRs. The MMRs configure memory addresses to access the memory directly, the memory size required for reading or writing operations, and control bits to execute the DMNI operations. The DMNI received new MMRs responsible for setting up to 5 pointers related to monitoring tables filled when receiving messages from BrLite. These tables are set by the Observer tasks using the API described in Section 5.2.

The monitoring messages are classified into five types, defined in the `SVC` field of the message header (Figure 5.2). These services are treated by BrLite as a `TARGET` service, only interrupting the destination processor. These five message types can be used to monitor constraints such as QoS, throughput, power, and temperature. Note that the

DMNI only receives the monitoring messages from BrLite. Other messages go to the BrLite buffer.

Figure 5.5 shows the format of a monitoring table line. Each monitoring table has its number of lines equal to the maximum number of tasks in the system. Each table line has two cells: one for the source task ID and one for a 32-bit monitoring payload. In an 8x8 many-core, supporting two tasks per PE (*#tasks_PE*), a monitoring table has 128 6-byte lines, resulting in 768 bytes allocated in the Observer task memory space. The number of tables allocated by the Observer task equals to the number of classes it monitors.

47 — 32	31 — 0
TASK ID	MON. DATA
16	32

Figure 5.5: Monitoring table line.

Since tasks are constantly entering and leaving the system and can migrate, the line for each monitored task can change at runtime. Therefore, the DMNI received a Lookup Table (LUT) to speed up the translation of the task ID to the monitoring table line. This LUT has the same number of lines that the monitoring table.

The LUT acts as an associative virtual memory scheme. When the DMNI receives a monitoring message and there is a valid pointer to the monitoring class, the DMNI accesses the lines of the LUT indexed by the source PE, which defines a search space of $\log_2 \#tasks_PE$. Note that the monitoring class and the source PE address are part of the BrLite message, as shown in Figure 5.2. The LUT line (*LL*) that matches the message task ID with one of the stored task IDs gives the monitoring table line address offset (`OFF`).

Figure 5.6 shows the format of a *LL*. It contains two fields: the monitored task identifier (`TASK ID`, 16 bits) and address offset (`OFF`, 1 bit). In the previous example, in an 8x8 many-core with *#tasks_PE* = 2, the LUT size is 128 17-bit lines, resulting in 272 bytes.

16 — 1	1
TASK ID	OFF
16	1

Figure 5.6: A single LUT line (*LL*).

Equation (5.1) shows how the DMNI obtains the monitoring table line (*MTL*) address.

$$MTL = MMR[msg.SVC] + (PE * \#tasks_PE + LL.OFF) * 6 \quad (5.1)$$

where: *MMR*[*msg.SVC*] – contents of the MMR related to the monitoring service embedded in the message, which contains the monitoring table base pointer; *PE* – the monitored message source PE number; 6 – the size in bytes of the monitoring table line.

As shown in Figure 5.4, a single LUT is used for all five monitoring tables. If the monitored task ID is not present in the LUT once a monitoring message arrives at the DMNI, it is written to a free LUT space. When the task finishes or migrates, a management message with the task identifier is generated to clear a LUT entry. This message is detailed in Section 5.2.

Finally, after computing the *MTL* address, the DMNI writes into the monitoring table. Note that monitoring messages can be overwritten without compromising the MA. Therefore there is no need to verify if the Observer task consumed the previous data before writing it into the table. For example, consider that the Observer task is monitoring the temperature. The samples sent by the observed task have slight differences between a couple of messages, being the behavior observed after several received messages. The same occurs when monitoring QoS. Note that this method does not apply to hard real-time constraints.

The previous discussion presented the reception of a monitoring or control message through the BrLite. In order to send a control message through BrLite, the processor does not need to configure the DMNI. Instead, it directly injects the single flit message into the BrLite.

5.2 Management Adaptation

In order to exploit the BrLite and the monitoring hardware (Section 5.1), the management software is adapted. The MA tasks can access a privileged BrLite communication API to send, but not receive, broadcast messages directed to the OS to trigger system services on the receiving end. The OS can also use the BrLite to send messages to trigger system services on other PEs.

Table 5.2 lists the BrLite MA API functions. There are two functions to send a broadcast: `send_all` and `send_tgt`. The `send_tgt` configures the BrLite to send a 40-bit payload composed of 8 bits indicating the *service* to trigger and a variable 32-bit *message* to a *target* PE. The `send_all` has the same functionality but does not specify the target PE since the message is sent to all processors.

In Table 5.2, another two functions are tailored for Observer tasks: `set_table` and `announce_observer`. The function `set_table` sets the monitoring table to a *pointer* and a defined *type*. The `announce_observer` sends a broadcast announcing the Observer task location for a defined monitoring *type*.

When an OS receives an Observation announcement message, it checks if the PE that transmitted this message is the nearest Observer of the announced class. If the condition is satisfied, the OS stores the Observer task address, that will be used by the LLM.

Table 5.2: Functions available by the BrLite MA API.

Declaration	Description
`int send_tgt(char service, unsigned message, short target)`	Sends a broadcast to a *target* containing a *service* and a *message*
`int send_all(char service, unsigned message)`	Sends a broadcast to all processors containing a *service* and a *message*
`void announce_observer(MONITOR_TYPE type)`	Announces the current Observer as a target for the monitoring *type* messages
`void set_table(void *pointer, MONITOR_TYPE type)`	Sets a table *pointer* as a target for monitoring *type* messages

This announcement replaces the need to embed the Observer task ID and address into the message sent to release the user tasks as detailed in Section 4.2.4.

The LLM presented in Section 4.2.4 is modified to use the BrLite. It sends the monitored payload via broadcast using the `TGT` service to the nearest Observer of the monitored class. This change simplified the kernel message buffer used previously to send, among other messages, the monitoring packets, reducing the kernel memory footprint by 1kB.

Using the BrLite to send management and monitoring messages through broadcast enables the system to decrease the exchanged flits previously increased by the number of management messages imposed by the MA organization, as detailed in Section 4.3.1. Other advantages of using the BrLite with MA are:

- A separate network for management messages reduces the interference in user application data traffic, enhancing performance;
- The management messages flow in a separate network which user tasks do not have access, enhancing system security;
- Broadcast messages follow a flood behavior that is fault-tolerant, thus desired for management purposes;
- The treatment performance of management messages is enhanced due to its small size;
- Provides a separate flow to monitoring messages through DMNI, reducing processor interrupts.

This work still uses Hermes for management messages that contain an extended payload. The reasoning for this is that the broadcast messages cannot carry a payload larger than what is defined by its flit, which in this implementation is 40 bits. Hermes can

carry variable-length payloads used to transmit task code, data, registers, and location of tasks on allocation and migration services.

Table 5.3 lists the messages sent through BrLite, where the first 8 bits contain the service triggered by the message, and the remaining columns are the 32-bit payload. The BrLite is used by three messages using the ALL service: (*i*) RELEASE_PERIPHERAL; (*ii*) CLEAR_LUT; (*iii*) ANNOUNCE_OBSERVER. The other 4 messages use the TARGET service: (*i*) DATA_AV; (*ii*) MESSAGE_REQUEST; (*iii*) TASK_MIGRATION; (*iv*) MONITORING.

Table 5.3: Messages sent through BrLite.

39 38 37 36 35 34 33 32	31 30 29 28 27	26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
RELEASE_PERIPHERAL	–	–	–	–
CLEAR_LUT	–	–	–	TASK_ID
ANNOUNCE_MONITOR	–	MON_TYPE	–	TASK_ID
DATA_AV	–	–	PROD_ADDR	CONS_ID
MESSAGE_REQUEST	–	–	CONS_ADDR	PROD_ID
TASK_MIGRATION	–	–	ADDR	TASK_ID
MONITORING	MON_PAYLOAD			

The messages listed in Table 5.3 are:

- RELEASE_PERIPHERAL: unlocks the many-core borders to enable communication with peripherals after loading the MA tasks. The 32-bit message field is unused. The mapper task generates this message.

- CLEAR_LUT: indicates to all PEs with an Observer task to clear the LUT with the index of the monitoring table. This message has a 16-bit field with the identifier of the task to remove from the LUT (TASK_ID). The OS that manages the task that finished or migrated generates this message. The source PE address from the message header is used to speed up the address translation.

- ANNOUNCE_MONITOR: announces to all PEs the location of an Observer task. The message has a 3-bit field with the monitoring class (MON_TYPE) and an 8-bit field with the Observer task address (ADDR). The Observer task location is inserted into the source PE address field of the message header.

- DATA_AV: first handshake step used by the MA communication API, indicating that the producer has generated a message for a consumer. The 8-bit field stands for the producer address (PROD_ADDR), and the 16-bit field is the consumer ID (CONS_ID). Although PROD_ADDR usually is the same as the SRC present in the message header, it cannot be used in case of this message is redirected when a migration occurred. The producer task identifier is obtained from the PROD field of the message header.

- MESSAGE_REQUEST: handshake for the communication API, used by user and MA tasks, indicating that the consumer has allocated buffers to receive a message. The 8-bit field stands for the consumer address (CONS_ADDR), and the 16-bit field is the producer ID (PROD_ID). Although CONS_ADDR usually is the same as the SRC present in the message header, it cannot be used in case of this message is redirected when a migration occurred. The consumer task identifier is obtained from the PROD field of the packet header.

- TASK_MIGRATION: requests a migration to the kernel, generated by the migration Actuator (mapper task). The 8-bit field stands for the target migration address (ADDR), and the 16-bit field is the identifier of the task to migrate (TASK_ID).

- MONITORING: sends a monitored data. The LLM generates it. The 32-bit field has a monitored data payload. The class exemplified in Table 5.3 is the QoS, where the 32-bit MON_PAYLOAD contains the difference between the slack time and the monitored task remaining execution time. The monitored task identifier is embedded in the PROD field of the packet header.

Note that DATA_AV and MESSAGE_REQUEST messages can still be sent by Hermes when they are directed to a peripheral to avoid the need for the peripheral to implement two network interfaces (one for Hermes and one for BrLite) and avoid peripherals flooding the network used for management, which could potentially result in a denial of service attack.

5.3 Results

The usage of BrLite for management with a monitoring framework for MA described in this Chapter are evaluated using the following criteria:

1. *Communication volume*: the number of generated flits transmitted through the networks, Hermes and BrLite.

2. *Communication performance*: the execution time of applications running in the reference platform presented in Chapter 4 versus the platform with the monitoring framework and BrLite presented in this Chapter.

3. *Management responsiveness*: the evaluation of the MA responsiveness with the added monitoring framework and BrLite, measured by the latency between actuations.

The QoS of RT tasks is evaluated with a monitoring window set to 500 microseconds in all evaluated scenarios. This window means that for every 500 microseconds, a monitoring message for all running RT tasks is generated. The reasoning for adopting this

period is that it provides the best response for both Hermes-only and BrLite with monitoring framework platforms. The period is smaller than the default scheduling timeslice for RT operating systems, such as FreeRTOS, to provide enough actuation triggers even with short simulation times.

Section 5.3.1 compares the MA platform with and without the monitoring framework and BrLite in a small scenario representing a cluster that can scale to a large many-core. Section 5.3.2 evaluates the scalability of the previous case study in a many-core with 49 PEs.

5.3.1 Cluster evaluation

Experiments in this Section adopt a 3x3 many-core as a cluster representation to evaluate the MA approach with an enhanced monitoring framework that uses the BrLite to exchange management messages. The MA in this scenario has three tasks: (*i*) mapper and migration Actuator mapped to PE 2x2; (*ii*) QoS Decider mapped to PE 2x0; and (*iii*) RT Observer mapped to PE 0x0. The RT Observer periodically polls its monitoring table to check for deadline violations. When a violation occurs, it is sent to the QoS Decider, that after three violations from the same task, triggers a task migration.

The benchmark is Dijkstra's shortest path algorithm, with seven tasks. Four tasks are statically mapped in the central PE (1x1) to simulate a scenario with high resource usage to induce deadline violations and trigger task migrations. The remaining three tasks are dynamically mapped to surrounding PEs with one hop of distance.

Figure 5.7 compares the execution timelines for both MA approaches: one with only the Hermes network, in red, and the other with the BrLite and monitoring framework, in blue. The time in the x-axis is counted since the application is released to execute, disregarding the many-core and MA setup latencies. The y-axis lists the migration decisions that occurred during the execution of the application, marked by dots and crosses. At the end of the chart, the dashed vertical line marks the application completion time for each approach.

In Figure 5.7, the first migration event occurs in the Hermes-only platform. This may seem that the proposed approach reacts later. In fact, deadline violations are being delayed due to two factors: (*i*) NoC interference reduction in application communication provided by BrLite; and (*ii*) enhanced application communication protocol due to the use of BrLite in its handshake (request).

The blue dots show that the monitoring framework with BrLite acts in bursts, further enhancing the parallel management power of the MA organization. These bursts make the three first migrations in the proposed approach occur within a 500 microseconds window. After the first three migrations, the system stops violating most deadlines, making a fourth

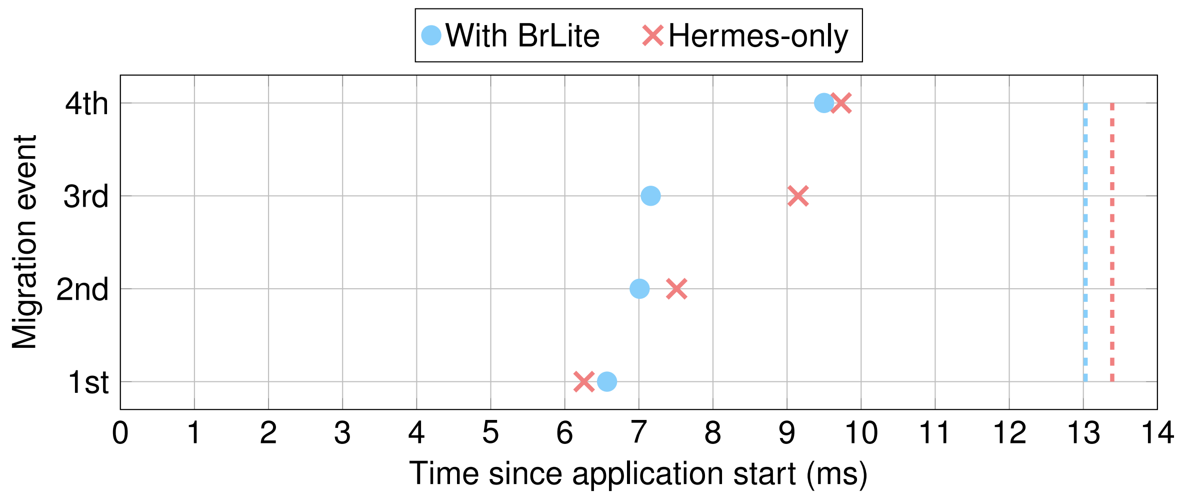


Figure 5.7: Reacting times for Hermes-only and BrLite with monitoring framework scenarios.

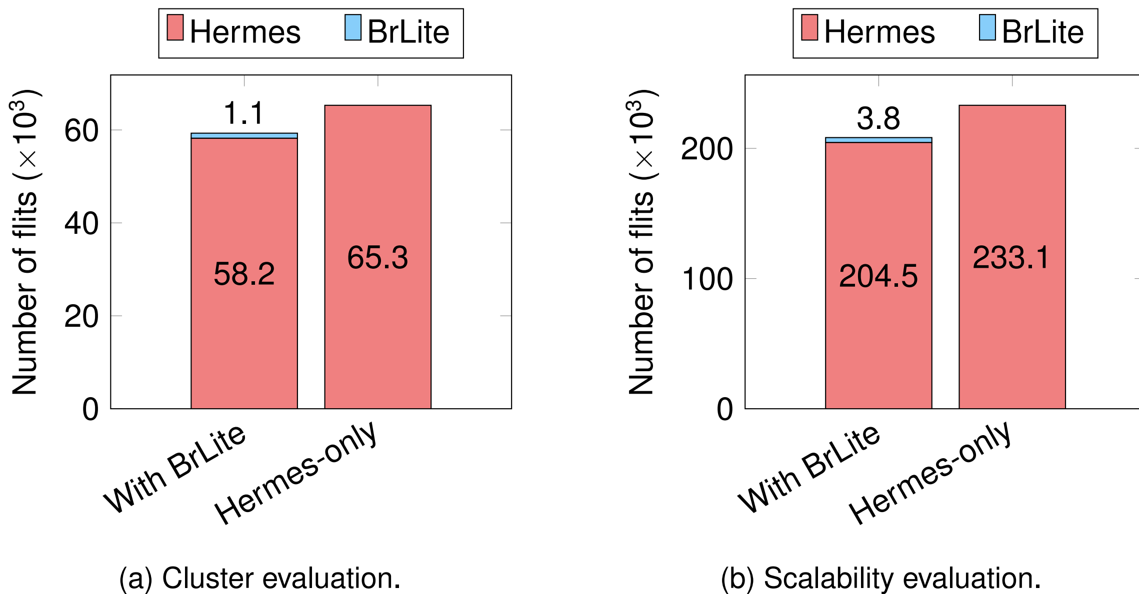
migration event later. The enhanced communication performance and the MA responsiveness contributed to a 3% reduced application execution time in the proposed approach.

