



National Technical University of Athens
School of Electrical & Computer Engineering
Division of Communication, Electronic and Information Engineering

Run-time resource management and application customization for many-core embedded platforms

Ph.D. Thesis

Iraklis Anagnostopoulos

Athens, January 2014

National Technical University of Athens
School of Electrical & Computer Engineering
Division of Communication, Electronic and
Information Engineering
Microprocessors and Digital Systems Lab

Run-time resource management and application customization for many-core embedded platforms

Ph.D. Thesis
of
Iraklis Anagnostopoulos

Submitted in School of Electrical & Computer Engineering of
National Technical University of Athens

National Technical University of Athens
School of Electrical & Computer Engineering
Division of Communication, Electronic and
Information Engineering
Microprocessors and Digital Systems Lab

Run-time resource management and application customization for many-core embedded platforms

Supervising committee

Dimitrios Soudris Ass. Professor N.T.U.A.	Kiamal Pekmestzi Professor N.T.U.A.	George Economakos Ass. Professor N.T.U.A.
--	--	--

Advisory committee

Dimitrios Soudris Ass. Professor N.T.U.A.	Kiamal Pekmestzi Professor N.T.U.A.	George Economakos Ass. Professor N.T.U.A.
--	--	--

Nectarios Koziris Professor N.T.U.A.	Ioannis Papaefstathiou Ass. Professor T.U.C.
---	---

Dimitris Gizopoulos Ass. Professor N.K.U.A.	Axel Jantsch Professor KTH
--	-------------------------------

©2014, Iraklis N. Anagnostopoulos
Electrical & Computer Engineer N.T.U.A.

This Ph.D. Thesis was cofinanced by the E.C. funded projects FP7-248716 2PARMA, FP7-215244 MOSART and ENIAC-2010-1 TOISE. Also, part of this Ph.D. Thesis was partially supported by Hellenic Funds and by the European Regional Development Fund (ERDF) under the Hellenic National Strategic Reference Framework (NSRF) 2007-2013, for the project “Next Generation Millimeter Wave Backhaul Radio”.

Abstract

In this Ph.D. Thesis, we present (i) memory management middleware acceleration and customization methodologies for applying customized dynamic memory managers (allocators) and (ii) frameworks for distributed run-time resource management on many-core platforms. Firstly, the customization is achieved by applying, on the middleware level, custom microcoded memory allocators. Secondly, the run-time resource management on the platform is achieved by using cores in different roles and by applying a distributed on-chip communication scheme. The proposed methodologies showed that the microcode approach is a good alternative to overcome the performance-flexibility dilemma, offering a programmable and flexible solution for accelerating a wide range of applications. Thus, we adopt the microcoded approach to address memory management issues on Distributed Shared Memory (DSM) many-core embedded platforms, aiming for hardware performance but maintaining the flexibility of programs. Also, the developed framework provides a flexible solution in the run-time mapping problem offering different levels of platform utilization according to application's needs and without a central point of failure.

Concerning microcoded memory management services, experimental results show that the gain, of the proposed approaches for designing customized microcoded memory managers, was approximately $7\times$ for served allocation requests with a small increase of approximately 14% to average energy consumption per allocation. The run-time resource management framework adapts to application's needs and application's execution restrictions by using the matching factor parameter and produces on average 21% and 10% better on-chip communication cost for homogeneous and heterogeneous platforms respectively. Last, concerning the malleable parallel applications, the developed framework has on average 70% less messages, 64% smaller message size and 20% application speed-up gain.

Contents

Abstract	9
List of Figures	15
List of Tables	19
1 Introduction	21
1.1 Memory management acceleration and customization on embedded systems	21
1.2 Convergence general and embedded computing	23
1.2.1 Heterogeneous computing systems	24
1.3 Dynamic applications	25
1.4 Thesis overview	28
2 Contribution	31
2.1 Objectives and Contributions	31
2.2 Related Work	35
2.2.1 Memory management middleware acceleration	35
2.2.2 Microcode-accelerated distributed dynamic memory management	36
2.2.3 Microcode-accelerated distributed dynamic memory management	38
3 Memory management middleware acceleration and customization	43
3.1 Introduction	43
3.2 Platform Used	44
3.2.1 Cache coherency and memory consistency	47
3.3 Memory management middleware acceleration	48
3.3.1 Custom Microcoded Dynamic Memory Management	48
3.3.1.1 Application mapping to platform's cores	50

3.3.1.2	Application dependent DMM customization	51
3.3.1.3	Platform dependent DMM customization	52
3.3.1.3.1	DMM Microcode Translation . .	52
3.3.1.3.2	Customization according to memory distribution	53
3.3.1.4	Evaluation of Custom Microcoded Dynamic Memory Management	56
3.3.2	Conclusions	63
3.4	Microcode-accelerated distributed dynamic memory management	64
3.4.1	Heap SPace Map	65
3.4.2	MAD-DMM implementation	66
3.4.3	MAD-DMM evaluation	71
3.5	Power-aware DMM on Many-core Platforms utilizing DVFS	74
3.5.1	Integration of DVFS mechanisms to DMM library	76
3.5.1.1	Monitor mechanisms	78
3.5.1.2	DVFS decision mechanism	82
3.5.1.3	Integrated DVFS interfaces	83
3.5.2	Experimental set-up	84
3.5.2.1	DVFS overview	84
3.5.2.2	Benchmarks and execution model	86
3.5.2.3	Selected DM managers	87
3.5.3	Evaluation	88
3.5.3.1	Power consumption and heap fragmentation of the selected DM managers . . .	88
3.5.3.2	Power consumptions	91
3.5.3.3	Performance overhead	95
3.5.3.4	Power consumption and performance overhead trade-off	97
3.5.4	Conclusions	99
4	Distributed Run-time resource management	101
4.1	Introduction	101
4.2	Divide and Conquer based Distributed Run-time Mapping on many-core platforms	104
4.2.1	Proposed run-time mapping methodology framework	106
4.2.1.1	Definitions	108

4.2.1.2	Homogeneous Platform	109
4.2.1.3	Heterogeneous Platform	111
4.2.2	Experimental results	114
4.2.3	Conclusions	118
4.3	Distributed run-time resource management for malleable applications	119
4.3.1	Methodology Framework	120
4.3.1.1	Definitions	122
4.3.1.2	Communication Scheme	123
4.3.1.3	Gain calculation	126
4.3.1.4	Self-optimization process	126
4.3.2	Experimental Results	128
4.3.2.1	Evaluation on C simulator	128
4.3.3	Evaluation on Intel SCC platform	130
4.3.4	Conclusions	135
5	High-level customization framework for resource management on NoC architectures	139
5.1	Introduction	139
5.2	NoC framework overview for resource management . . .	141
5.2.1	Resource management in regular NoC design . . .	144
5.2.2	Resource management in irregular NoC design . .	147
5.2.2.1	Application partitioning	148
5.2.2.2	Clustering	150
5.2.2.3	Routing Table Generation	151
5.2.3	Buffer Sizing	153
5.3	Evaluation	154
5.3.0.1	NoC's throughput	156
5.3.0.2	NoC's average dealy	157
5.3.1	Buffer's power consumption	158
5.3.2	Conclusions	159
6	Conclusions	161
6.1	Summary of Ph.D. Thesis	161
6.2	Perspectives and Future Extensions	163
	Publications	167
	Bibliography	171

List of Figures

1.1	Gap between hardware and software development for embedded systems	22
1.2	Performance gap between processors and memory [42]	22
1.3	HSA solution stack [6]	25
1.4	Data growth vs. Moore’s Law trends in the last 5 years [4]	26
1.5	Traffic management requirements for mobile broadband applications [3]	27
1.6	Thesis overview	29
2.1	Flow of the ADAM algorithm [9]	39
2.2	General task migration mechanism [63]	40
3.1	16-node mesh McNoC; Processor-Memory (PM) node [29]	45
3.2	The DMC architecture and synthesis results [29]	46
3.3	DSM organization and V2P translation	46
3.4	Framework for supporting custom microcoded DMM on McNoC platforms with distributed memories	49
3.5	MTh-DMM Explorer tool [89]	51
3.6	Code translation example. From C++ to microcode. First Fit algorithm.	54
3.7	NoC memory distribution-aware DMM customization example: a) Selected topology and mapped cores, b) Thread to memory priority table, c) SLL structure. d) Microcoded topology aware function templates.	57
3.8	The used multi-threaded application. Squares define the different threads which communicate asynchronously through asynchronous FIFO queues	58

3.9	a) Topology used for evaluation 2×2 NoC with 3 processing nodes with local memory (LM) and 1 memory node. b) Pure Distributed Memory c) Centralized single Heap d) Distributed multiple-Heap with global Heap e) Memory distribution-aware multiple-Heap with global Heap	61
3.10	Performance comparison and DMM event distribution.	63
3.11	Average cycles and energy consumption per DMM event.	64
3.12	The implementation of Heap Space Map on top of V2P translation service.	66
3.13	Overview of the MAD-DMM distributed allocation allocation procedure.	67
3.14	Abstract presentation of MAD-DMM interfaces	69
3.15	Communication between nodes using message passing instructions. The src node triggers the malloc microcode in the dst node and after the completion of the function, the dst node returns the block address back to the src DMC. Last, the block address is returned to the C application	70
3.16	Performance comparison of MAD-DMM with [10] and [57] under a pure private heap.	73
3.17	Average cycles per malloc/free call and number of microcode instructions needed for intercommunication for various NoC sizes. (a) FAST, (b) Matrix, (c) Gaussian and (d) Integral.	74
3.18	Power-aware dynamic memory management flow.	76
3.19	Monitoring process and DVFS decision mechanism integrated to (de)allocation process	78
3.20	Abstract view of monitoring mechanism for gathering allocator's accesses. The total number of accesses is propagated to the DVFS decision mechanism (Section 3.5.1.2).	80
3.21	GRLS Overview [26]	85
3.22	Power management architecture of the McNoC platform [26]	86
3.23	Normalized power consumption comparison of the selected DM managers without any monitoring or DVFS changing mechanism.	88
3.24	Heap fragmentation of the selected DM managers.	89
3.25	Normalized power consumption for DM manager 1 compared with the integration of DVFS mechanisms and different window sizes (WS).	90

3.26	Normalized power consumption for DM manager 2 compared with the integration of DVFS mechanisms and different window sizes (WS).	91
3.27	Normalized power consumption for DM 3 manager compared with the integration of DVFS mechanisms and different window sizes (WS).	92
3.28	Normalized power consumption for DM manager 4 compared with the integration of DVFS mechanisms and different window sizes (WS).	92
3.29	Normalized power consumption for DM manager 5 compared with the integration of DVFS mechanisms and different window sizes (WS).	93
3.30	Number of accesses in DM manager's linked lists for FAST benchmark in comparison with DM manager's decisions for frequency changes for $WS = 4$ and $WS = 8$	94
3.31	Number of accesses in DM manager's linked lists for FAST benchmark in comparison with DM manager's decisions for clock frequency changes for $WS = 16$ and $WS = 32$	95
3.32	$P(e_i)$ values, for all DM Managers, in comparison to normalized power consumption (compared to no DVFS), performance overhead and window size (WS).	98
4.1	Illustration of the mapping problem [46]	101
4.2	Divide and Conquer in computer science.	104
4.3	Divide and Conquer on many-core platform	106
4.4	Flow of our D&C methodology.	107
4.5	Communication Cost comparison in homogeneous platforms to ADAM [9] and design-time mapping [46].	114
4.6	Mapping computational effort in homogeneous platforms in comparison with ADAM [9] and design-time mapping [46].	115
4.7	Communication cost comparison for the five selected applications with ADAM [9] and design-time mapping [46].	116
4.8	Run-time mapping scenarios on an heterogeneous many-core platform compared with ADAM [9]	117
4.9	Overall flow of the proposed methodology.	121
4.10	Example of the communication scheme.	125
4.11	More fair resource allocation through self-optimization process [50]	128
4.12	Overview of the Inter SCC Platform [81]	131

4.13	Memory architecture of the SCC processor [58]	132
4.14	Symmetric name space model for the MPB as designed for RCCE library [58]	133
4.15	Total number of messages sent for intercommunication by all nodes for various applications compared with DistRM [50].	134
4.16	Total size of sent messages in <i>bytes</i> compared with DistRM [50].	135
4.17	Average application speed-up using the speed-up function presented in [50].	136
4.18	Computational effort comparison with DistRM [50].	136
5.1	Examples of irregular NoC topologies. Each router (r) can serve more than one Processing Element (PE) by having multiple ports.	142
5.2	Proposed high-level simulation framework for automatic generation of application-specific NoC architectures	143
5.3	VOPD (a) application task graph , (b) partitioning using <i>Multilevel – KL</i> algorithm [41] and (c) clustering.	149
5.4	Configuration file structure	151
5.5	Example of the generated topology using the configuration file presented in Figure 5.4	152
5.6	Average, best and worst gains ,in terms of cycles, for packet delivery time while employing priorities.	155
5.7	NoC’s throughput for the selected applications	157
5.8	NoC’s average delay for the selected applications	158
5.9	Normalized power consumption for both regular and irregular design path with the addition of the proposed buffer sizing algorithm	159
6.1	Trends of increased complexity in Systems-on-Chip targeting the market of consumer mobile devices [74]	164