Note that the minimum latency between migration events is reduced by 74% for the proposed approach. This is because the monitoring framework is not reactive like the standard LLM implemented in the Hermes-only approach. The Observer task now runs periodically, triggering a burst of actuations in a single monitoring window (1st, 2nd, and 3rd conditions). This fact, along the use of DMNI to transfer monitoring data, enhances the performance of the Observer task by reducing by 63% the interrupts to the PE holding the Observer task in this scenario.

Figure 5.8a shows the communication volume in both MA approaches. Each approach is separated in flits transmitted via Hermes, in red, and BrLite, in blue. Hermes is still used for messages with big payloads in the proposed framework approach, including the task injection, which greatly contributes to the communication volume. Despite this, removing small management messages from Hermes resulted in about an 11% reduction in communicating volume through this network. This reduction results in less switching actions in Hermes, which is more complex than BrLite. Besides, the reduction represents better communication availability to applications, translating to the observed reduction in application execution time.

5.3.2 Scalability evaluation

Experiments in this Section adopt a 7x7 many-core to evaluate the scalability of the monitoring framework and BrLite usage. The MA in this scenario has 4 Observers, virtually dividing the many-core into observation clusters. The Observers are mapped to PEs 2x2,



(a) Cluster evaluation.

(b) Scalability evaluation.

Figure 5.8: Communication volume in each NoC for the both evaluated scenarios. A BrLite flit, 80 bits, is normalized to 32 bits, i.e., multiplied by 2.5.

2x4, 4x2, and 4x4. The Deciders are fitted between the Observers, in PEs 2x3, 3x2, 2x4, and 4x2. Finally, the mapper and migration Actuator is mapped to the center of Observer and Decider tasks, which is the system center, in PE 4x4.

Table 5.4 lists the four applications that execute in this scenario. The first column lists the application name. The second column has a brief description of the application. The third column marks the parallel model used by the application. The fourth and fifth columns list the total number of tasks and the number of worker tasks, respectively. Finally, the sixth column has the PE in which the worker tasks are mapped.

Table 5.4: Applications used to evaluate the monitoring framework.

Application	**Description**	**Model**	**Tasks**	**Workers**	**PE**
Advanced Encryption Standard (AES)	Encripts and decripts data	Master-slave	9	8	5x5
Audio/Video	Audio and video encoding and decoding	Fork-join	7	5	5x1
Dijkstra	Finds the shortest path in a graph	Fork-join	7	5	1x5
Dynamic Time Warping (DTW)	Pattern recognition	Fork-join	6	4	1x1

In Table 5.4, each application has all its worker tasks mapped to the same PE to simulate a scenario with high resource usage and induce deadline violations to trigger task migrations. The remaining tasks of each application are mapped dynamically near the statically mapped workers, resulting in virtual clusters for each application surrounding the mapping location in the sixth column of the table.

Figure 5.9 presents the timeline for each application in the two compared approaches. The time in the y-axis is computed since the first application, DTW, is released to execute. The scenario with BrLite and the monitoring framework is marked in blue, and the Hermes-only platform is marked in red. The lines represent a migration event triggered by the MA.

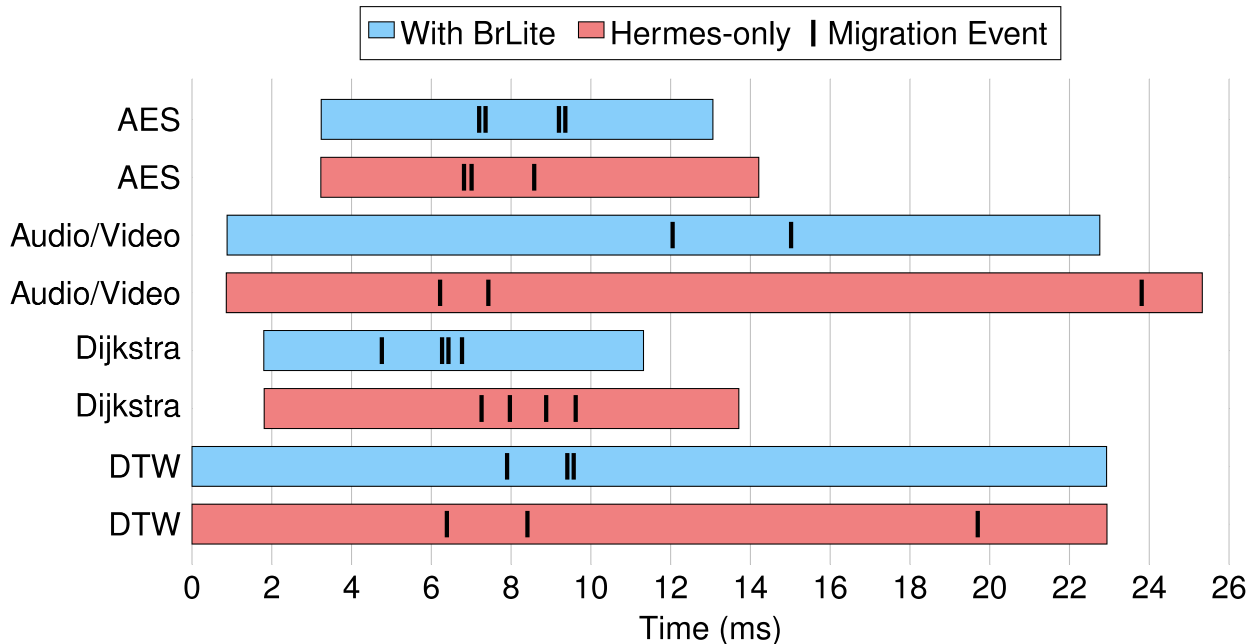


Figure 5.9: Reacting times for Hermes-only and BrLite with monitoring framework equipped scenarios.

In Figure 5.9, the applications AES, Audio/Video, and DTW start migrating earlier in the Hermes-only platform. This follows what is presented in the cluster evaluation, where enhanced communication is achieved for user applications using BrLite. Applications AES, Dijkstra, and DTW also react in instants of migration bursts, as observed in the cluster evaluation. This is enabled by the enhanced parallelism of the proposed monitoring framework, resulting in an average minimum latency between actuations reduced by 77% when disregarding the application Audio/Video due to the different number of performed migrations.

To summarize, the average execution time of applications under the proposed approach is reduced by 8%. The explanation follows the same as in the cluster evaluation. However, in this scenario, the more significant number of applications running in the system evidences the superiority of the proposed approach. On average, the four PEs holding Observer tasks were interrupted 45% fewer times in the proposed approach than in the Hermes-only platform.

Figure 5.8b pictures the communication volume in each approach. The transmitted flits by the Hermes, in red, are reduced by 12% in the proposed approach. The number of messages transmitted by the BrLite is only 2% of the emitted flits by the Hermes network

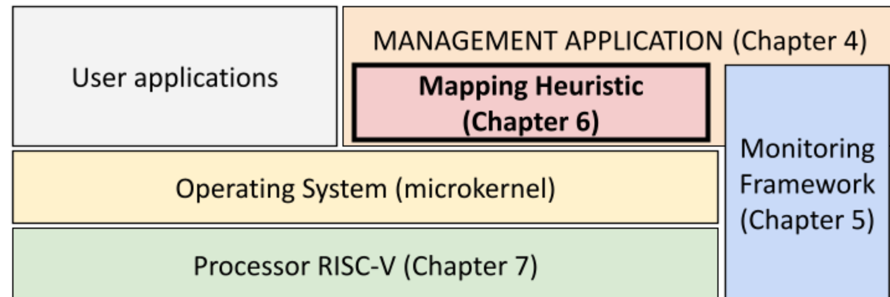
in the proposed approach, evidencing the efficiency of the broadcast NoC for management purposes.

5.4 Final Remarks

This Chapter presented a monitoring framework and a dedicated NoC for management purposes, the BrLite, used alongside the MA organization. The proposed approach achieved, on average, 8% lower application execution compared to the Hermes-only approach. The advantages of the proposed platform that enabled this reduction are: (*i*) 12% lower communication volume through the data NoC, reducing application communication congestion; and (*ii*) use of broadcast for small management messages, enhancing its handling and delivery latency.

The proposed monitoring framework further enhances the management reactivity of the MA organization. The results evidenced burst actuations explained by exploiting the DMNI, which resulted in an average of 77% less latency between actuations. This enhancement is explained by the reduced interrupts on processors holding the Observer tasks by an average of 45%.

6. MA MAPPING HEURISTIC



This Chapter presents the proposed mapping, migration, and defragmentation algorithm to be used in the Memphis platform with MA [Dalzotto et al., 2021a]. The portability of the heuristic is due to its implementation loosely coupled to the operating system. For example, mapping algorithms commonly use data structures shared with the OS, such as task tables, or commonly know the status of the running tasks. The proposal does not need to share any information with the OS, running as a standard application.

The literature proposes distributed algorithms [Singh et al., 2013], because they tend to reduce the mapping search space. These algorithms have the advantage of reducing execution time and mapping more than one application in parallel. However, there are two main weaknesses related to distributed mapping algorithms. The first one refers to the fact that, when restricting the search space to a particular region (e.g., a cluster), there may be not enough resources to map an application, due to a lack of knowledge of the other regions of the many-core. Some distributed algorithms use reclustering methods [Castilhos et al., 2013] to increase the search space if there are not enough resources to map an application in the cluster. Mapping more than one application in parallel only makes sense if the many-core can receive requests in parallel, which is usually not the case, as in the Memphis platform, which has a single interface dedicated to sending applications (*Application Injector*).

Thus, this work proposes using a centralized mapping (or a mapping with a low degree of parallelism), allowing the mapper to have a complete many-core view, improving decision making. Despite appearing to be a strategy that goes against the established literature on mapping algorithms, the technique detailed in this Chapter uses *virtual clusters* to reduce the search space and allow better load balancing on the processors. A distributed version of this heuristic can be implemented by multiple mappers running in parallel, each one negotiating with a different application source.

Figure 6.1 shows a virtual cluster, or *window*, in orange, in an 8x7 many-core. Memphis uses a 2D-mesh NoC topology. Therefore, a window is defined by the *x* and *y*

coordinates of its bottom-left corner and by *Wx* and *Wy* representing the window size in the x-axis and the y-axis, respectively.

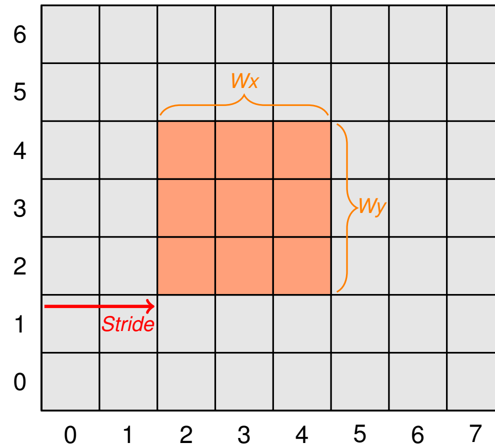


Figure 6.1: A virtual cluster, or window, in the many-core. *Wx* = 3, *Wy* = 3, and *Stride* = 2.

In Figure 6.1, the window x and y are located at the coordinate $(2, 2)$, and its W ($Wx \times Wy$) size is 3x3. The reasoning to adopt virtual clusters is to reduce the search space to map a given application, sliding the window similar to a convolution matrix, advancing each time by a value called *stride*, which in the Figure is 2. The mapper assumes that every PE supports a parameterizable number of memory pages, each page able to contain a task, without considering memory or processing constraints.

The mapping heuristic contains three main phases: window selection, mapping order, and task mapping. Before the heuristic execution, the mapper verifies if the system can run the incoming application, i.e., if there are enough pages not loaded by tasks (free pages) to map the application. A counter implements this verification, which increases with the number of application tasks to be mapped, and decreases when an application finishes.

Section 6.1 details the window selection phase for incoming applications. Section 6.2 describes the task mapping ordering algorithm. The final phase, the task mapping algorithm, is detailed in Section 6.3. Finally, Section 6.4 presents the heuristic results in terms of functional validation, computational complexity, and an evaluation of mapping quality against Memphis with CBM and hierarchical PAM.

Section 6.5 presents a defragmentation heuristic, executed when a given task finishes its execution. This heuristic aims to improve an application performance when its mapping is non-continuous, i.e., fragmented. Fragmentation is expected in dynamic workloads scenarios, and few works in the literature add this procedure in their mapping heuristics. Section 6.6 concludes this Chapter.

6.1 Window Selection Algorithm

This Section details the first phase of the heuristic, the window selection algorithm, presented in Algorithm 6.1. The goal of mapping an application into a window is to reduce the mapping search space, decreasing its computational complexity. The search procedure to find a window to receive the application starts from the last selected window. The reasoning for adopting this method is to map different applications side by side, avoiding using the same many-core region to balancing the execution load, and in the long term, increase the system lifetime.

Algorithm 6.1 four inputs are the number of tasks of the application (*app.#tasks*), the last selected window, *Wx*, and *Wy*. The algorithm always succeeds, because there is a previous resource availability verification. The algorithm returns a window, which contains the tuple $\{x, y, Wx, Wy\}$.

Line 3 advances to the next virtual window, using the current window size (*Wx*, *Wy*) and the stride value. Initial values of *Wx*, *Wy*, and *stride* are 3, 3, and 2, respectively. The loop between lines 4 to 9 searches the first window with available resources to execute the application, after the selected *last_window*, returning it if it exists. The function `window_-pages` returns the number of available memory pages in a window. If the first loop does not find a window, the same process is repeated from the first window (coordinates (0,0), line 10) up to the last window (loop between lines 11–16).

Note that these two loops are inside a “while true” external loop. Suppose the application requires more resources than those available in the current window size, or the windows have processors executing tasks belonging to other applications. In this case, it is necessary to increase the window size.

Lines 17–23 increase *Wx* or *Wy* alternately, avoiding to increase the number of PEs in the window by a large value. The two main loops rerun after increasing the window size in one dimension. This process continues until a suitable window is found. In the worst-case scenario, the window may be equal to the system size, with an application mapped with a high degree of fragmentation and a great hop count between communicating tasks.

Figure 6.2 illustrates the window sliding in a 8x7 many-core, with *Wx* = 3, *Wy* = 3, and *stride* = 2. The window slides in the x-direction by adding the stride value to the current x value. After sliding in the x-direction, the y value receives the current y value plus the stride value. Note that the figure has red windows. These windows are the ones that the stride reduces to reach the boundaries of the many-core while keeping window size.

Two events use a simpler window definition algorithm. The first one is the static mapping of some tasks of a given application, which are necessary to map the remaining task(s). The second one is the request to migrate a given task from a mapped application. In both cases, there is already a set of pre-mapped tasks. The algorithm determines

Algorithm 6.1: Window selection algorithm.

```
   Input: app.#tasks, last_window, Wx, Wy, stride
   Output: window
 1 increased_x ← false
 2 while true do
       // slide from the following after the last window until the many-core end
 3     window ← window_next(last_window, Wx, Wy, stride)
 4     while window < last possible position do
 5         if window_pages(window, Wx, Wy) ≥ app.#tasks then
 6             return window
 7         end
 8         window ← window_next(last_window, Wx, Wy, stride)
 9     end
       // slide the window from the many-core beginning until the last window
10     window ← (0, 0)
11     while window ≤ last_window do
12         if window_pages(window, Wx, Wy) ≥ app.#tasks then
13             return window
14         end
15         window ← window_next(last_window, Wx, Wy, stride)
16     end
       // increase the window alternately in x and y dimensions
17     if increased_x then
18         Wy ← Wy + 1
19         increased_x ← false
20     else
21         Wx ← Wx + 1
22         increased_x ← true
23     end
24 end
```


Figure 6.2: Virtual window sliding in a 8x7 many-core, *Wx* = 3, *Wy* = 3, and *stride* = 2.

the bounding box relative to the tasks already mapped, excluding the one to migrate, if it is the case. The bounding box is the window, defined by its left-bottom coordinates and its width/height (the *Wx*/*Wy* parameters, respectively). Thus, the task(s) to be mapped or migrated use this window as the search space. If it is necessary to increase its size, the algorithm uses the same procedure of Algorithm 6.1.

6.2 Task Mapping Ordering Algorithm

This Section describes the second phase of the heuristic, the task mapping ordering algorithm. This step is, in effect, independent of the window selection and thus could run in parallel with the first phase. The result of this phase is the mapping order used in the mapping algorithm (Section 6.3). An appropriate mapping order, although not guaranteed to be optimal, reduces the mapping fragmentation and the hop distance between communicating task pairs.

Before detailing the task mapping ordering algorithm, Definition 4 to Definition 6 detail the application model adopted by the current work.

Definition 4. Application (*App*) – is a directed and connected $CTG(T, E)$ that models each application. Each vertex $t_i \in T$ represents a task, and each edge $e_{ij} \in E$ represents communication from t_i to t_j. Assuming that edges e_{ij} are modeled implicitly in t_i (see Definition 5), an application with N tasks is represented as:

$$App = \{t_0, t_1, ..., t_{N-1}\}$$

Definition 5. Task (t_i) – is a vertex of the *CTG*. Each task t_i is a tuple with its identification and a list of communicating tasks connected by its edges e_{ij}. Communicating tasks are divided into successors and predecessors. Successors, su_i, are tasks receiving data from t_i. Predecessors, pr_i, generate data to t_i.

$$t_i = \{id, \{su_0, su_1, ...\}, \{pr_0, pr_1, ...\}\}$$

Definition 6. Initial task (in_i) – is a task t_i whose predecessors set is empty, i.e., there are no edges directed to it.

Figure 6.3 illustrates a 5-task application modeled as a CTG. In this example, t_0 is the initial task because it does not have predecessors. The t_0 successors are $\{t_1, t_2\}$, while t_4 predecessors are $\{t_2, t_3\}$.

Algorithm 6.2 details the task mapping ordering algorithm. The loop between lines 5–9 creates the *Initials* set, i.e., a set with all application initial tasks. The loop between lines 11–21 acts as a breadth-first search algorithm to traverse the CTG. At line 12, the *Order* list receives an initial task, in_i. Next, lines 14–28 add all non-added successors of in_i into

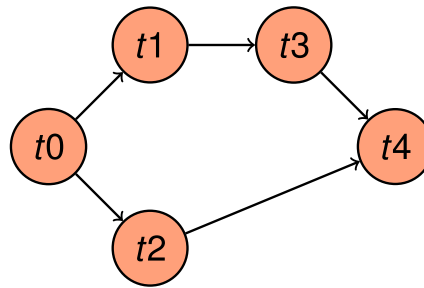


Figure 6.3: Application modeled as a CTG.

Algorithm 6.2: Task mapping ordering algorithm.

```
   Input: App // Definition 4
   Output: Order
1  Order ← ∅
2  Initials ← ∅ // Definition 6
3  task_index ← 0
4  inserted ← 0
   // search for initial tasks
5  foreach t_i ∈ App do
6  |  if t_i.Predecessor = ∅ then
7  |  |  Initials.insert(t_i)
8  |  end
9  end
10 while inserted ≠ |App| do
   |  // add successors one depth at a time starting from initials - breadth-first search
11 |  foreach in_i ∈ initials do
12 |  |  Order[inserted++] ← in_i
13 |  |  while task_index < inserted do
   |  |  |  // behave like a fifo: order the successors of the first inserted tasks
14 |  |  |  foreach su_i ∈ Order[task_index].Successors do
15 |  |  |  |  if su_i ∉ Order then
16 |  |  |  |  |  Order[inserted++] ← su_i
17 |  |  |  |  end
18 |  |  |  end
19 |  |  |  task_index++
20 |  |  end
21 |  end
   |  // add first not ordered task found as an initial to solve cyclic dependencies
22 |  Initials ← ∅
23 |  foreach t_i ∈ App do
24 |  |  if t_i ∉ Order then
25 |  |  |  Initials.insert(t_i)
26 |  |  |  break
27 |  |  end
28 |  end
29 end
30 return Order
```

the Order list. At line 19, the *task_index* increments, making the loop 14–18 to add the successors of the next task in the *Order* list into the *Order* list.

Note that an application may have cyclic dependencies that make the *Initials* set empty or the graph traversal not fully completed by the first run of the loop between lines

11–21. For example, the *Initials* set would be empty for the CTG in Figure 6.3 if t_4 sent data to t_0. Lines 22–28 clears the *Initials* set and adds the first task not present in the *Order* list into the *Initials* set. This makes the loop between lines 10–29 to run until the CTG is fully traversed, i.e., all *App* tasks are in the *Order* list.

Table 6.1 illustrates how the application is traversed using the CTG depicted in Figure 6.3.

Table 6.1: Execution of the task mapping ordering algorithm, using Figure 6.3 as input.

Iteration	Lines	*Order*	*task_index*	*inserted*	remark
	1-9	$\emptyset$	0	0	initial task: t_0
	12	t_0		1	
1	14-18 19	t_0, t_1, t_2	 1	3	t_0 successors
2	14-18 19	t_0, t_1, t_2, t_3	 2	4	t_1 successors
3	14-18 19	t_0, t_1, t_2, t_3, t_4	 3	5	t_2 successors
	19	t_0, t_1, t_2, t_3, t_4	4,5		repeats 14-18 twice, exiting

The algorithm adds the initial task, t_0, at line 12. Next, the first iteration of lines 14-18 adds the successors of the first *Order* element to the *Order* list. The counter *inserted* is now equal to three, meaning that there are 3 elements in the *Order* list. At line 19, *task_index* increments, moving the traversal index for the next iteration. At the end of the third iteration, all tasks are in the *Order* list (*inserted* = 5). The loop 14-18 repeats twice, increasing *task_index* up to be possible to exit the loop. This process is repeated for each initial task. The resulting task mapping order is t_0, t_1, t_2, t_3, t_4.

6.3 Task Mapping Algorithm

This Section details the third phase of the sliding window mapping heuristic: the task mapping algorithm. This phase uses the results of the two previous phases, the *window*, and the *Order* list. The *window* reduces the mapping complexity due to the limited search space. The *Order* list defines the sequence to map tasks to minimize the communication cost. The heuristic does not guarantee an optimal mapping result. However, Section 6.5 details a defragmentation procedure that migrates tasks with high communication cost when space in the system becomes available.

The platform designer can tune the mapping cost-function using two parameters:

- `COST_DIFF_APP`: cost related to tasks not belonging to the application being mapped running in the PE under evaluation. This cost prevents PE sharing among different applications, which is desirable for applications with RT constraints. If two or more

different applications run in the same PE, one can interfere with the other, leading to deadline misses.

- `COST_SAME_APP`: cost related to tasks of the application being mapped running in the PE under evaluation. This value defines CPU sharing. A large value distributes the tasks in several PEs, increasing the application parallelism, while small values increase the CPU sharing, reducing the number of resources used by the application.

Algorithm 6.3 details the mapping algorithm. The external loop (lines 2–24) maps the tasks sequentially, according to the *Order* list. The algorithm creates, at line 5, the *Neighbors* set with all tasks communicating with t_i.

Algorithm 6.3: Task mapping algorithm.

```
   Input: Order, window
   Output: Mapping
 1 Mapping ← ∅
 2 foreach t_i ∈ Order do
 3     cost ← ∞
 4     selected_PE ← None
 5     Neighbors ← t_i.Predecessors ∪ t_i.Successors
 6     foreach PE_xy ∈ window do
 7         if PE_xy.pages > 0 then
 8             diff_app_cost ← n_tasks_diff_app(PE_xy, t_i) * COST_DIFF_APP
 9             same_app_cost ← n_tasks_same_app(PE_xy, t_i) * COST_SAME_APP
10             comm_cost ← 0
11             foreach t_c ∈ Neighbors do
12                 if t_c is mapped then
13                     comm_cost ← comm_cost + manhattan_distance(t_i, t_c)
14                 end
15             end
16             c ← diff_app_cost + same_app_cost + comm_cost
17             if c < cost then
18                 cost ← c
19                 selected_PE ← PE_xy
20             end
21         end
22     end
23     Mapping[t_i.id] ← selected_PE
24 end
25 return Mapping
```

The loop between lines 6–22 evaluates all PEs in the *window*, with available resources to receive tasks. The functions `n_tasks_diff_app` and `n_tasks_same_app` get the number of tasks in the PE running different and same applications as the task to map, respectively, multiplying this by the respective costs. Next, lines 10–15 evaluate the communication cost, in number of hops, between t_i and its neighbor tasks already mapped. Finally, lines 16–20, select the PE with the smallest cost. The last step executed by the algorithm, line 23, is to add the PE address to the Mapping set.

The current implementation of the mapping algorithm evaluates the PEs first in the y-direction and then in the x-direction (lines 6–22 of Algorithm 6.3). The reasoning for this

order is to complement the window selection, which slides the window first in the x-direction and then in the y-direction. In this case, a better usage of the window regions is achieved by setting the *stride* to less than *Wx*, overlapping the windows in the x-direction.

Figure 6.4 shows step-by step the mapping of the CTG presented in Figure 6.3, considering one memory page per PE. Tasks are mapped as follows:

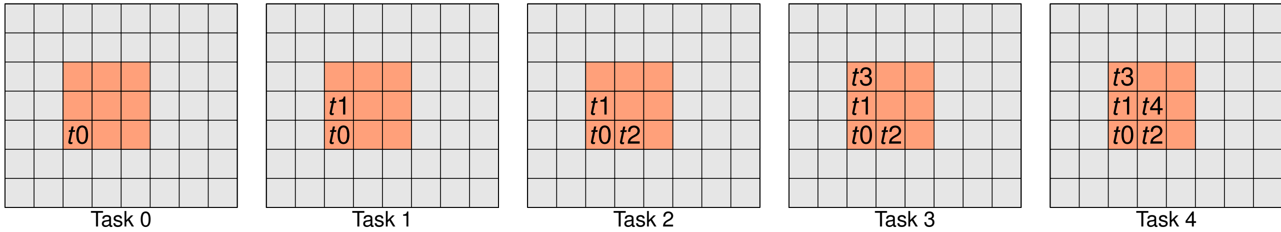


Figure 6.4: Execution of the mapping algorithm related to CTG presented in Figure 6.3, for *Wx* = 3 and *Wy* = 3.

- t_0: this is the initial task, mapped at the bottom-left corner of the window;
- t_1: communicates with t_0, mapped above it, due to the evaluation order (y-first). The communication cost between t_0 and t_1 is 1 hop, i.e., $comm_cost(t_0, t_1) = 1$;
- t_2: mapped at the right of t_0, being $comm_cost(t_0, t_2) = 1$;
- t_3: communicates with t_1, mapped above it, being $comm_cost(t_3, t_1) = 1$;
- t_4: communicates with $\{t_2, t_3\}$ mapped above t_2. being $comm_cost(t_4, t_2, t_3) = 3$.

The average estimated communication cost of this mapping is 1.2 (6 hops divided by 5 edges). The average communication cost is the main metric used to compare this mapping heuristic to other state-of-art heuristics.

Note that in the case of task migration, the heuristic does not run the task mapping ordering phase. This is because the application is already mapped, and just a single task is migrated. Therefore there is no task mapping order. Thus, it is equivalent to adding only the task to migrate to the Order set and running the mapping algorithm. With static mapped tasks, a flag indicating that the task is already mapped controls if the mapping algorithm needs to compute its costs and map the task. The task mapping ordering phase still runs the same way to obtain the correct mapping order when the successors of a static mapped task are dynamically mapped.

6.4 Mapping Results

Results in this Section are divided in functional validation and computational complexity in Section 6.4.1, and an evaluation of the mapping heuristic latency and the mapping quality in Section 6.4.2.

6.4.1 Functional validation and computational complexity

This Section presents the functional validation, enabled by a Python implementation, of the mapping heuristic and details the computational cost of the sliding window mapper. This implementation generates logs readable by the many-core debugger software [Ruaro et al., 2014], allowing debugging and algorithmic optimizations.

Figure 6.5 displays the window of the many-core debugger, showing the application mapping for a scenario in an 8x8 many-core, with a single memory page per PE. This scenario contains eleven different applications, each one marked by a different color in the Figure. The mapping generated contiguous regions for the applications, minimizing fragmentation.

PE0x7 divider 1541 RUN	PE1x7 dijkstra_0 1536 RUN	PE2x7 dijkstra_1 1537 RUN	PE3x7 dijkstra_2 1538 RUN	PE4x7 JUG1 1284 RUN	PE5x7 MEM3 1288 RUN	PE6x7 SE 1290 RUN	PE7x7 cons 2048 RUN
PE0x6 p1 1025 RUN	PE1x6 p3 1027 RUN	PE2x6 recognizer 1029 RUN	PE3x6 dijkstra_3 1539 RUN	PE4x6 VS 1291 RUN	PE5x6 JUG2 1285 RUN	PE6x6 BLEND 1280 RUN	PE7x6 prod 2049 RUN
PE0x5 bank 1024 RUN	PE1x5 p2 1026 RUN	PE2x5 p4 1028 RUN	PE3x5 dijkstra_4 1540 RUN	PE4x5 HS 1281 RUN	PE5x5 MEM1 1286 RUN	PE6x5 HVS 1282 RUN	PE7x5 cons 1792 RUN
PE0x4 taskC 770 RUN	PE1x4 taskD 771 RUN	PE2x4 taskB 769 RUN	PE3x4 print 1542 RUN	PE4x4 IN 1283 RUN	PE5x4 NR 1289 RUN	PE6x4 MEM2 1287 RUN	PE7x4 prod 1793 RUN
PE0x3 taskA 768 RUN	PE1x3 taskE 772 RUN	PE2x3 taskF 773 RUN	PE3x3 prod 2305 RUN	PE4x3 cons 2304 RUN	PE5x3 VOPREC_0 523 RUN	PE6x3 PAD_0 517 RUN	PE7x3 STRIPEM_0 519 RUN
PE0x2 aes_slave_3 3 RUN	PE1x2 aes_slave_6 6 RUN	PE2x2 aes_slave_8 8 RUN	PE3x2 iquant 257 RUN	PE4x2 idct 256 RUN	PE5x2 UPSAMP_0 520 RUN	PE6x2 VOPME_0 522 RUN	PE7x2 IQUANT_0 515 RUN
PE0x1 aes_slave_1 1 RUN	PE1x1 aes_slave_4 4 RUN	PE2x1 aes_slave_7 7 RUN	PE3x1 ivlc 258 RUN	PE4x1 print 259 RUN	PE5x1 IDCT2_0 514 RUN	PE6x1 RUN_0 518 RUN	PE7x1 ISCAN_0 516 RUN
PE0x0 aes_master 0 RUN	PE1x0 aes_slave_2 2 RUN	PE2x0 aes_slave_5 5 RUN	PE3x0 start 260 RUN	PE4x0	PE5x0 ARM_0 513 RUN	PE6x0 VLD_0 521 RUN	PE7x0 ACDC_0 512 RUN

Figure 6.5: Mapping validation on a 8x8 many-core, *Wx* = 3, *Wy* = 3, and *stride* = 2.

Figure 6.6 shows the previous scenario, mapped on a 6x6 many-core, with two pages per PE. By increasing the number of tasks each PE can run, the result is similar, with a small hop count between communicating tasks. The cost function of the task mapping penalizes the PE sharing but still tries to reduce communication costs, so tasks of the same application tend to share a PE when the number of hops between them is equal or higher than `COST_SAME_APP` (2 in the current implementation). Fragmentation is also avoided by the sliding window, where applications are mapped next to the last used window. This fact also results in application superposition only when most of the PEs in the system have at least

one task, with the window reaching the many-core boundaries and returning to its starting point.