List of Tables

3.1	Description of the Selected DMM Configurations	59
3.2	Power management commands and description [26]	84
3.3	Performance overhead in terms of cycles	96
4.1	Utilization of platform's resources	118
4.2	Comparison of the proposed technique to the DistRM [50] in the C simulator.	129

Chapter 1

Introduction

1.1 Memory management acceleration and customization on embedded systems

Future integrated systems will contain billion of transistors [73], composing tens to hundreds of IP cores. Modern embedded platforms take advantage of this manufacturing technology advancement and are moving from Multi-Processor Systems-on-Chip (MPSoC) towards Many-Core architectures employing high numbers of processing cores. Intel has already created platforms with 80 and 48 general purpose processing cores [45, 72, 85], while Networks-on-Chip (NoC) are already supported by the industry (such as the *Æ*thereal NoC [38] from NXP and the STNoC [80] from STMicroelectronics). The industrial vision goes as far as thousand core chips [20]. Also, the development of such many-core architectures is driven by the development of highly parallel/multi-threaded demanding applications.

The gap between hardware and software development for embedded systems is depicted in Figure 1.1. We see that the needs for software double every 10 months while the performance of hardware platforms is doubling every 24 months. Despite the fact that the productivity of hardware designers improved the recent years, additional functions and services provided only by software. The red arrow in Figure 1.1 shows the size of this gap.

Memory is an important contributor to the performance and power consumption of embedded systems. As the number of on-chip cores increased, the embedded memory content also increased from 20% ten

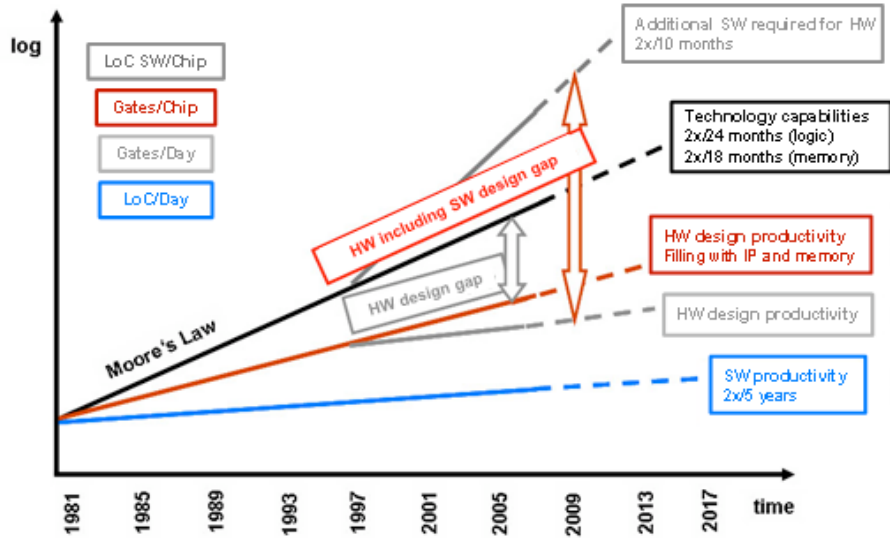


Figure 1.1: Gap between hardware and software development for embedded systems

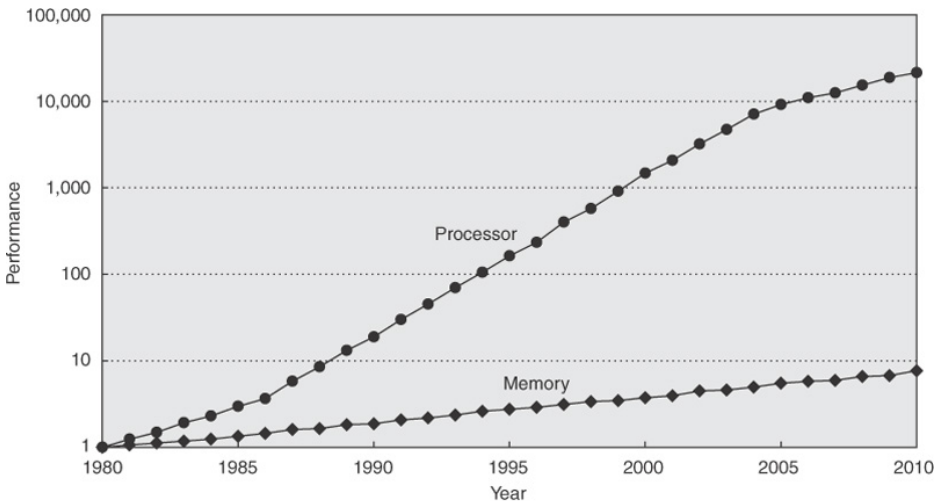


Figure 1.2: Performance gap between processors and memory [42]

years ago to 85% of the chip area today and will continue to increase in the future. Memories are preferably distributed for medium and large scale system sizes, since centralized memory has already become the bottleneck of performance, power and cost. Traditional memory optimization uses compile-time information and focuses on static allocation in respect to memory hierarchy [25]. For modern dynamic applications, using many-core architectural templates, this is no longer possible since there is a lot of memory unpredictability, which cannot be captured by source code analysis alone and the increased dynamism in data storage leads to unexpected memory footprint variations, unknown at design time. Insufficient memory management leads to overall performance degradation, big memory footprint and increased power consumption as shown in Figure 1.2.

1.2 Convergence general and embedded computing

Computing Systems have a tremendous impact on everyday life in all domains, from the internet to consumer electronics, transportation to manufacturing, medicine, energy, and scientific computing. Also, the traditional perspective of single-purpose/single-core embedded systems and devices is rapidly changing as increased computing performance and functionality is being added by industry leaders. Convergence, both in hardware and software platforms, is rampant throughout the industry with desktop processors and embedded processors merging and applications moving from local systems to commodity cloud platforms and the web. For example, there was a time when mobile phones were invented just for the ease of communication and the hardware requirements were minimal. Nowadays, mobile phones accommodate up to four cores, with high definition cameras and hardware accelerators and they are available to consumers at an affordable cost. Samsung Galaxy S4, HTC One X, Huawei Ascend P2 etc. are considered among the best mobile phones in terms of performance, with computing capacity bigger than yesterday's high-performance systems.

The computing systems industry is likewise experiencing a range of disruptive trends. Dr. Doug Burger, Director of Client and Cloud Applications at Microsoft Research, in his HiPEAC2013 keynote stated *“there is more uncertainty in the field of computing now than at any time in the past 40*

years; computing needs a revolution that will leave no part of it untouched". We can see companies working as hardware vendors for computing systems (e.g., Apple, Samsung, etc.) expanding to higher levels producing software products targeting and claiming a bigger market share (e.g., iOS, Samsung Store, etc.) Similarly, classic software development companies (e.g., Microsoft, Google, Canonical, etc.) are moving towards to mobile and embedded computing markets (e.g., MS Surface, Kinect, Google Glass, and Ubuntu Tablet/Phone). These trends are putting increased pressure on companies to effectively integrate software and hardware components and the main result of this pressure is system heterogeneity and the development and usage of custom hardware accelerators (e.g., GPU accelerators).

1.2.1 Heterogeneous computing systems

It is evident that the desktop PC is no longer driving the computing industry, since smart phones and tablets are attracting more and more consumers than PCs, and traditional applications, such as communication, sharing images, videos and chatting, are being offered by mobile systems that can be used anywhere, anytime. The use of "cloud" resources pushes this trend even further. As a result, the market sees most growth in the cloud and mobile devices, which pushes hardware design resources in those directions. Thus, modern embedded systems are including programmable custom accelerators to improve performance and energy efficiency. However, these accelerators even though prevail as the key feature in performance improvement; they are still not fully exploited due to the lack of software interfaces and programming paradigms (middleware and its corresponding APIs).

So, heterogeneity and hardware accelerators are the dominant characteristics of modern computing systems. In order to face the problem of the proper hardware exploitation, many industrial leaders grouped and founded the Heterogeneous System Architecture (HSA) Foundation. HSA is a not-for-profit consortium of SoC IP vendors, OEMs, academia, SoC vendors, OSVs and ISVs whose challenging the normal of how whole system architecture is structured for combing CPUs, GPUs, DSPs, and other accelerators to bring about forward progress in computing foundation to make it dramatically easier to program heterogeneous parallel devices [5]. HSA Foundation members are building a hetero-

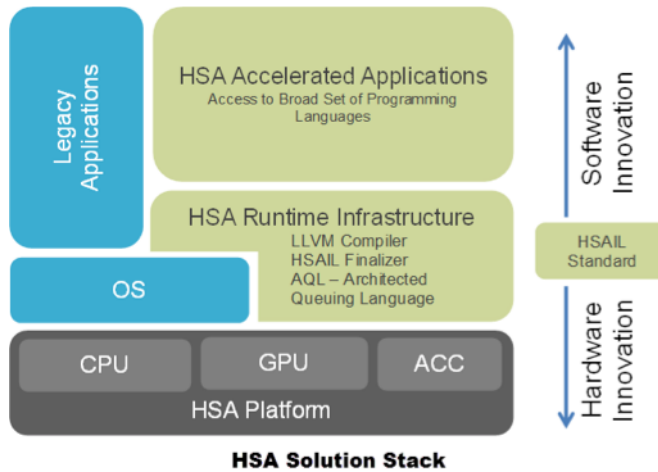


Figure 1.3: HSA solution stack [6]

geneous compute software ecosystem which is rooted on open royalty free industry standards. Members of this foundation are world leading industrial companies such as AMD, ARM, Samsung, Qualcomm etc. To enable easier programming, HSA allows developers to program at a higher abstraction level using mainstream programming languages, with the addition of libraries targeting HSA. Figure 1.3 shows a high-level view of the HSA Solution Stack. The key to enabling one language for heterogeneous core programming is to have an intermediate runtime layer that abstracts hardware specifics away from the developer, leaving the hardware-specific coding to be done once by the hardware vendor or IP provider.

1.3 Dynamic applications

These hardware changes are also driven by application changes. All facets of society are generating increasing amounts of data confirming the term Big Data for modern applications: multimedia (streaming videos and images), medical (in-silico drug development, population databases and records), security (financial transactions) and cloud applications (search histories, weather prediction). Figure 1.4 shows the comparison of data growth with Moore's law. It is clear that data

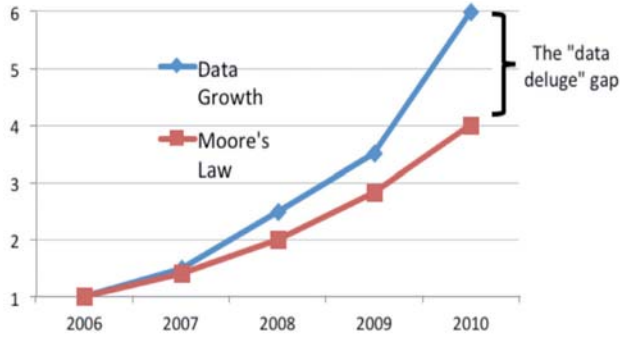


Figure 1.4: Data growth vs. Moore's Law trends in the last 5 years [4]

is on a higher exponential growth than computation capacity. This unprecedented growth of data is forcing industry to reevaluate communication and memory middleware services in computing platforms using hardware accelerators for improving its performance and energy efficiency.

The next generation of embedded systems will be dominated by mobile and smart devices that will be capable of offering a wide range of communications services and applications anywhere anytime. The integration of multimedia and network applications such H.264/AVC/SVC, JPEG 2000 and WiMax, creates complex and dynamic applications that have significant resource real-time requirements (Figure 1.5). The main characteristic of such applications is the increased need for data storage and transfer for efficient memory management. Resource management is a key technology for the successful use of such embedded platforms. The run-time resource management paradigm has become prominent recently because it can deal with the run-time dynamics of applications and platforms. Thus, the efficient run-time application resource management enables the efficient usage of the platform resources.