PE0x5	PE1x5	PE2x5	PE3x5 IN 1283 RUN NR 1289 RUN	PE4x5 HS 1281 RUN MEM1 1286 RUN	PE5x5 BLEND 1280 RUN MEM2 1287 RUN
PE0x4 taskC 770 RUN taskE 772 RUN	PE1x4 taskD 771 RUN	PE2x4 taskB 769 RUN prod 2305 RUN	PE3x4 cons 2304 RUN p3 1027 RUN	PE4x4 SE 1290 RUN VS 1291 RUN	PE5x4 JUG1 1284 RUN MEM3 1288 RUN
PE0x3 taskA 768 RUN prod 2049 RUN	PE1x3 cons 2048 RUN taskF 773 RUN	PE2x3 bank 1024 RUN p2 1026 RUN	PE3x3 p1 1025 RUN recognizer 1029 RUN	PE4x3 p4 1028 RUN	PE5x3 HVS 1282 RUN JUG2 1285 RUN
PE0x2 dijkstra_1 1537 RUN aes_slave_5 5 RUN	PE1x2 dijkstra_0 1536 RUN divider 1541 RUN	PE2x2 ivlc 258 RUN dijkstra_3 1539 RUN	PE3x2 iquant 257 RUN	PE4x2 MCPU_0 516 RUN SRAM1_0 520 RUN	PE5x2 IDCT_0 515 RUN RISC_0 518 RUN
PE0x1 aes_slave_2 2 RUN aes_slave_4 4 RUN	PE1x1 dijkstra_2 1538 RUN aes_slave_7 7 RUN	PE2x1 start 260 RUN dijkstra_4 1540 RUN	PE3x1 idct 256 RUN cons 1792 RUN	PE4x1 AU_0 513 RUN RAST_0 517 RUN	PE5x1 SRAM2_0 521 RUN VU_0 523 RUN
PE0x0 aes_master 0 RUN aes_slave_1 1 RUN	PE1x0 aes_slave_3 3 RUN aes_slave_6 6 RUN	PE2x0 print 1542 RUN aes_slave_8 8 RUN	PE3x0 prod 1793 RUN print 259 RUN	PE4x0 ADSP_0 512 RUN SDRAM_0 519 RUN	PE5x0 BAB_0 514 RUN UPSAMP_0 522 RUN

Figure 6.6: Mapping validation on a 6x6 many-core with multitasking (2 tasks per PE), $Wx = 3$, $Wy = 3$, and $stride = 2$.

Table 6.2 presents the computational cost [Cormen et al., 2009] of each phase of the mapping algorithm. The first phase of the heuristic, the window selection, starts by searching a window of size W^2, given by the initial condition of $Wx = Wy$. In most cases, when the many-core is not overloaded, the first evaluated window can contain the incoming application, resulting in an average case expressed by $\Theta(W^2)$.

Equation 6.1 computes the number of evaluated windows for a given Wx and Wy in a complete iteration of the loop between lines 2–23 of Algorithm 6.1. For simplicity purposes, suppose Wx and Wy are equal, with $Wx = Wy = W$, thus, each window computation takes W^2 steps. Also suppose that the many-core X_{size} and Y_{size} are equal, with $X_{size} = Y_{size} = \sqrt{|PE|}$. Thus, Equation 6.1 can be simplified, resulting in Equation 6.2.

Table 6.2: Complexity of each phase of the heuristic. W – Window size in one dimension, N – number of application tasks, $|PE|$ – number of PEs.

	Window Selection	Task Mapping Ordering	Task Mapping
Average Case	$\Theta(W^2)$	–	$\Theta(W^2 \times N)$
Worst Case	$O(\|PE\|^2)$	$O(N)$	$O(\|PE\| \times N^2)$

$$nb_windows = \left\lceil \frac{X_{size} - Wx}{stride} + 1 \right\rceil \times \left\lceil \frac{Y_{size} - Wy}{stride} + 1 \right\rceil \quad (6.1)$$

$$nb_windows = \left\lceil \frac{\sqrt{|PE|} - W}{stride} + 1 \right\rceil^2 \tag{6.2}$$

For the worst case, W increases from an initial value of 1 until it reaches the many-core size in one dimension ($\sqrt{|PE|}$), the same way $nb_windows$ decreases until it reaches 1. To contribute to the worst case, suppose that $stride = 1$, increasing the value of $nb_windows$ for every iteration. The total number of steps required by the window selection phase of the heuristic could be described by Equation 6.3. Solving this summation, the worst-case complexity is expressed by $O(|PE|^2)$.

$$\sum_{w=1}^{\sqrt{|PE|}} w^2 \times (\sqrt{|PE|} - w + 1)^2 \tag{6.3}$$

The third column in Table 6.2 shows the complexity of the task mapping ordering phase. The worst-case complexity of this phase is expressed by $O(N)$, where N is the number of application tasks (Definition 4), since it verifies all tasks once.

The fourth column in Table 6.2 shows the complexity of the task mapping phase. The average-case complexity is expressed $\Theta(W^2 \times N)$. The average case arises when the selected window has the initial size, and each task only has a few successors or predecessors, which typically occurs. This phase searches for all PEs in the window for each application task. The worst case occurs when the window W has grown to the many-core size in one dimension $\sqrt{|PE|}$, and all tasks communicate to each other. Therefore, the worst-case complexity can be expressed by $O(|PE| \times N^2)$.

6.4.2 Mapping quality

This Section presents the mapping quality evaluation after its implementation in the Memphis many-core. The sliding window mapper for MA is compared to CBM and PAM approaches. The CBM mapping refers to the standard Memphis heuristic, which consists of selecting one PE of the cluster furthest from tasks of other applications [Tsoutsouras et al., 2018]. From this PE, the tasks are mapped based on a diamond search [Zhu and Ma, 2000]. The mapping in PAM uses the same heuristic as CBM without restricting the search space in clusters. Both approaches are distributed since CBM can map one application per cluster simultaneously, and PAM assigns initial tasks to the initial PE, which maps the application by negotiating with cluster managers to keep a global view of the system.

Experiments are conducted on a 10x10 many-core, with each PE supporting a single task. The CBM approach is divided into four 5x5 clusters. The PAM approach is also divided into the same number of clusters with hierarchical management. All evaluated scenarios, **14**, contain the same set of **9** applications, each one with a different number of

tasks, listed in Table 6.3, with a total of **78** tasks, occupying 78% of the memory pages in the many-core.

Table 6.3: Number of tasks of the applications used in the scenarios.

Application	#tasks
AES	9
Dijkstra	6
DTW	6
JPEG	5
Matrix multiplication	6
MPEG4	7
MWD	12
Sorting	15
VOPD	12
Total	78

What differentiates the 14 evaluated scenarios is the order in which the applications enter the system. One scenario maps applications from the smaller to largest number of tasks, one from the largest to the smaller number of tasks, one balanced, and 11 random scenarios.

The performance figure used to evaluate a mapping result is its average communication cost, *comm_cost*, detailed in Definition 7.

Definition 7. Average mapping communication cost (*comm_cost*) – is the average between the communication costs of mapped applications. The communication cost of an application, here, is the total number of hops between communicating pairs using a Manhattan distance measure, divided by the number of communicating pairs, corresponding to the CTG edges.

Figure 6.7 shows the mapping of scenario with a balanced application arrival order in terms of task count for the three management paradigms. The CBM mapping produced mostly contiguous mapping, except for the MWD application, in blue, which needed reclustering to fit in the many-core. CBM had a *comm_cost* = 2.27 hops. The PAM mapping solved this issued by allowing a search space equal to the system size. In addition to the 4 managers of the CBM, the hierarchical PAM needs one extra manager per application, resulting in an overhead of 13% of lost mapping space. Besides searching the entire many-core for the best location to map, PAM reduced the *comm_cost* just 1.32%, resulting in a *comm_cost* = 2.24 hops.

In Figure 6.7, the mapping using the proposed sliding window heuristic has the overhead of a single manager, which is only 1% of mapping space. Moreover, the average communication cost dropped by 22% compared to CBM, with a *comm_cost* = 1.77 hops, due to the algorithm considering a restricted search space and a task mapping order that considers the communicating task pairs.

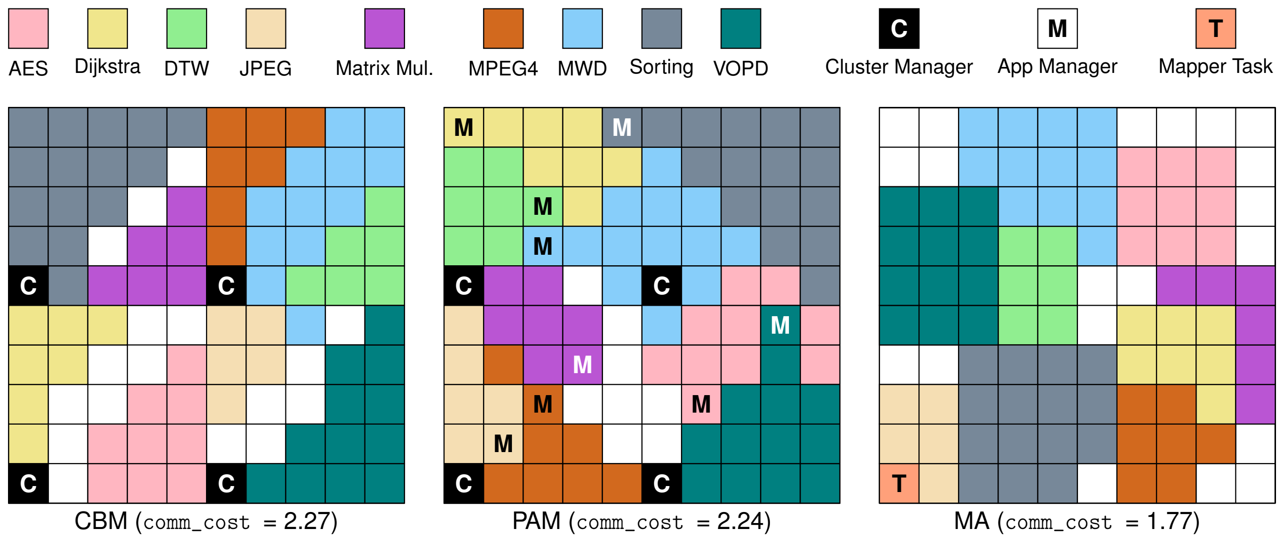


Figure 6.7: Mapping heuristic comparison between CBM, PAM, and MA.

The analysis of the 14 scenarios showed that on average, PAM and CBM produced the same average *comm_cost* = **2.33** hops, indicating that clustering does not penalize the mapping. The standard deviation of 0.07 in CBM and 0.11 in PAM shows that both approaches have low fragmentation independent of the application order, indicating that the algorithm keeps the communication distance close to the average for all scenarios. For the proposed mapping, the average *comm_cost* = **1.71** hops, 27% smaller than CBM and PAM, for the 14 scenarios. A small standard deviation, 0.06, is observed, confirming that the sliding window mapper is also independent of the order and the size of the applications entering the system.

The Figure 6.8 shows a graph of the average mapping latency in kilo clock cycles (Kcycles) for each management paradigm. Each bar represents the average heuristic latency from all evaluated scenarios with the lines marking the standard error. The latency is measured from the start to the end of the heuristic in the manager processor (CBM, PAM) or in the mapper task (MA).

In Figure 6.8, the heuristic used by CBM and PAM is the same, but PAM has the disadvantage of searching the whole many-core for the initial PE, resulting in average 4.3 times higher latency. The sliding window heuristic shows similar results than CBM, being in average just 3% slower despite being a centralized heuristic. Additionally, it is important to note that CBM runs directly at kernel level, which incurs in less execution time overhead compared to MA, which runs at application level.

The standard error showed by the sliding window mapper (1.26) is lower than CBM (2.2) and PAM (8.87). The reason explaining this result twofold: (a) the MA mapper task is centralized, thus does not need to spend time synchronizing the system status with managers, as CBM does and PAM does more heavily; and (b) the MA mapper task running the

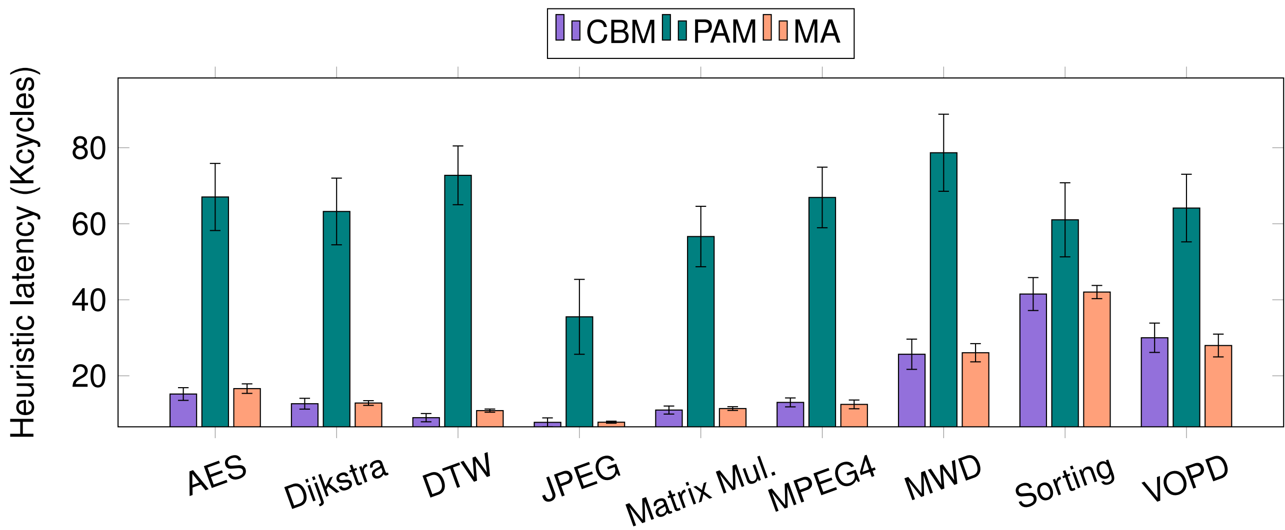


Figure 6.8: Mapping heuristic latency for different sized applications in CBM, PAM and sliding window mapper.

heuristic serves the single purpose of mapping, being more available than the CBM manager that runs another management goals in the same software.

6.5 Defragmentation

Many-cores supporting dynamic application admission assume that they can start and terminate their execution at runtime, entering and leaving the system, respectively. Such a process leads to fragmentation in the application mapping since the availability of free cores usually will be scattered through non-contiguous regions [Ng et al., 2016]. This is observed in Figure 6.7, where restricting the search space in clusters (CBM) or virtual clusters (MA) resulted in separated areas with free cores. The adverse effects of a fragmented mapping include:

- Degradation of the mapped application performance due to increased hop count;
- Interference in the performance in other applications due to the new traffic crossing regions with already mapped applications.

Figure 6.9 shows the MA scenario of Figure 6.7 when a new instance of the AES is mapped. The added application, in orange, is fragmented, i.e., it is not in a contiguous region. The other colored applications are: (*i*) AES, in pink; (*ii*) MWD, in blue; and (*iii*) Matrix Multiplication, in purple. The remaining applications are greyed out for simplicity purposes. The colored applications are highlighted because some of its mapped tasks are inside the new AES application *bounding box*. The bounding box is the minimum rectangular-shaped

PE set in which the application is mapped, marked by red lines for the new AES application in Figure 6.9

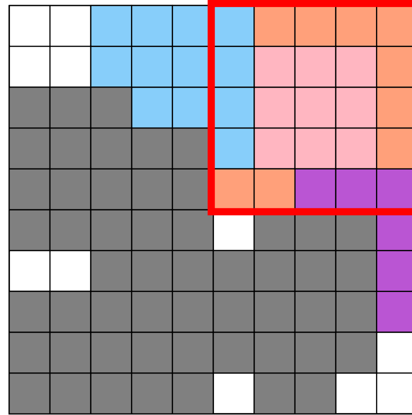


Figure 6.9: Fragmentation in a 10x10 many-core. The application in orange is fragmented, with its bounding box, in red, disturbed by the pink, purple, and blue applications.

For the mapping heuristic to identify the fragmentation, a metric should be defined. Ng et al. [Ng et al., 2016] identify scattered free cores and act on mapped applications when the scattering reaches a certain threshold. This way, a contiguous region with free cores is created before the application arrival. The Authors use three fragmentation metrics:

1. *Peri*, which is the perimeter enclosing the free cores, demanding migrations to keep the free core region contiguous all the time;
2. Half-Perimeter Length (HPL), which is the number of hops from the two most distant points in the free core region bounding box, reducing the restrictiveness of *Peri*;
3. *Minus*, which is the difference between the number of free cores in the greatest free core region and the predicted number of tasks from the next incoming application, in order to reduce to the maximum the migration quantity without compromising the mapping efficiency, but relying on the prediction accuracy.

The migration algorithm proposed in [Ng et al., 2016] needs to evaluate the many-core space to decide where to migrate the free cores. By computing a central free core with the minimum manhattan distance to other free cores, the resulting free core region is obtained. Afterward, each free core is migrated one hop at a time, i.e., the running tasks are shifted one by one to minimize the application communication cost penalty resulting from the migration. The Authors' experiments revealed less than 2.6% overhead related to the total execution time, with 41% overall execution time reduction and 42% energy reduction compared to existing mapping algorithms without defragmentation.

Pathania et al. [Pathania et al., 2017] also defragment a many-core prior to the application mapping, but in a cached system, that induces cold cache misses when migrations

occur. Their application mapping demands several constraints to its shape, such as being limited to using a power of two cores. This way, the number of free cores in a contiguous area will also be a power of two.

To defragment the mapping proposed in [Pathania et al., 2017], an area with free cores, called *fragment*, can be joined with another of the same size to be suitable for the next incoming application. The migrations involved in this process occur by swapping the fragment with tasks from a running application instead of sliding the application as in [Ng et al., 2016]. The Authors achieved an average of 15% improved performance with overall 4.85% less energy, with an execution overhead of 1.77% in a 64-core system compared to their mapping approach without migration.

Modarressi et al. [Modarressi et al., 2013] propose a defragmentation that can migrate tasks from all running applications when one application terminates. Their work aims to reduce the communication energy on a circuit-switched many-core. Compared to their mapping algorithm without migration, the Authors' migration approach achieves on average 13% better performance, and 10% reduced energy. It is also stated that bigger applications benefit more from task migrations due to the likelihood of being mapped in different non-contiguous regions. On average, each migration step moved a task by 2.69 hops.

In common, these proposals do not consider the actual cost of task migration, allowing multiple migrations simultaneously to release a region of the system. We propose a defragmentation procedure using the mapping algorithm presented in this Chapter. This defragmentation is reactive, occurring when a task exits the system. Contrary to the related work, the main **differences** of our proposal include:

- Our defragmentation acts on fragmented applications instead of unallocated areas, reducing the number of migrations;
- There is no threshold to trigger the defragmentation process;
- The defragmentation is a fine-grain process, i.e., for each finished task, the proposal evaluates if it is possible to use the freed resource to improve the mapping quality of another application.

The defragmentation method adopts two metrics:

- Communication cost: enables to evaluate the applications which are more penalized by a fragmented mapping, represented by a high hop count between communicating pairs;
- Bounding box of the application: enables to evaluate if the finished task was interfering in the application mapping space.

Algorithm 6.4 details the proposed defragmentation algorithm. It has two inputs: *freed_pe* and *Applications*. When a given task finishes its execution, it frees a memory

page in the allocated PE, called *freed_pe*. *Applications* are sorted in the reverse order by its communication cost into the *SortedApps* set (line 1) to evaluate most fragmented applications initially, i.e., with greater communication cost.

Algorithm 6.4: Defragmentation algorithm.

```
   Input: freed_pe, Applications
 1 SortedApps ← sort(Applications)
 2 foreach a_i in SortedApps do
 3 |   if freed_pe ∈ a_i.bounding_box then
 4 |   |   SortedTasks ← sort(a_i.tasks)
 5 |   |   foreach t_i ∈ SortedTasks do
 6 |   |   |   if compute_cost(t_i, freed_pe) < compute_cost(t_i, current_pe) then
 7 |   |   |   |   migrate(t_i, freed_pe)
 8 |   |   |   |   return
 9 |   |   |   end
10 |   |   end
11 |   end
12 end
```

After creating the *SortedApps* set, a loop starts looking for the candidate application to have a migrated task. The initial condition to evaluate a given application is if the freed position (*freed_pe*) is inside the application bounding box (line 3). If this condition is satisfied, a set named *SortedTasks* orders the application tasks according to the hop distance between its predecessors and successors (line 4). Thus, tasks with a higher hop count are prioritized to migrate.

The loop between lines 5-10 evaluates the cost to migrate t_i from its current position to the freed PE, according to the mapping cost, as defined in Algorithm 6.3, line 16. If the new cost is lower than the previous one, the task can migrate. When migration occurs, the algorithm finishes. Thus, only a single migration can occur per finished task. The reasoning to execute one migration per finished task is to keep the algorithm simple. Note that other tasks from the same application may also exit the system when a given task is completed. Thus, an application leaving the system can trigger multiple migrations.

Note that Algorithm 6.4 does not restrict the migration to fragmented scenarios. When an application terminates, migrations can occur to enhance another application mapping, even if it is not fragmented.

Figure 6.10 shows the previous example of Figure 6.9 when one of the “disturbing” applications finishes its execution. In this case, multiple migrations are triggered for the AES application, in orange. Note that the advantage of this method is not only the defragmentation but also the migration of tasks to locations where the mapping quality is better, translated by its lower mapping cost.

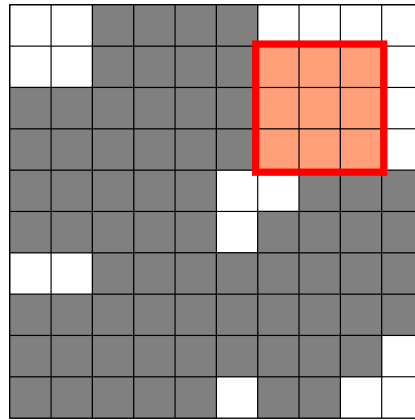


Figure 6.10: Defragmentation in a 10x10 many-core. The application in orange is defragmented, with its bounding box, in red, now smaller and not disturbed by the remaining applications.

6.5.1 Defragmentation results

This Section evaluates a case study of the proposed defragmentation algorithm. For the experiment, an 8x8 many-core without multitasking is used. The mapper task executes at the same PE as an application task, being the only PE running two tasks simultaneously.

Figure 6.11 presents four mapping states of the experiment. The first state, shown in Figure 6.11a, is the initial state of the experiment. Initially, the many-core is loaded with six applications, occupying 100% of the mapping space. In this case, applications DTW, in green, and JPEG, in purple, are fragmented. The initial *comm_cost* is 2.11 hops.

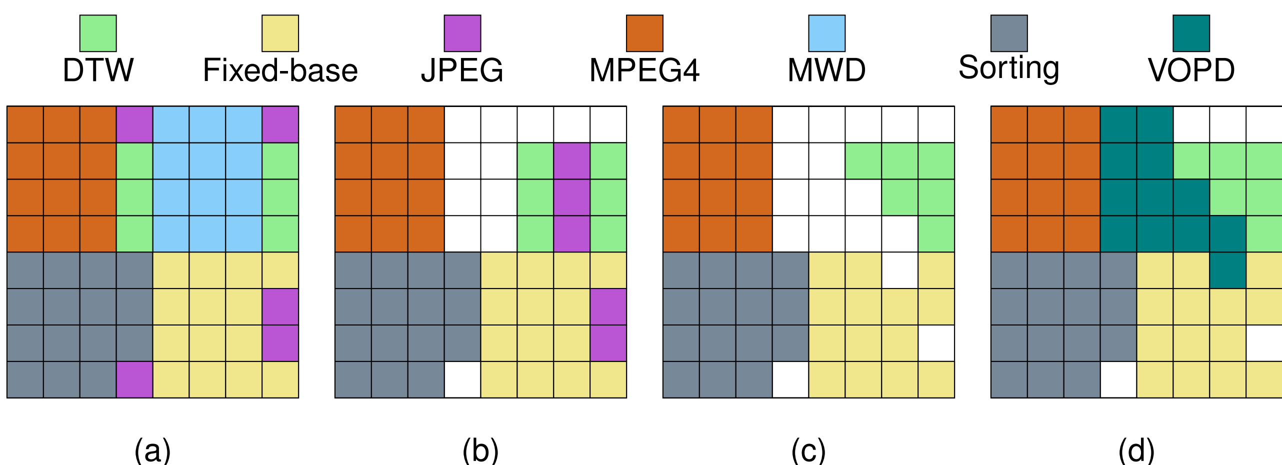


Figure 6.11: Mapping states before defragmentation (a), after defragmentation (b,c), and with an extra application after defragmentation (d).

The first time a set of migrations occurs is with the end of the MWD application, in blue. This is detailed in Figure 6.11b, with the applications DTW and JPEG achieving better

communication cost through 6 migrations. Although the Figure shows both applications are still fragmented, DTW reduced its communication cost from 3 hops to 2 hops, while JPEG reduced from 3.75 hops to 2.5 hops. Overall, both migrated applications had the cost reduced by 33%, and the total *comm_cost* to 1.87 hops, 11.4% less than the initial value.

In the mapping state shown in Figure 6.11c, there is no fragmentation. This occurred by terminating the application JPEG, triggering two migrations in the application DTW, and 1 for the Fixed-base application. DTW had its communication cost reduced by 40%, resulting in 1.5 hops. Note that despite not being fragmented, the application Fixed-base also had one migration that enhanced its communication cost, reducing by 6.2%, from 1.78 hops to 1.67 hops. The *comm_cost* reduced by 16.6%, to 1.56 hops.

Finally, a scenario where the system has terminated applications and a new incoming application, VOPD, is added is detailed in Figure 6.11d. The VOPD mapping is initiated in a system without fragmentation and is also mapped without being fragmented. Its communication cost is 1.36 hops, which kept the *comm_cost* in 1.52 hops in the system 92.2% occupied by tasks.

6.6 Final Remarks

This Chapter presented a mapping heuristic for the MA, loosely coupled to the OS, that is also unified with the task migration and application defragmentation.

The cost function of many mapping heuristics is reducing the communication energy or enhancing application performance. These cost functions are related to the hop distance reduction between communicating tasks, which is the primary cost function of the proposed mapping. Results obtained with the proposed heuristic are superior to state-of-the-art approaches due to a centralized decision process, which allows the mapping heuristic to make decisions based on the global state of the system. Although the literature presents distributed approaches to reduce the computational complexity, we demonstrate that our approach does not penalize the execution time due to the adoption of virtual clusters.

The characteristics of the virtual clusters also enhance the communication on XY routing schemes, prioritizing the application mapping on rectangular areas and decreasing the communication interference between different applications.

The presented heuristic has a built-in defragmentation procedure. Contrary to the state-of-the-art, it acts on fragmented applications instead of fragmented free spaces and enhances mapped applications that are not fragmented. This defragmentation reduced the average communication cost of the applications mapped in the system by up to 30%.

7. RISC-V INTEGRATION

RISC-V is an open-source ISA, allowing its use royalty-free in open-source projects or proprietary products. Its creators avoided defining implementation details as much as possible, allowing efficient microarchitecture designs for different purposes and technologies, with support for highly-parallel multicore or many-core implementations [Waterman et al., 2016a]. Despite this, RISC-V instructions are typically designed to execute in one clock cycle as long as cache misses are ignored [Patterson and Waterman, 2017].

The Authors in [Patterson and Waterman, 2017] argue that *incremental* ISAs, which are the conventional approach to architectures, lead to every new design having to implement the mistakes of past extensions to maintain backward binary compatibility. RISC-V offers a base integer instruction set of 32 or 64 bits, with a predefined set of available *extensions*, such as multiply-divide (M), atomic operations (A), and single (F), double (D), and quadruple (Q) precision floating-point. RISC-V can also use extensions to aid simple designs, such as using the embedded (E) base integer ISA, which has 16 instead of 32 registers present in the standard ISA, and using the compressed instructions (C) extension [Waterman et al., 2016a].

RISC-V is also extensible concerning its privilege levels that protect software stack components. RISC-V has four privilege levels: (*i*) hypervisor, for virtualization purposes; (*ii*) machine, for low-level hardware access; (*iii*) supervisor, for OS execution; and (*iv*) user, for conventional application execution. Only machine-level implementation is mandatory. Although, to enable a secure embedded system, at least user and machine modes should be used, while the supervisor mode is intended for Unix-like operating systems [Waterman et al., 2016b].

Recently, RISC-V was used in many-core platforms. OpenPiton+Ariane [Balkind et al., 2019] is a platform composed of general-purpose 64-bit RISC-V cores, where each PE has a private cache and a shared L2 cache slice, interconnected by an NoC. This platform has its own set of peripherals and can be validated by simulation, FPGA, or ASIC implementations.

BlackParrot [Petrisko et al., 2020] is another general-purpose 64-bit RISC-V core platform. A PE in this platform can be a RISC-V core, an L2 cache, an accelerator, or a peripheral. BlackParrot uses a collection of NoCs to guarantee cache-coherence in a 2D mesh organization. It is validated by simulation and ASIC implementation.

Jang et al. [Jang et al., 2021] propose a mechanism to keep cache coherence in many-core systems without adding cache coherence logic to the core. The Authors' primary motivation is that despite free RISC-V cores being available, most are impractical of using due to the effort needed to modify to comply with cache-coherent systems. They integrate

RISC-V cores with their proposed mechanism through an NoC and evaluate using an FPGA prototype.

This Chapter presents the integration of the RISC-V ISA to the MA-enabled Memphis platform. The motivation for this is twofold. First, to exploit the MA approach characteristics that enable the architectural portability (according to Definition 3). Second, the development of the MIPS architecture previously used in the Memphis platform is discontinued, and the MIPS company focus switched to RISC-V architectures [Turley, 2021]. The MIPS I ISA, which dates back to 1985, despite being a load-store modular architecture that shares characteristics with RISC-V, has *delay slots*, limiting its microarchitectural implementations.

This Chapter is organized as follows. Section 7.1 presents the RISC-V hardware model. Section 7.2 describes the adaptations to execute Memphis software stack on top of a RISC-V processor. Finally, Section 7.3 presents the results of the RISC-V integration.

7.1 RISC-V Hardware Support

This Section describes the RISC-V hardware model integrated into the Memphis platform with MA support. The RISC-V ISA is implemented in this work by an Instruction Set Simulator (ISS). This means that it is not a microarchitectural model. It simulates the instruction set behavior without implementing an actual microarchitecture but still complies with Memphis memory interface, interrupts, and connections. Note that it is possible to export the ISS execution statistics and fit them into an existing model to obtain actual performance and energy values.

The implemented ISS includes the RV32IM set. This means the RISC-V (RV) processor has a 32-bit integer base ISA (32I) with the multiply-divide extension (M). Figure 7.1 shows the privilege stack supported by this ISS. Applications have access to the limited user-mode ISA. The OS requires, in addition to the user-mode ISA, supervisor-mode instructions and Control and Status Registers (CSR) to execute. The RISC-V Supervisor Execution Environment (SEE) provides lower-level access to the processor through its machine-mode instructions and CSRs [Waterman et al., 2016b]. The three implemented privilege modes with the RV32IM instruction set are satisfied by 59 instructions.

Note that despite the ISS implementing the three main privilege levels of RISC-V, the Memphis platform uses only user-mode and machine-mode. This way, in Memphis, the OS is joined with the SEE and executes directly in machine-mode. The OS low-level functions and Application Binary Interface (ABI) responsible for application-OS interaction are further detailed in Section 7.2.

RISC-V has a paged virtual-memory architecture to support Unix-based operating systems. However, to keep compatibility with Memphis platform generation and operating

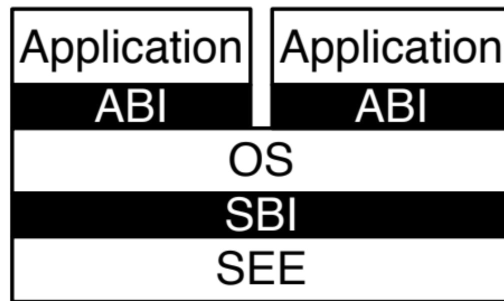


Figure 7.1: Privilege stack implemented by the RISC-V ISS. Adapted from: [Waterman et al., 2016b]. **ABI** – Application Binary Interface. **SBI** – Supervisor Binary Interface.

system without significant changes, this standard paging system is not used by the Memphis OS. Instead, a memory relocation technique is implemented as an extension to the ISA, allowing the OS to execute in machine-mode while keeping a simple virtual memory address translation to allow multitasking with low implementation cost.