The memory management can be achieved in several levels of abstraction: management of application's virtual/dynamic memory, unlike to the management of physical memory (the application source code is platform independent). Also, the memory management can be applied with greater detail/accuracy in the hierarchy/architecture memory level. The inefficient memory management leads to reduced per-



Figure 1.5: Traffic management requirements for mobile broadband applications [3]

formance (mainly due to the performance gap between processors and memory modules), to increased memory demands and high energy consumption.

Adopting the many-core architectural template, applications running on yesterday's high-performance computers start to appear in embedded systems. From high-intensive applications (video decoding, games etc.) to internet-based ones, new applications take over the embedded world. However, the application developers are not willing to change the way of developing applications' source code leading to insufficient memory usage. The memory restrictions raised by embedded systems are not taken into consideration resulting in performance degradation. Keeping the same APIs and application source code while maximizing memory utilization in embedded systems appears to be the challenge for the designers.

1.4 Thesis overview

The design challenge in the above cases is the difficulty of achieving cooperation between software applications and heterogeneous platforms. This Thesis focuses on the run-time resource management and application customization for many-core embedded platforms. An overview of the developed techniques and frameworks is presented in Figure 1.6. The starting point of the developed methodologies is the application and the many-core platform that the application will be executed on. The proposed techniques have a manyfold aim. They are specially focused on the field of developing hardware accelerated Dynamic Memory Managers (DMM) and run-time resource management. Key point in all techniques is the exploitation of platform's heterogeneity and of any available hardware accelerators. On the field of memory management middleware acceleration and customization, we firstly perform application-based DMM customization followed by the customizations according to the platform heterogeneity. This results in better exploitation of memory locality, offering distributed functionality, and utilization of platform's heterogeneity and hardware accelerators. Middleware proves out to be a solution for keeping software flexibility while taking advantage of hardware acceleration. On the field of run-time resource management, a run-time mapping framework takes into account any platform's heterogeneity and utilizes best the presence of hardware accelerators. Also, following the trends in application domains, a run-time resource management framework for parallel applications is presented while it is also focused on a self-reconfiguration process.

The presented Thesis is organized in six chapters as follows:

- In Chapter 1, a presentation of the industry trends for software and hardware in embedded systems is presented. Also, the abstract Thesis overview is depicted and it is clarified the focus of the developed techniques.
- In Chapter 2, the Thesis contribution is presented and it is analyzed in topics. Also, the solutions for specific problems are pinpointed and last, the contribution and differentiations to other state-of-art approaches is mentioned.
- Chapter 3 introduces the concept of accelerating dynamic mem-

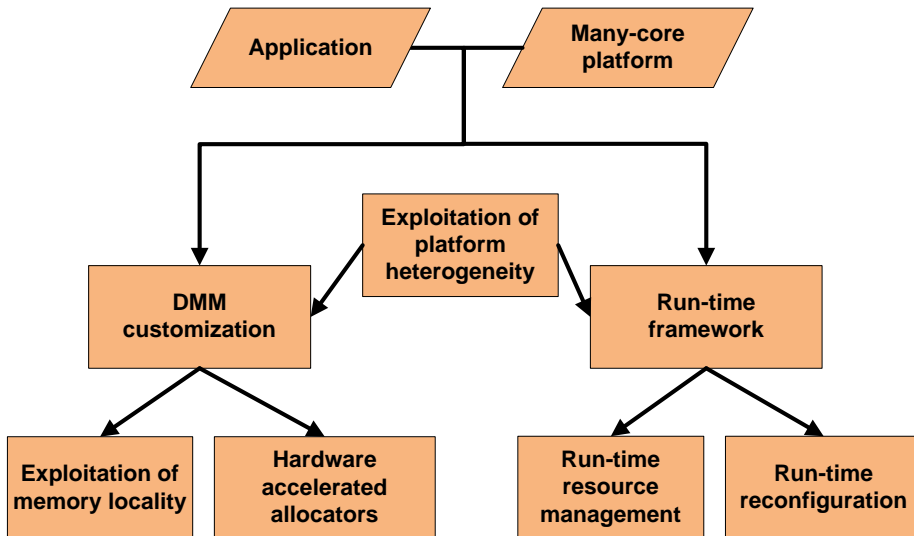


Figure 1.6: Thesis overview

ory management functions in the middleware level. A framework for custom and distributed microcoded Dynamic Memory Management is presented. Last, a framework for coupling the concept of dynamic memory management with DVFS techniques targeting low power consumption is shown.

- Chapter 4 presents the run-time resource management frameworks. Firstly, a distributed run-time mapping framework for heterogeneous platforms is presented. Secondly, an algorithm for run-time resource management, in a distributed way, for malleable parallel applications is shown.
- Chapter 5 presents a high-level customization framework and methodology for resource management on Network-on-Chip (NoC) architectures, both regular and irregular, based on application needs.
- Last, Chapter 6 summarizes the innovative points and findings of this thesis and highlights future extensions.

Chapter 2

Contribution

In this chapter an overview of the research area and problems are presented as well as the contribution of the proposed solutions is mentioned.

2.1 Objectives and Contributions

Our work has a manyfold aim. We focus on the run-time resource management and application customization for many-core embedded platforms. The research work is specially focused on the field of (i) memory management middleware acceleration and customization; (ii) run-time resource management and (iii) application-based platform customization and resource management.

Firstly, we present the memory management acceleration and customization framework. We apply these techniques to efficiently address the problems of:

1. Providing customized dynamic memory allocators on many-core embedded platforms.
2. Exploiting platform's characteristics in order to provide distributed functionality over a Distributed Shared Memory (DSM) environment.
3. Coupling the concept of Dynamic Memory Management (DMM) with DVFS techniques targeting low power consumption on many-core embedded platforms.

We adopt the microcoded approach for supporting custom Dynamic Memory Managers (DMMs), also known as allocators on MPSoC platforms, aiming for hardware performance but maintaining the flexibility of software implementations. In order to guarantee high performance, the proposed DMM services are developed on top of a hardware dual-microcoded controller (DMC) [29] that a) works transparent to the on-chip communication, b) is responsible for handling distributed memory requests and c) mitigates processor's workload.

The main contributions of memory management acceleration and customization framework are:

1. **Microcoded allocator:** We propose a framework for generating microcoded DMM services on top of a hardware dual-microcoded controller (DMC) [29] exploiting platform's DSM characteristics. By uploading microcoded DMM to DMC instruction memory, the processors of the Many-core Network-on-Chip (McNoC) platform stop being responsible for servicing DMM requests or keeping track of the status of the heaps, thus alleviating their load.
2. **Distributed functionality and scalability:** A microcode-accelerated flexible, distributed and scalable allocator, called MAD-DMM, is proposed. MAD-DMM provides distributed functionality over a DSM environment, under microcode implementation, while keeping the standard C-API (`malloc()`/`free()`) thus being transparent to the application. MAD-DMM handles the heap continuous space under a heap space map scheme and all nodes on the many-core platform are seamlessly aware of the heap state. Unlike high-level allocators, information about the allocator metadata is not stored at high-level but at microcode-level as part of a local heap table.
3. **Power monitoring:** A new design strategy is proposed for the the implementation of an efficient high-level methodology of applying transparent monitor and DVFS decision mechanisms into any any C-allocator targeting low power consumption.
4. **Evaluation:** Evaluation of the proposed frameworks is conducted through extensive experimentation and explorative results.

Concerning microcoded DMM services, experimental results show that the proposed approach for designing customized microcoded memory

distribution-aware DMM [10] (a) can serve more DMM events by using all available Heaps of the platform; (b) increases Heap lifetime; (c) is fully configurable and easy to use (offering microcoded templates); (d) achieves better performance exploiting the presence of the DMC for handling distributed memory requests, thus mitigating processor's workload; and (e) has a negligible penalty regarding energy consumption. Specifically, the gain was approximately $7\times$ for served allocation requests with a small increase of approximately 14% to average energy consumption per allocation. Also, the microcode approach is on average 25% faster than the C implementation, enhancing the reason for choosing a hardware controller for handling distributed memory requests. Furthermore, MAD-DMM proved to be on average 25% slower than the microcoded allocator [10] and 10% faster than the high-level one [57], since most of the (de)allocation operations are performed at microcode level and only the high-level heap address manipulation is performed in C. On the other hand, MAD-DMM proved to be more scalable, since purely microcoded allocator needs on average 29% more cycles to serve an event each time the platform increases, whereas MAD-DMM needs approximately 20% more cycles. Last, the proposed framework for coupling DVFS techniques along with DMM showed that by using the proposed method for monitoring and applying DVFS mechanisms, the power consumption concerning heap management was reduced by approximately 37%. In addition, by combining this method with heap fragmentation-aware DMMs, we can achieve low power consumption with low heap fragmentation values.

Secondly, we present distributed run-time resource management frameworks for many-core platforms. Specifically, we (i) adopt the concept of the *Divide & Conquer (D&C)* method in order to perform distributed run-time mapping on both homogeneous and heterogeneous many-core platforms; (ii) couple the concept of distributed computing with parallel applications and present a workload-aware distributed run-time framework for malleable applications running on many-core platforms and (iii) present a high-level customization framework and methodology for resource management on NoC architectures, both regular and irregular. The proposed frameworks are based on the idea of using multiple cores in different roles while, in all case, an on-chip intercommunication scheme ensures decision distribution. The proposed frameworks were evaluated on the experimental many-core platform presented in [29] and on the Intel Single Cloud Chip (SCC) many-core one [45]. Last,

a high-level customization framework and methodology for resource management on NoC architectures, both regular and irregular, based on application needs is presented.

The main contributions of the presented distributed run-time resource management frameworks are:

1. **Support of heterogeneous platforms:** The proposed framework for distributed run-time mapping supports both homogeneous and heterogeneous many-core platforms
2. **Resource utilization:** The proposed distributed run-time frameworks can achieve different levels of platform's resources utilization depending on application's needs in comparison with other state-of-the-art distributed algorithms [9, 50]
3. **Application's needs:** Make sure that the application will get the optimum number of cores avoiding dominating effects
4. **Heterogeneity exploitation :** Takes into account the type of processors best utilizing any platform's heterogeneity while having a small overhead in overall core intercommunication.
5. **Evaluation:** Evaluation of the proposed frameworks is conducted through extensive experimentation and explorative results on experimental [29] and industrial [45] many-core platforms.

Experimental results shows that the proposed D&C based distributed run-time application mapping framework, for both homogeneous and heterogeneous many-core platforms, produces on average 21% and 10% better on-chip communication cost for homogeneous and heterogeneous platforms respectively, compared to other state-of-the-art distributed scheme[9] with almost the same computational effort. The random implemented runtime scenarios showed that the proposed algorithm can have different behavior according to the selected matching factor and resulting to different platform's resources utilization. Also, the proposed distributed run-time manager for malleable applications proved to have 70% less messages, 64% smaller message size and 20% application speed-up gain compared to the DistRM distributed scheme for malleable applications [50]. Also, the presented framework has a small communication overhead, takes into account platform's heterogeneity

and makes sure that the application will maximize its speed-up function.

2.2 Related Work

Increasingly, the current design trend in System-on-Chip devices (SoC) is based on utilizing multiple processors, thus shifting towards the MPSoC design paradigm. This design trend stands for both general-purpose computing architectures [78] and embedded computing systems [8, 43]. To utilize this high number of cores, modern embedded applications are increasingly becoming multi-threaded. Furthermore, multiple use-case scenarios and increased interaction with the environment expose the high dynamic behavior of these systems.

2.2.1 Memory management middleware acceleration

There are three ways to offer memory management services on Distributed Shared Memory (DSM) platforms. Historically, software-only solutions are the current practice, being flexible but consuming many processor cycles, limiting system performance. Extensive research has been conducted for general-purpose dynamic memory management, which target either the single processor [87, 88], or the multi-processor domain [16, 48, 54, 55, 91]. However, the amount of work that the DMMs perform is not always the same, it changes at run-time since the state of the heap and the number of applications accessing it varies, and it also depends on the type of calls and their parameters. Developing dynamic multi-threaded applications, using worst-case estimates for managing memory in a static manner, would impose severe overheads in memory footprint and power consumption. To avoid such type of costly over-estimations, developers are motivated to efficiently utilize dynamic memory.

Dedicated hardware solutions can achieve high performance, but any small change in functionality leads to re-design of the entire hardware module. A memory allocator which favors cache locality on specific SMP systems is proposed in [71]. Authors in [12] study heap management in the Cell processor, a relevant hardware architecture, but they do not

handle shared memory; instead of this, the processing units have to handle their own, dedicated memory and they communicate with the system through explicit DMA calls, a limitation posed by the individual hardware platform.