Figure 7.2 shows the Machine Relative Address Register (`mrar`). This register is implemented in a space reserved for implementation-specific CSRs, and is set by the standard CSR instructions (`csrr` and `csrw` pseudo-instructions). The `mrar` register has two fields: (*i*) `OFFSET`, corresponding to the 30 upper bits of a 32-bit memory page offset; and (*ii*) `MODE`, which selects the paging operating mode by the following combinations:

- `MODE`=0 and privilege=MACHINE: the address translation is disabled, and the kernel accesses directly the physical addresses;
- `MODE`=0 and privilege=USER: the `mrar` memory relocation is enabled, adding the value of the `OFFSET` to the task virtual address to produce the physical address.
- `MODE`=non-zero: the paging mode is delegated to the configuration set in the standard RISC-V paging system. This is not used in Memphis.

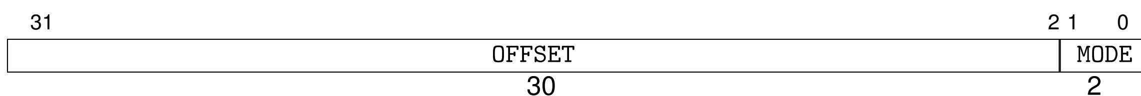


Figure 7.2: The `mrar` register.

7.2 RISC-V Software Support

This Section details the adaptations needed to execute Memphis software stack in a RISC-V core. Note that the goal is to make the least possible number of adaptations in

the software, to allow a fair comparison between the platform execution in MIPS and RISC-V processors, and evaluate the portability (Definition 3) of the MA approach.

Most adaptations are in the HAL, mostly written in assembly. The HAL is responsible for bootloading the OS, context switching, interruption and exception entry points, and loading and exiting user tasks. The only high-level change in the kernel is the removal of the structures to store a TCB for an idle task. The idle task TCB is not needed, and idling is now adequately treated by the HAL when the task scheduler queues a null task to execute. This modification resulted in a reduction of 1.25 kB in the kernel `.bss` section.

The HAL also modifies the TCBs. Each task has its TCB, which contains control data on mapping, migration, messaging, and scheduling. The TCB has a space for the task registers, program counter, and memory offset, used for context switching. The HAL modifies the TCB by obtaining a pointer to the *current* task being executed and loading or storing the register values to the structure.

The software build system uses a standard bare-metal C compiler with standard flags, except for `-fno-builtin` and `-ffreestanding` options, to provide Memphis own implementations of standard libraries and program initialization, respectively.

Address composition optimizations at link-time are enabled by default in the RISC-V compiler. These optimizations require the initialization of the *global pointer* (`gp`). Using the `gp` enables relative-addressing of global symbols without requiring additional instructions to compose a memory address. Thus reduces the number of instructions to compose load, store, jump, and branch addresses compared to absolute addressing.

The following Sections detail the modifications made to the HAL. Section 7.2.1 details the kernel bootloading process. Section 7.2.2 details the trap handling. Section 7.2.3 details the interruption handling. Section 7.2.4 details the environment call (system call) handling. Section 7.2.5 details the exception handling. Finally, Section 7.2.6 details the Application Binary Interface (ABI) to integrate user tasks to the Memphis OS.

7.2.1 Kernel bootloading

Figure 7.3 presents the kernel bootloader assembly code. When the processor switches on, it starts in machine-mode. First, it needs to set the Machine Status (`mstatus`) to zero (line 2), globally disabling interrupts during the boot process and setting the following execution privilege to user-mode. Any pending interrupt from an unknown state is also cleared (line 3), and the interrupt mask is set to accept external interrupts (lines 4-5). It is also configured not to delegate interrupts (line 7) and exceptions (line 8) to supervisor-mode or user-mode, handling both directly in machine-mode.

```
_start: # Boot in machine-mode
    csrw    mstatus, zero
    csrw    mip, zero
    li      t0, 0x800
    csrw    mie, t0

    csrw    mideleg, zero
    csrw    medeleg, zero

    .option push
    .option norelax
    la      gp, __global_pointer$
    .option pop

    la      t0, trap_handler
    csrw    mtvec, t0

    li      sp, sp_addr
    jal     main

    csrw    mscratch, sp
    j       idle_entry
```

Figure 7.3: Kernel bootloader assembly code.

Lines 10-13 detail how the global pointer is set by disabling the linker optimization temporarily to load the `__global_pointer$` value defined by the linker. The vector mode is configured to *direct* addressing, with both interrupts and exceptions entries set to the `trap_handler` address (lines 15-16). Finally, the *stack pointer* (`sp`) is loaded with the highest address of the kernel memory page, defined at compile-time, and the *main* high-level kernel function is called (lines 18-19). On return from *main*, the kernel `sp` is saved to the Machine Scratch register (`mscratch`), where the user tasks cannot access, and the execution jumps to an idle procedure entry, where it will wait until an interrupt is received (lines 21–22).

7.2.2 Kernel trap handling

Figure 7.4 presents the trap handler assembly code. A trap can occur due to: (*i*) interrupts (Section 7.2.3); (*ii*) exceptions (Section 7.2.5); and (*iii*) environment calls (Section 7.2.4). On a trap occurrence, the execution jumps to the `trap_handler` label. First, it needs to swap the trapped task `sp` with the kernel `sp` saved in the `mscratch` register (line 2). Then, a minimum context is saved to the stack until it is known which trap occurred (lines 3-5). The kernel `gp` is restored to access global symbols (lines 7-10).

The first verification in Figure 7.4 is whether there was a task running before the trap or if the system was idling (lines 13-14). If the system was idling, an interrupt occurred, which will jump to the high-level interrupt entry (`isr_entry`) described in Section 7.2.3 without

```
1  trap_handler:   # Switched from user- to machine-mode
2      csrrw    sp, mscratch, sp
3      addi     sp, sp, -8
4      sw       gp, 4(sp)
5      sw       s0, 0(sp)
6
7      .option push
8      .option norelax
9      la       gp, __global_pointer$
10     .option pop
11
12 # If no task is running, interrupt occurred
13     lw       s0, current
14     beqz     s0, isr_entry
15 # Else
16     addi     sp, sp, -8
17     sw       t1, 4(sp)
18     sw       t0, 0(sp)
19 # If task was interrupted
20     csrr     t0, mcause
21     li       t1, INTR_MASK
22     and      t1, t1, t0
23     bnez     t1, intr_handler
24 # Else if is ecall
25     addi     sp, sp, 8
26     li       t1, ECALL_MASK
27     and      t1, t1, t0
28     bnez     t1, ecall_handler
29 # Else, continues to exception handling
```

Figure 7.4: Kernel trap handling assembly code.

saving task context. Otherwise, the kernel saves more task context to the stack (lines 16-18) to verify whether the task was interrupted, called the environment, or generated an exception.

The trap cause is loaded from the Machine Cause register (`mcause`) at line 20. It is verified against an interrupt mask to jump to the `intr_handler` (lines 21-23), described in Section 7.2.3, or if an exception occurred. The only exception supported by Memphis is the *environment call from user-mode*, which is a system call, also verified by a mask (lines 26-28), which jumps to the `ecall_handler` described in Section 7.2.4. Otherwise, the execution continues to the exception handler described in Section 7.2.5.

7.2.3 Kernel interrupt handling

Figure 7.5 presents the interrupt handler assembly code. If a running task was interrupted, the program execution goes to the `intr_handler` in Figure 7.5a, where it saves the full context of the running task and then continues to the `isr_entry`. The context is saved to the task TCB, which its pointer is loaded to the `s0` register. Note that the `t0` register

is used to store temporary values before saving to the TCB of the task `sp`, which is in the `mscratch` (lines 3–4), and of task `gp`, `s0`, `t0`, and `t1`, which were temporarily saved in the stack (lines 5–11,13–14). The program counter of the interrupted task is obtained from the Machine Exception Program Counter register (`mepc`) and saved to the TCB (lines 38–39).

```
1  intr_handler:
2    sw     ra,  0(s0)
3    csrr   t0,  mscratch
4    sw     t0,  4(s0)
5    lw     t0, 12(sp)
6    sw     t0,  8(s0)
7    lw     t0,  0(sp)
8    sw     t0, 12(s0)
9    lw     t0,  4(sp)
10   sw     t0, 16(s0)
11   addi   sp, sp, 8
12   sw     t2, 20(s0)
13   lw     t0,  0(sp)
14   sw     t0, 24(s0)
15   sw     s1, 28(s0)
16   sw     a0, 32(s0)
17   sw     a1, 36(s0)
18   sw     a2, 40(s0)
19   sw     a3, 44(s0)
20   sw     a4, 48(s0)
21   sw     a5, 52(s0)
22   sw     a6, 56(s0)
23   sw     a7, 60(s0)
24   sw     s2, 64(s0)
25   sw     s3, 68(s0)
26   sw     s4, 72(s0)
27   sw     s5, 76(s0)
28   sw     s6, 80(s0)
29   sw     s7, 84(s0)
30   sw     s8, 88(s0)
31   sw     s9, 92(s0)
32   sw    s10, 96(s0)
33   sw    s11,100(s0)
34   sw     t3,104(s0)
35   sw     t4,108(s0)
36   sw     t5,112(s0)
37   sw     t6,116(s0)
38   csrr   t0, mepc
39   sw     t0,PC_ADDR(s0)
40 # Continue to high-
     level interrupt
     handler (isr_entry)
```

(a) Context saving procedure.

```
1  isr_entry:
2    addi   sp, sp, 8
3    li     t0, MMR_ADDR
4    lw     t1, 0x20(t0)
5    lw     t2, 0x10(t0)
6    and    a0, t1, t2
7    jal    os_isr
8    csrw   mscratch, sp
9
10 # If no task will run,
     idle
11   beqz   a0, idle_entry
12 # Else if the
     interrupted task is
     the same scheduled
13   beq    a0, s0,
       restore_minimum
14 # Else, a new task was
     scheduled
15   lw     s1, 28(a0)
16   lw     s2, 64(a0)
17   lw     s3, 68(a0)
18   lw     s4, 72(a0)
19   lw     s5, 76(a0)
20   lw     s6, 80(a0)
21   lw     s7, 84(a0)
22   lw     s8, 88(a0)
23   lw     s9, 92(a0)
24   lw    s10, 96(a0)
25   lw    s11,100(a0)
26   lw     t0,PC_ADDR(a0)
27   csrw   mepc, t0
28   lw     t0,OFF_ADDR(a0)
29   csrw   mrar, t0
30 # Continue to restore
     the remaining context
      (restore_minimum)
```

(b) High-level handling entry point.

```
1  restore_minimum:
2    lw     ra,  0(a0)
3    lw     sp,  4(a0)
4    lw     gp,  8(a0)
5    lw     t0, 12(a0)
6    lw     t1, 16(a0)
7    lw     t2, 20(a0)
8    lw     s0, 24(a0)
9    lw     a1, 36(a0)
10   lw     a2, 40(a0)
11   lw     a3, 44(a0)
12   lw     a4, 48(a0)
13   lw     a5, 52(a0)
14   lw     a6, 56(a0)
15   lw     a7, 60(a0)
16   lw     t3,104(a0)
17   lw     t4,108(a0)
18   lw     t5,112(a0)
19   lw     t6,116(a0)
20   lw     a0, 32(a0)
21
22   mret # Switch to
       user-mode
```

(c) Minimum context restoration.

Figure 7.5: Kernel interrupt handling assembly code.

In Figure 7.5b, the handler sets the arguments of the received external interrupt by reading the MMRs and then calls the high-level interrupt handler – **jal os_isr** (lines 2-7). The returned value from the interrupt is a scheduled task. If no task is returned, the execution

continues to the idle entry (line 11). If the scheduled task is the same as the interrupted task, there is no need to restore all registers, since the only modified ones are the callee-saved registers, therefore, it jumps to the minimum context restoration procedure (line 13) and further detailed in Figure 7.5c. Otherwise, if a different task from the interrupted was scheduled, the other registers should also be restored (lines 15–29) before continuing to the minimum context restoration. The `mrar` is also loaded with the task page offset present in its TCB (lines 28–29).

Figure 7.5c presents the minimum context restoration. It loads the registers from the TCB (lines 2–20) and then calls the Machine Return instruction (`mret`) in line 22, causing the execution to jump to the value present in the `mepc` register and the execution mode to change to user-mode.

7.2.4 Kernel environment call handling

Figure 7.6 presents the assembly code to handle environment calls triggered by task calls such as *Send*, *Receive*, and *Exit*. When the trap handler detects the environment call, it jumps to the `ecall_handler` in Figure 7.6a. First, it jumps to the high-level call handling (line 3), which changes two global variables: (*i*) `current`, which holds the scheduled TCB; and (*ii*) `task_terminated`, which flags if the environment call terminated the calling task.

In Figure 7.6a, after the high-level call, it verifies if the caller task terminated (lines 7–8) to restore the next scheduled task without saving the context of the terminated task. Otherwise, it continues and verifies if the next scheduled task is the same as the caller task (line 10) to return to its execution without saving or restoring any context (Figure 7.6c). In Figure 7.6a, if the caller task has not terminated, but the kernel scheduled a different task to execute, it saves the previously running task callee-saved registers (lines 12–31) and continues to the subsequent task context restoration (Figure 7.6b). Note that the context restoration still verifies if the next scheduled task is valid (line 5). Otherwise, it will idle.

7.2.5 Kernel exception handling

If the trap handler detects an exception not caused by an environment call, it continues its execution to the exception handler. This can occur when a task performs a forbidden action, such as accessing a misaligned or invalid address, or executing an invalid instruction. Figure 7.7 presents the exception handling procedure. In addition to the exception cause, the handler loads the Machine Trap Vale (`mtval`) and the `mepc` to compose the arguments before calling the high-level exception handler (lines 2–5). The high-level handler aborts the

```
ecall_handler:
  sw    a1, 36(s0)
  jal   os_syscall

# If the task terminated, no need to save context
  lw    t0, current
  lb    t1, task_terminated
  bnez  t1, restore_complete
# Else if scheduled the caller task, return without changing context
  beq   t0, s0, ecall_return
# Else, save the task callee-saved registers
  csrr  t1, mscratch
  sw    t1,  4(s0)
  lw    t1,  4(sp)
  sw    t1,  8(s0)
  lw    t1,  0(sp)
  sw    t1, 24(s0)
  sw    s1, 28(s0)
  sw    a0, 32(s0)
  sw    s2, 64(s0)
  sw    s3, 68(s0)
  sw    s4, 72(s0)
  sw    s5, 76(s0)
  sw    s6, 80(s0)
  sw    s7, 84(s0)
  sw    s8, 88(s0)
  sw    s9, 92(s0)
  sw   s10, 96(s0)
  sw   s11,100(s0)
  csrr  t1, mepc
  sw    t1,PC_ADDR(s0)
# Continue to scheduled task (restore_complete)
```

(a) Call handling.

```
restore_complete:
# If scheduled idle task, go to idling
  addi  sp, sp, 8
  csrw  mscratch, sp
  beqz  t0, idle_entry
# Else, restore scheduled task context
  lw    t1,PC_ADDR(t0)
  csrw  mepc, t1
  lw    t1,OFF_ADDR(t0)
  csrw  mrar, t1
  lw    ra,  0(t0)
  lw    sp,  4(t0)
  lw    gp,  8(t0)
  lw    t1, 16(t0)
  lw    t2, 20(t0)
  lw    s0, 24(t0)
  lw    s1, 28(t0)
  lw    a0, 32(t0)
  lw    a1, 36(t0)
  lw    a2, 40(t0)
  lw    a3, 44(t0)
  lw    a4, 48(t0)
  lw    a5, 52(t0)
  lw    a6, 56(t0)
  lw    a7, 60(t0)
  lw    s2, 64(t0)
  lw    s3, 68(t0)
  lw    s4, 72(t0)
  lw    s5, 76(t0)
  lw    s6, 80(t0)
  lw    s7, 84(t0)
  lw    s8, 88(t0)
  lw    s9, 92(t0)
  lw   s10, 96(t0)
  lw   s11,100(t0)
  lw    t3,104(t0)
  lw    t4,108(t0)
  lw    t5,112(t0)
  lw    t6,116(t0)
  lw    t0, 12(t0)

  mret  # Switch to user-mode
```

(b) Context restoration.

```
ecall_return:
  lw     s0,  0(sp)
  lw     gp,  4(sp)
  addi   sp, sp, 8
  csrrw  sp, mscratch, sp

  mret   # Switch to user-mode
```

(c) Call return.