Microcode approach is a good alternative to overcome the performance-flexibility dilemma, offering a programmable and flexible solution to accelerate a wide range of applications [86]. Thus, *we adopt the microcoded approach to address DSM issues on McNoCs, aiming for hardware performance but maintaining the flexibility of programs.* The microcode approach has been used in previous multiprocessor systems to solve DSM related memory management issues. The Alewife [7] machine addresses the problem of providing a single address space machine with integrated message passing mechanism. However, it is a dedicated hardware solution and also does not support virtual memory. Both FLASH [52] and Typhoon [67] use a programmable co-processor for supporting flexible cache coherence policy and communication protocol. However, the FLASH and the Typhoon host only one programmable coprocessor to deal with requests from the network and the CPU. If two or more requests come concurrently, only one can compete to be handled while the others have to be delayed, resulting in contention delay. Furthermore, the FLASH and the Typhoon organize memory banks to form a cache-coherent shared memory. Memory accesses are handled by the programmable coprocessor. However, in comparison with the dedicated hardware solution, the local processor spends much more time in accessing data even only used by itself. In our memory organization, the memory is partitioned into the private part and a shared one. The private memory access is local and fast so as to improve performance. The SMTp [28] exploits SMT in conjunction with a standard integrated memory controller to enable a coherence protocol thread used to support DSM multiprocessors. The protocol programmability is offered by a system thread context rather than an extra programmable coprocessor.

2.2.2 Microcode-accelerated distributed dynamic memory management

Traditional optimizations use compile-time, manifest information and have focused on static allocation and how to synthesize memory hi-

erarchies for SoCs [25]. For modern dynamic applications this is no longer possible, as the dynamicity of the behavior due to the input dynamics cannot be captured by source code analysis alone. As presented in [82], dynamic memory management for MapReduce [33] algorithms plays an essential role to overall system performance. More specifically, shared memory MapReduce performance is sensitive to memory allocation pressure among others. This is the reason why the memory allocator limits scalability [92]. Furthermore, static allocation of memory leads to an inefficient memory utilization [11]. Thus, the application behavior and memory requirements significantly vary during run-time. The dynamic memory management (DMM) is a critical component in NoCs, since it often forms the main performance, and scalability bottleneck of multi-threaded applications [16]. Also, it greatly affects the energy and memory consumption of the overall system [11]. Extensive research has been conducted for general purpose dynamic memory allocators targeting either the single processor or the multiprocessor domain [16, 88]. General-purpose, scalable allocators have been suggested for multi-core systems [69], but they focus mostly on synchronization issues and do not study distributed memory schemes. In the field of high-performance computing, global address spacing is considered important and distributed solutions for dynamic memory management have been proposed [56], but this work is aimed on cluster networks, not exploiting the locality and latency benefits of an NoC. Hardware accelerators for dynamic memory management have been proposed by many researchers [27, 88]. A hardware memory management unit (SoCDMMU), responsible for the dynamic allocation and de-allocation of memory is presented in [75]. However, this is a centralized unit and could be a potential bottleneck in McNoCs. Furthermore, SoCDMMU is able to allocate only complete global memory pages and the management of the data (de)allocation of the local (or private) memories is left out to the processors. A hardware MMU (HwMMU) offering dynamic allocation of data on the DSM space of an NoC is proposed in [59]. HwMMU supports dynamic allocation and de-allocation of shared memory with a granularity of complete memory pages, supported by new API calls.

2.2.3 Microcode-accelerated distributed dynamic memory management

The authors of [65] present a mapping and scheduling strategy for hard real-time embedded systems, which communicate over a shared medium (i.e., bus) aiming at minimizing the system modification cost. A run-time application mapping onto homogeneous NoC platforms with multiple voltage levels is presented in [30]. That technique consists of a region selection algorithm and a heuristic for run-time application mapping. Broersma et al. [21] propose the MinWeight algorithm for solving the minimum weight processor assignment problem but only for task graphs with maximum degree at most two. Smit et al. in [79] extend the aforementioned algorithm by solving the problem of run-time task assignment on heterogeneous processors with task graphs restricted to a small number of vertices or a large number of vertices with degree of no more than two [21]. A unified single-objective algorithm, called UMARS, couples path selection, mapping of cores and TDMA time-slot allocation, such that the network required to meet the constraints of the application is minimized [39].

Faruque et al. [9] present a runtime application mapping in a distributed manner using agents targeting for adaptive NoC-based heterogeneous multi-processor systems. The main idea is that in order to achieve the distributed computation of the mapping, the platform is partitioned in virtual clusters and computation of the mapping on each cluster is performed individually. More specifically, a cluster is a subset of the set of tiles of the NoC. Its boundaries are not set and may change at any time, including more tiles, or excluding previously owned tiles. One of the cluster's tiles is selected to act as the cluster agent. An agent is a computational entity which acts on behalf of others. The cluster agent specifically, is an agent that is responsible for mapping operations within its cluster (Figure 2.1).

Authors claim that a centralized run-time resource management may bear a series of problems such as single point of failure and large volume of monitoring-traffic. However, Nollet et al. [63] present a centralized runtime resource management scheme that is able to efficiently manage a NoC containing fine grain reconfigurable hardware tiles and two task migration algorithms. The resource management heuristic consists of a basic algorithm completed with reconfigurable add-ons. The

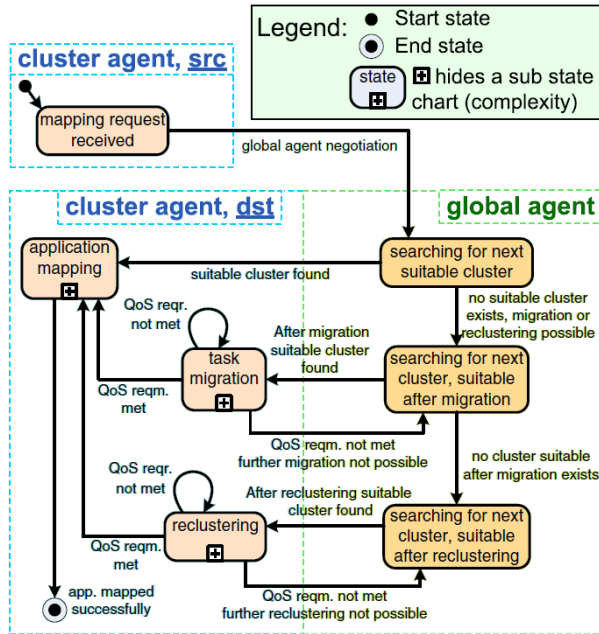


Figure 2.1: Flow of the ADAM algorithm [9]

basic heuristic contains ideas from multiple resource management approaches. The proposed mechanism is based on the assumption that many algorithms are pipelined and contain stateless points. Stateless points are moments where new and independent data is put into the pipeline. This assumption allows a migration mechanism to move multiple pipelined tasks at once without being concerned about transferring task state. This mechanism is useful when new QoS requirements affect an application and tasks must be reallocated. The mapping algorithm proposed in [63], isn't the most effective possible, since it encounters the constraints of being centralized. Nevertheless, the migration mechanisms (Figure 2.2) proposed can be very useful as parts of any run-time manager that uses migration techniques. *However, even though these approaches handle application mapping at run-time in a good way, they are designed for fixed-size applications without any malleability aspect.* Thus, no application reconfiguration is performed in response to any dynamic changes of platform's available resources and no resizing or remapping of the applications is allowed.