Figure 7.6: Kernel environment call handling assembly code.

task that generated the exception and returns the TCB of the next scheduled task, which is used to call the restoration procedure (lines 7–8).

```
1  # Continued from trap handler
2      csrr      a0, mcause
3      csrr      a1, mtval
4      csrr      a2, mepc
5      jal       hal_exception_handler
6
7      mv        t0, a0
8      j         restore_complete
```

Figure 7.7: Kernel exception handling assembly code.

7.2.6 Application binary interface

The ABI is responsible for creating a layer between application and system. In Memphis, the RISC-V ABI is responsible for booting the task and providing an entry point for the environment call. Figure 7.8 presents the ABI assembly code. The initialization code provides the `gp` and `sp` set up (lines 2–6).

```
1  _start: # Boot in user-mode
2      .option push
3      .option norelax
4      la       gp, __global_pointer$
5      .option pop
6      li       sp, sp_addr
7
8      jal      main
9      mv       s0, a0
10     mv       a0, zero
11 try_exit:
12     mv       a1, s0
13     ecall    # Switch to machine-
          mode
14     beqz     a0, try_exit
```

(a) Task boot.

```
1  system_call:
2      addi     sp, sp, -4
3      sw       ra, 0(sp)
4      ecall    # Switch to machine-
          mode
5      lw       ra, 0(sp)
6      addi     sp, sp, 4
7      ret
```

(b) System call entry.

Figure 7.8: Application binary interface assembly code.

In Figure 7.8a, the task *main* is called, and its return value, the exit code, is saved to the `s0` register (lines 8–9). The `ecall` instruction calls the environment call *exit* with the first argument zeroed and the second containing the exit value (lines 10–13). Since the *exit* procedure can fail due to pending messages by the task, it retries until it succeeds (line 14) and the task is deallocated.

The environment call entry point, in Figure 7.8b, is used by the high-level implementation of the Memphis API, which will generate an environment call exception, as described in Section 7.2.4. In the ABI, the entry point saves the caller function return address (`ra`) to the stack and generates the environment call via the *ecall* instruction (lines 2–4). On return

from the environment call, the return address is recovered, and the execution returns to the caller (lines 6–7).

7.3 Results

This Section presents the experimental results of the RISC-V integration to the Memphis platform compared to the MIPS I-based Memphis. Experiments in this Section use a 4x4 cluster representation with one executing application. Each application is dynamically mapped with the algorithm described in Chapter 6, limiting the number of PE tasks to a maximum of 2.

In every scenario, the MA mapping task is mapped to the PE 3x3, with the Application Injector connected to the same PE and the MA Injector connected to the PE 0x0. The kernel, user applications, and MA are compiled with the GCC 11.2.0 using the `-O2` optimization flag in RISC-V and MIPS.

Figure 7.9 presents the total number of executed instructions of all cores for each application scenario. On average, RISC-V reduced the number of executed instructions by 27.4%, and all scenarios showed a reduction with similar rates, evidenced by a relative standard error of 1.6%.

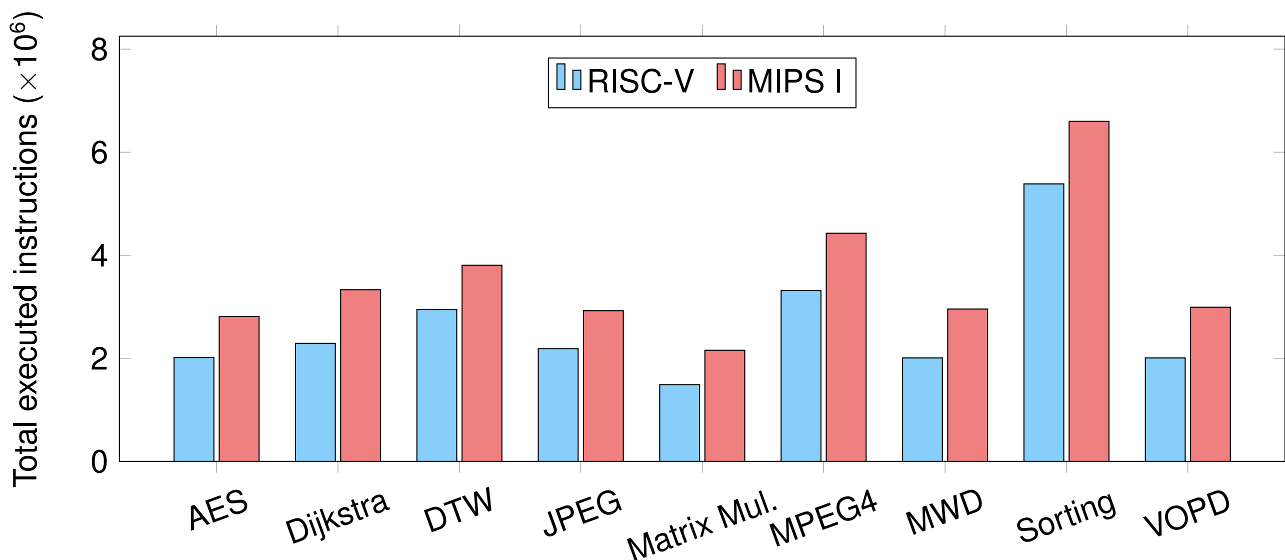
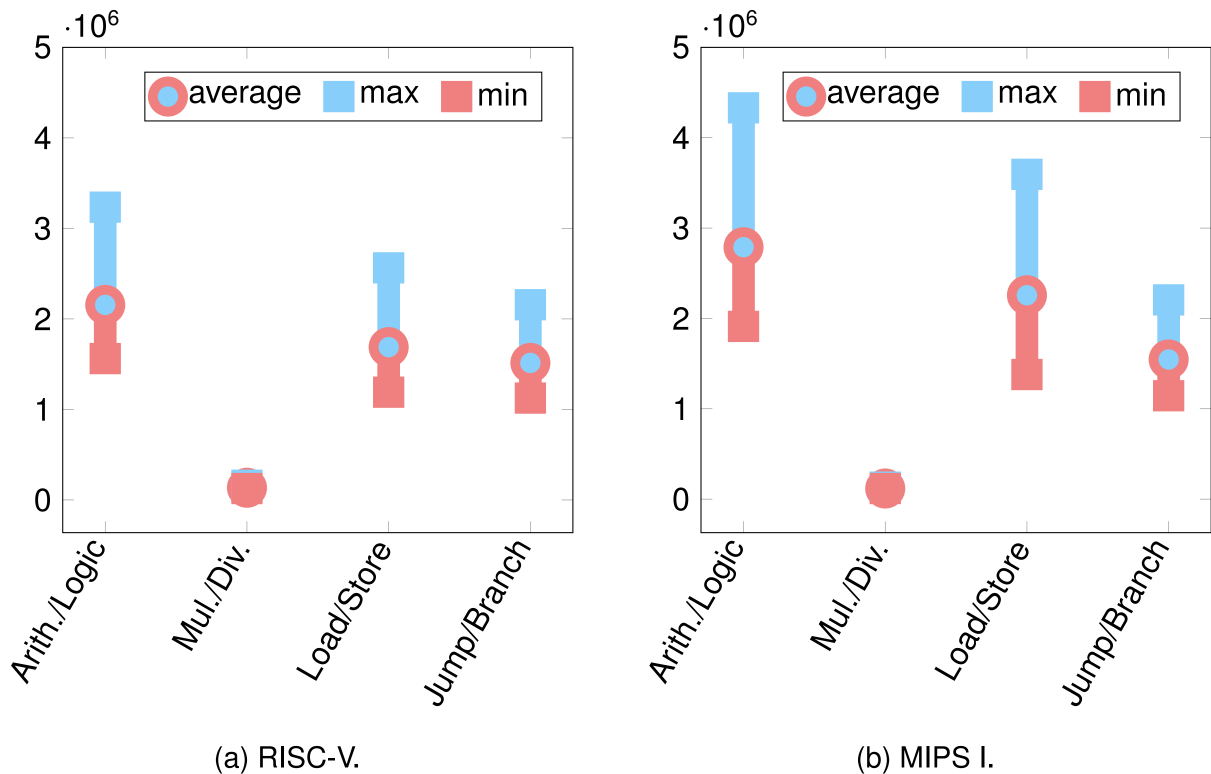


Figure 7.9: Total executed instructions of RISC-V vs. MIPS I for each application scenario.

Figure 7.10 shows a breakdown of the executed instructions classes by the evaluated scenarios. The x-axis lists the instruction classes divided in arithmetic and logic, multiplication and division, load and store, and jump and branch. Figure 7.10a details the RISC-V executed instructions compared to the MIPS I showed in Figure 7.10b.



(a) RISC-V.

(b) MIPS I.

Figure 7.10: Executed instruction breakdown for RISC-V and MIPS I.

In Figure 7.10 all instructions classes, except multiplication and division, were reduced using RISC-V. Arithmetic and logic instructions were reduced by an average of 31%, and jump and branch instructions were reduced by 2.7%. Despite increasing the multiplication and division instructions by an average of 1.8 times using RISC-V, these instructions represent only 1% of the total executed instructions by the system.

The most notable reduction in Figure 7.10 is the load and store instructions, by 36%. This can represent a significant decrease in energy because memory operations cost more than computing operations [Zaruba and Benini, 2019]. Both ISAs follow the Reduced Instruction Set Computer (RISC) philosophy and share a similar instruction set. Therefore, reducing the number of executed instructions represents gains in performance and energy.

Figure 7.11 compares the memory footprint of applications in RISC-V and MIPS I. Every application, including the Kernel and the Mapper task, presented a reduction in the memory footprint. On average, the reduction reached 10.7%. The relative standard error for all evaluated applications is 0.9%, evidencing similar reduction rates in all scenarios.

7.4 Final Remarks

This Chapter presented the integration of a RISC-V processor in the Memphis platform, replacing the MIPS I processor. The developed ISS is not cycle-accurate but capable

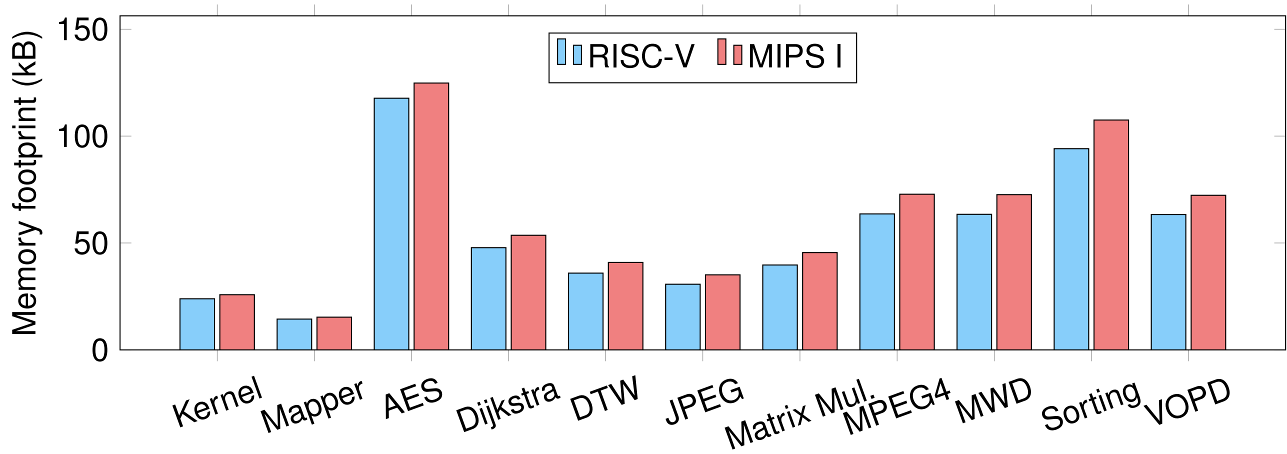


Figure 7.11: Memory footprint for different applications in RISC-V and MIPS I.

of interacting with the cycle-accurate hardware of the platform and allows evaluation related to the number of executed instructions.

The Memphis portability (Definition 3), which allowed the processor change, was enabled by two factors. First, the previously implemented HAL separated the high-level functions of the OS from the low-level operations, such as context switching and interrupt handling. The second factor is separating system management from the OS using the Management Application. These two factors are detailed in Chapter 4.

The experiments in this Chapter revealed significant optimization related to the number of executed instructions. The most relevant is the average decrease of 36% in load and store instructions, implying reduced energy and improved performance. Furthermore, the memory footprint is reduced by an average of 10% by using the RISC-V processor.

The final result of this Chapter is the Memphis-V platform, a many-core with a state-of-the-art ISA. All applications that previously ran in Memphis also executed correctly in Memphis-V, both in single- and multi-tasking.

8. CONCLUSION AND FUTURE WORK

This work presented the Management Application, an alternative for state-of-the-art management approaches with enhanced portability (Definition 3) and modularity (Definition 2), enabled by the loose coupling (Definition 1 between management and platform. Additionally were presented a monitoring framework with a broadcast network (Chapter 5), a mapping heuristic with defragmentation support tailored for the MA approach (Chapter 6), and the platform update to the RISC-V processor (Chapter 7). The final result is the Memphis-V platform, managed by the Management Application with a state-of-the-art processor.

To conclude this work, we resume below the specific objectives presented in the Introduction. We detail the contributions related to each specific objective and how they were achieved throughout the work.

1. **Remove the many-core management tasks from the target platform OS, reducing its memory footprint and discarding dedicated OSs for management purposes:**

   This is the first contribution of this work: make the management tasks loosely coupled to the OS. Chapter 4 details the OS modifications made to the Memphis platform and the MA Injector addition. With these modifications, the Memphis platform had its Cluster Managers removed, and all processors run the same OS without management functions. The MA Injector made it possible to separate user tasks and management tasks injection, increasing the platform security by inserting into the system the initial management tasks at start-up.

2. **Define the method to execute ODA tasks in userspace, making the MA modular (Definition 2) using a proper communication Application Programming Interface (API), system monitors, and actuation mechanisms:**

   This is the second contribution of this work: create a framework to support the Management Application. Chapter 4 describes the activities carried out to implement the MA framework. First, an MPI-like management communication API is developed to allow task-to-kernel, task-to-peripheral, and peripheral-to-kernel communications in both directions and in any communicating pair sequence. Then, the LLM is added as a simple monitor called periodically by the OS, and the migration procedure is the first AE supported by the system. A set of ODA tasks are defined as a RT task monitor Observer, a QoS Decider, and a task migration Actuator.

   The MA framework is enhanced using the BrLite, as described in Chapter 5. BrLite allows fast communication of small management and monitoring messages without disturbing the data NoC used by the user tasks. This network is further exploited by a proposed monitoring framework to be used as the LLM, which also employs the

Memphis DMNI to rapidly send monitored data directly into the Observer task memory space.

3. **Develop a mapping heuristic tailored to the MA:**

   The first two specific objectives exposed the need to develop a mapping heuristic. First, the ODA task set developed for the second specific objective centers around the task migration Actuation, which depends on the mapping. Second, the clustering removal makes the previous clustered mapping unoptimized, generating a large hop count between communicating tasks Therefore, Chapter 6 detailed the proposed mapping heuristic.

   The proposed mapping heuristic is based on sliding windows, achieving low computational effort even when executed as a centralized mapper in larger many-cores, with only a 3% increased execution time than a clustered approach. The mapping quality is 27% better than a state-of-the-art heuristic executed in CBM and PAM concerning the average application communication cost. This work resulted in a publication in [Dalzotto et al., 2021a].

   Furthermore, the mapping heuristic is expanded with a built-in defragmentation procedure. This procedure is triggered on any task termination and acts on fragmented applications or applications with high communication costs due to poor mapping caused by a heavy-loaded system. With few migrations, the defragmentation procedure can reduce the average communication cost of the system by up to 30%.

   Note that the mapping heuristic was implemented without any change to the OS. Therefore, its implementation evidenced the MA modularity (Definition 2). In systems with low modularity, the heuristic implementation would require modifications to the target OS.

4. **Use the Quality of Service (QoS) management objective to evaluate the proof-of-concept MA with task migration:**

   Here, the MA is evaluated compared to CBM and PAM, as detailed in Chapter 4. This is possible by using the ODA loop to act into the system from the monitored data, according to a decision heuristic and a set of Actuators developed for the second objective. Note that the main goal is not the ODA decision quality but the MA framework itself.

   The MA showed 11.7% enhanced management throughput due to its pipelined structure. Besides, the MA is loosely coupled to the OS, reducing management memory footprint by 39% and enhancing its portability (Definition 3) and modularity (Definition 2). This work is published in [Dalzotto et al., 2021b]. This framework paves the way for future works related to managing large many-core systems and multi-objective management.

Furthermore, BrLite improved the MA organization. Besides general improvements to the platform, such as enhancing user application communication and resulting in 8% decreased execution times, the BrLite also reduced the management latency by up to 77%.

5. **Make the MA organization agnostic to the hardware, turning it possible to replace the processor, evidencing portability (Definition 3):**

   This is the third contribution of this work. One objective of this work is allowing the MA tasks to be reused in other platforms, and the initial step is proving the portability of the approach. The goal is to make the MA agnostic to the hardware, making it possible to replace the processor.

   Chapter 7 details how the existing Memphis OS is ported to the RISC-V processor. Additionally, it describes a RISC-V ISS integrated into the Memphis platform, replacing the MIPS processor. The final result is a platform with a state-of-the-art processor, which reduced the executed instructions compared to MIPS by 27.4% and the memory footprint by 10.7%.

   Note that the RISC-V integration required no modifications to the management tasks of the MA. Only modifications in the OS HAL were made, which is not related to management functions. Therefore, this evidences the MA portability (Definition 3).

8.1 Future Work

This Section presents the future research directions of this work. This work can be further expanded in two directions: (*i*) enhancing the platform hardware with better hardware modeling and an actual processor; and (*ii*) enhancing the management with machine learning and a proven OS. Some future work options are:

- **Unify the hardware modeling:** Replace the platform SystemC and VHDL modeling with Verilog and SystemVerilog. This way, the same model is used for both simulation and prototyping.

- **Use a cycle-accurate RISC-V core:** Use a proven free RISC-V core available in Verilog or SystemVerilog. This core makes it possible to extract precise information from the platform execution.

- **Port the platform to another OS:** With the portability enabled by the MA, use a proven OS, such as FreeRTOS, with the Memphis platform.

- **Synthesize and prototype the Memphis-V:** Prototyping the platform enables power, performance, and area evaluation.

- **Employ machine learning to multi-objective management:** Use machine learning techniques to implement Deciders and Actuators in the MA organization. This can be done directly in software or with the aid of hardware accelerators developed as peripherals.

REFERENCES

Anagnostopoulos, I., Tsoutsouras, V., Bartzas, A., and Soudris, D. (2013). Distributed run-time resource management for malleable applications on many-core platforms. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 1–6.

Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. (2009). A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67.

Ausavarungnirun, R., Fallin, C., Yu, X., Chang, K. K.-W., Nazario, G., Das, R., Loh, G. H., and Mutlu, O. (2014). Design and evaluation of hierarchical rings with deflection routing. In *Proceedings of the IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 230–237.

Balkind, J., Lim, K., Gao, F., Tu, J., Wentzlaff, D., Schaffner, M., Zaruba, F., and Benini, L. (2019). OpenPiton+Ariane: The First Open-Source, SMP Linux-booting RISC-V System Scaling From One to Many Cores. In *Proceedings of the ACM Workshop on Computer Architecture Research with RISC-V (CARRV)*, pages 1–6.

Biggs, S., Lee, D., and Heiser, G. (2018). The Jury is in: Monolithic OS design is flawed: Microkernel-based designs improve security. In *Proceedings of the ACM Asia-Pacific Workshop on Systems (APSys)*, pages 1–7.

Carara, E. A., de Oliveira, R. P., Calazans, N. L. V., and Moraes, F. G. (2009). HeMPS - a framework for NoC-based MPSoC generation. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1345–1348.

Carlson, T. E., Heirman, W., and Eeckhout, L. (2011). Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In *Proceedings of the IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12.

Castilhos, G., Mandelli, M., Madalozzo, G., and Moraes, F. G. (2013). Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 153–158.

Catania, V., Mineo, A., Monteleone, S., Palesi, M., and Patti, D. (2016). Cycle-Accurate Network on Chip Simulation with Noxim. *ACM Transactions on Modeling and Computer Simulation*, 27(1):1–25.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press. 1313 pages.

Dalzotto, A. E., Ruaro, M., Erthal, L. V., and Moraes, F. G. (2021a). Dynamic Mapping for Many-cores using Management Application Organization. In *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 1–6.

Dalzotto, A. E., Ruaro, M., Erthal, L. V., and Moraes, F. G. (2021b). Management Application - a New Approach to Control Many-Core Systems. In *Proceedings of the IEEE Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6.

Dutt, N., Jantsch, A., Sarma, S., and others (2015). Self-Aware Cyber-Physical Systems-on-Chip. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 46–50.

Faruque, M. A. A., Krist, R., and Henkel, J. (2008). ADAM: Run-time agent-based distributed application mapping for on-chip communication. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 760–765.

Gregorek, D. and Garcia-Ortiz, A. (2018). The Agamid design-space exploration framework. *Springer Design Automation for Embedded Systems*, 22(4):293–314.

Gregorek, D., Rust, J., and Garcia-Ortiz, A. (2019). DRACON: A Dedicated Hardware Infrastructure for Scalable Run-Time Management on Many-Core Systems. *IEEE Access*, 7:121931–121948.

Haghbayan, M. H., Miele, A., Zouv, Z., Tenhunen, H., and Plosila, J. (2020). Thermal-Cycling-aware Dynamic Reliability Management in Many-Core System-on-Chip. In *Proceedings of the IEEE Design, Automation Test in Europe Conference (DATE)*, pages 1229–1234.

Hoffmann, H., Maggio, M., Santambrogio, M. D., Leva, A., and Agarwal, A. (2013). A generalized software framework for accurate and efficient management of performance goals. In *Proceedings of the IEEE International Conference on Embedded Software (EMSOFT)*, pages 1–10.

Howard, J., Dighe, S., Hoskote, Y., Vangal, S., Finan, D., Ruhl, G., Jenkins, D., Wilson, H., Borkar, N., Schrom, G., et al. (2010). A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, pages 108–109.

Huang, X., Wang, X., Jiang, Y., Singh, A. K., and Yang, M. (2020). Dynamic Allocation/Reallocation of Dark Cores in Many-Core Systems for Improved System Performance. *IEEE Access*, 8:165693–165707.

Jang, H., Han, K., Lee, S., Lee, J.-J., Lee, S.-Y., Lee, J.-H., and Lee, W. (2021). Developing a Multicore Platform Utilizing Open RISC-V Cores. *IEEE Access*, 9:120010–120023.

Kobbe, S., Bauer, L., Lohmann, D., Schröder-Preikschat, W., and Henkel, J. (2011). DistRM: Distributed resource management for on-chip many-core systems. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 119–128.

Liao, X., Jigang, W., and Srikanthan, T. (2011). A Modular Simulator Framework for Network-on-Chip Based Manycore Chips Using UNISIM. *Transactions on High-Performance Embedded Architectures and Compilers*, 4:234–253.

Liao, X. and Srikanthan, T. (2011). A scalable strategy for runtime resource management on NoC based manycore systems. In *Proceedings of the IEEE International Symposium on Integrated Circuits (ISIC)*, pages 297–300.

Manferdelli, B. J. L., Govindaraju, N. K., and Crall, C. (2008). Challenges and Opportunities in Many-Core Computing. *Proceedings of the IEEE*, 96(5):808–815.

Mariani, G., Palermo, G., Zaccaria, V., and Silvano, C. (2013). ARTE: An Application-specific Run-Time managEment framework for multi-cores based on queuing models. *Elsevier Parallel Computing*, 39(9):504–519.

Martins, A. L. d. M., da Silva, A. H. L., Rahmani, A. M., Dutt, N., and Moraes, F. G. (2019). Hierarchical adaptive Multi-objective resource management for many-core systems. *Elsevier Journal of Systems Architecture*, 97:416–427.

Modarressi, M., Asadinia, M., and Sarbazi-Azad, H. (2013). Using task migration to improve non-contiguous processor allocation in NoC-based CMPs. *Elsevier Journal of Systems Architecture*, 59(7):468–481.

Moraes, F. G., Calazans, N. L. V., Mello, A. V., Möller, L. H., and Ost, L. C. (2004). HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip. *Integration, the VLSI journal*, 38(1):69–93.

Ng, J., Wang, X., Singh, A. K., and Mak, T. (2016). Defragmentation for Efficient Runtime Resource Management in NoC-Based Many-Core Systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(11):3359–3372.

Olsen, D. and Anagnostopoulos, I. (2017). Performance-aware resource management of multi-threaded applications on many-core systems. In *Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI)*, pages 119–124.

Pathania, A., Venkataramani, V., Shafique, M., Mitra, T., and Henkel, J. (2017). Defragmentation of Tasks in Many-Core Architecture. *ACM Transactions on Architecture and Code Optimization*, 14(1):1–21.

Patterson, D. A. and Waterman, A. (2017). *The RISC-V reader: an open architecture atlas*. Strawberry Canion. 172 pages.

Peckham, O. (2020). Esperanto Unveils ML Chip with Nearly 1,100 RISC-V Cores. Source: https://www.hpcwire.com/2020/12/08/esperanto-unveils-ml-chip-with-nearly-1100-risc-v-cores. 2021.

Petrisko, D., Gilani, F., Wyse, M., Jung, D. C., Davidson, S., Gao, P., Zhao, C., Azad, Z., Canakci, S., Veluri, B., Guarino, T., Joshi, A., Oskin, M., and Taylor, M. B. (2020). BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro*, 40(4):93–102.

Pressman, R. S. and Maxim, B. R. (2019). *Software Engineering: a practitioner's approach*. McGraw-Hill Education, 9th edition. 704 pages.

Rahmani, A. M., Donyanavard, B., Mück, T., Moazzemi, K., Jantsch, A., Mutlu, O., and Dutt, N. (2018a). SPECTR: Formal supervisory control and coordination for many-core systems resource management. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 169–183.

Rahmani, A.-M., Haghbayan, M.-H., Kanduri, A., Weldezion, A. Y., Liljeberg, P., Plosila, J., Jantsch, A., and Tenhunen, H. (2015). Dynamic power management for many-core platforms in the dark silicon era: A multi-objective control approach. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 219–224.

Rahmani, A. M., Haghbayan, M. H., Miele, A., Liljeberg, P., Jantsch, A., and Tenhunen, H. (2017). Reliability-Aware Runtime Power Management for Many-Core Systems in the Dark Silicon Era. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(2):427–440.

Rahmani, A. M., Jantsch, A., and Dutt, N. (2018b). HDGM: Hierarchical Dynamic Goal Management for Many-Core Resource Allocation. *IEEE Embedded Systems Letters*, 10(3):61–64.

Rauber, T. and Rünger, G. (2013). *Parallel Programming for Multicore and Cluster Systems*. Springer, 2nd edition. 516 pages.

Reid, J. F. and Caelli, W. J. (2005). DRM, trusted computing and operating system architecture. In *Proceedings of the ACM Conferences in Research and Practice in Information Technology Series (CRPIT)*, pages 127–136.

Renau, J., Fraguela, B., Tuck, J., Liu, W., Prvulovic, M., Cezze, L., Sarangi, S., Sac, P., Strauss, K., and Montesinos, P. (2005). SESC simulator. Source: http://sesc.sourceforge.net. 2021.

Rhoads, S. (2001). Plasma CPU. Source: http://plasmacpu.no-ip.org/author.htm. 2021.

Ruaro, M., Caimi, L. L., Fochi, V., and Moraes, F. G. (2019a). Memphis: a framework for heterogeneous many-core SoCs generation and validation. *Springer Design Automation for Embedded Systems*, 23(3-4):103–122.

Ruaro, M., Carara, E. A., and Moraes, F. G. (2014). Tool-set for NoC-based MPSoC debugging - A protocol view perspective. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2531–2534.

Ruaro, M., Jantsch, A., and Moraes, F. G. (2019b). Self-adaptive QoS management of computation and communication resources in many-core SOCs. *ACM Transactions on Embedded Computing Systems*, 18(4):1–21.

Ruaro, M., Lazzarotto, F. B., Marcon, C. A., and Moraes, F. G. (2016). DMNI: A Specialized Network Interface for NoC-based MPSoCs. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1202–1205.

Ruaro, M., Santana, A., Jantsch, A., and Moraes, F. G. (2021). Modular and Distributed Management of Manycore SoCs. *ACM Transactions on Computer Systems*, 38(1-2):1–16.

Schmaus, F., Maier, S., Langer, T., Rabenstein, J., Honig, T., Schroder-Preikschat, W., Bauer, L., and Henkel, J. (2020). System software for resource arbitration on future many-* architectures. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 967–975.

Shalf, J., Bashor, J., Patterson, D., Asanovic, K., Yelick, K., Keutzer, K., and Mattson, T. (2009). The Manycore Revolution: Will HPC Lead or Follow. *SciDAC Review*, 14:40–49.

Shamsa, E., Kanduri, A., Rahmani, A. M., Liljeberg, P., Jantsch, A., and Dutt, N. (2019). Goal-Driven Autonomy for Efficient On-chip Resource Management: Transforming Objectives to Goals. In *Proceedings of the IEEE Design, Automation Test in Europe Conference (DATE)*, pages 1397–1402.

Singh, A. K., Shafique, M., Kumar, A., and Henkel, J. (2013). Mapping on multi/many-core systems: Survey of current and emerging trends. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 1–10.

Tanenbaum, A. S. and Bos, H. (2014). *Modern Operating Systems*. Pearson, 4th edition. 1136 pages.

Tsoutsouras, V., Xydis, S., and Soudris, D. (2018). Application-arrival rate aware distributed run-time resource management for many-core computing platforms. *IEEE Transactions on Multi-Scale Computing Systems*, 4(3):285–298.

Turley, J. (2021). Wait, What? MIPS Becomes RISC-V. Source: https://www.eejournal.com/article/wait-what-mips-becomes-risc-v. 2021.

Wachter, E., Caimi, L. L., Fochi, V., Munhoz, D., and Moraes, F. G. (2017). BrNoC: A broadcast NoC for control messages in many-core systems. *Microelectronics Journal*, 68:69–77.

Wang, H., Ma, J., Tan, S. X., Zhang, C., Tang, H., Huang, K., and Zhang, Z. (2016). Hierarchical dynamic thermal management method for high-performance many-core microprocessors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(1):1–21.

Wang, X. and Mak, T. (2013). On self-tuning networks-on-chip for dynamic network-flow dominance adaptation. *ACM Transactions on Embedded Computing Systems*, 13(2s):1–21.

Waterman, A., Lee, Y., Patterson, D., and Asanović, K. (2016a). The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.1. Technical Report UCB/EECS-2016-118, University of California, Berkeley. 133 pages.

Waterman, A., Lee, Y., Rimas, A., Patterson, D. A., and Asanović, K. (2016b). The RISC-V Instruction Set Manual, Volume II: Privileged Architecture Version 1.9. Technical Report UCB/EECS-2016-129, University of California, Berkeley. 89 pages.

Woo, D. H. and Lee, H. H. S. (2008). Extending Amdahl's law for energy-efficient computing in the many-core era. *IEEE Computer*, 41(12):24–31.

Yu, L., Schach, S. R., Chen, K., and Offutt, J. (2004). Categorization of Common Coupling and Its Application to the Maintainability of the Linux Kernel. *IEEE Transactions on Software Engineering*, 30(10):694–706.

Zaruba, F. and Benini, L. (2019). The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640.

Zhu, S. and Ma, K.-K. (2000). A new diamond search algorithm for fast block-matching motion estimation. *IEEE Transactions on Image Processing*, 9(2):287–290.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br