On the field of self-organized and dynamic systems and from the aspect

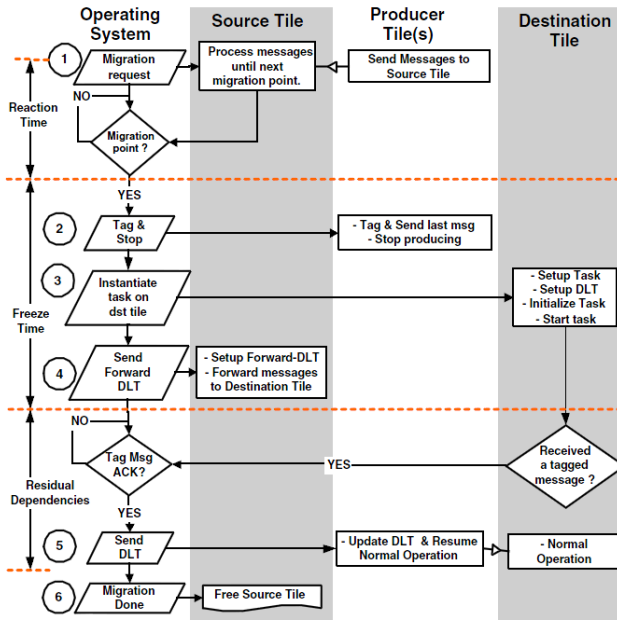


Figure 2.2: General task migration mechanism [63]

of malleable or parallel applications in general, Sabin et al. [68] present a greedy centralized scheduling strategy and demonstrate that the importance of efficiency varies with respect to the characteristics of the workload a scheduler encounters. The main idea, is that the scheduler associates defines a maximum allowable partition size on the processing elements for an application and afterward uses a greedy scheduling strategy to choose an actual partition size with aim to minimize response time. The fundamental problem of an unrestricted greedy approach to choose partition sizes for all jobs is that most jobs tend to choose very large partition sizes as far as their processing element usage is concerned. As a result, the authors [68] employ a fair-share based allocation scheme with system-wide factor which takes into account the weight of each job in order to utilize the recourses in a more fair way. Desell et al. [34] show that the application malleability provides up to a 15% speedup over component migration alone on a dynamic cluster environment. For the examination of the malleability, two representative applications have been chosen and modified to utilize malleability features. The first is an astronomical one while the second is an appli-

cation which simulates heat diffusion. Both of them have their unique need in data manipulation and malleability design. The resource availability is altered by making clusters available and unavailable to the applications. Kobe et al. [50] present a distributed agent-based task mapping for malleable applications supporting also self-organization. The agent is assigned at run-time to a random core when a new application arrives having the disadvantage of a possible communication bottleneck when a randomly selected already occupied core serves the new request. To manage regional information in a distributed way, the authors suggest that there is a directory service distributed to the entire platform. This directory service enables the agents to communicate with other agents without the need of broadcast communication. All available cores are split into evenly distributed clusters and each cluster contains one directory service running on one of the cores. The agents register themselves at the directories corresponding to the cores occupied by the own application. *The differentiators of the proposed methodologies lay in the fact that (i) the aforementioned approaches do not take into account platform's heterogeneity, leaving unutilized platforms' special characteristics and (ii) the developed approach has a small overhead in overall core intercommunication.*

Chapter 3

Memory management middleware acceleration and customization

3.1 Introduction

The current trend in computing and embedded architectures is to replace complex superscalar architectures with many processing units connected by an on-chip network. Future integrated systems will contain billion of transistors [78], composing tens to hundreds of IP cores and the number of cores to be integrated in a single chip is expected to rapidly increase in the coming years, moving from multi- to many-core architectures. Modern embedded platforms take advantage of this manufacturing technology advancement and they are moving from Multi-Processor Systems-on-Chip (MPSoC) towards Many-Core architectures employing high numbers of processing cores. From an industrial point of view, the vision goes as far as thousand core chips [20]. Additionally, the development of such many-core architectures is driven also by the development of highly parallel/multi-threaded demanding applications.

Dynamic Memory Managers (DMMs), also known as heap managers or allocators, are responsible for organizing the dynamically allocated data in memory and servicing the application memory requests at runtime [88]. The efficient implementation of dynamic memory managers, which can be implemented either in software or in hardware, plays an important role to the application performance and platform's power consumption. As illustrated in [16] simple dynamic memory management implementations often form a performance and scalability bottle-

neck in the case of multi-threaded applications, affecting the memory and energy consumption of the overall system. Thus, customized DMM solutions are critical components during the design phase of modern systems. Moreover, power consumption in embedded architectures is an important issue that system designers always try to reduce as much as possible respecting applications' performance constraints affecting the design of DMMs themselves.

3.2 Platform Used

The system we used to evaluate and develop memory management middleware acceleration and customization techniques is composed of Processor-Memory (PM) nodes interconnected via a packet-switched mesh network (Figure 3.1). A PM node is composed of a LEON3 processor with its own I-Cache and D-Cache, a Dual Microcoded Controller (DMC) and memory which can be shared among the nodes.

The key module, on which the developed techniques are based for memory and data management, is the DMC, able to simultaneously serve various requests from the local core and the remote ones via the network. Figure 3.2 shows the structure of the DMC. More information regarding its hardware characteristics can be found in [29]. The platform offers base distributed shared memory (DSM) services such as:

- virtual-to-physical (V2P) address translation
- synchronization
- cache coherency
- memory consistency
- shared memory access

These services are implemented in DMC micro-code and they are stored in DMC's control store. Moreover, they are executed by the DMC transparently to any high-level (C) application running on LEON3 processors thus alleviating the processors from performing memory management

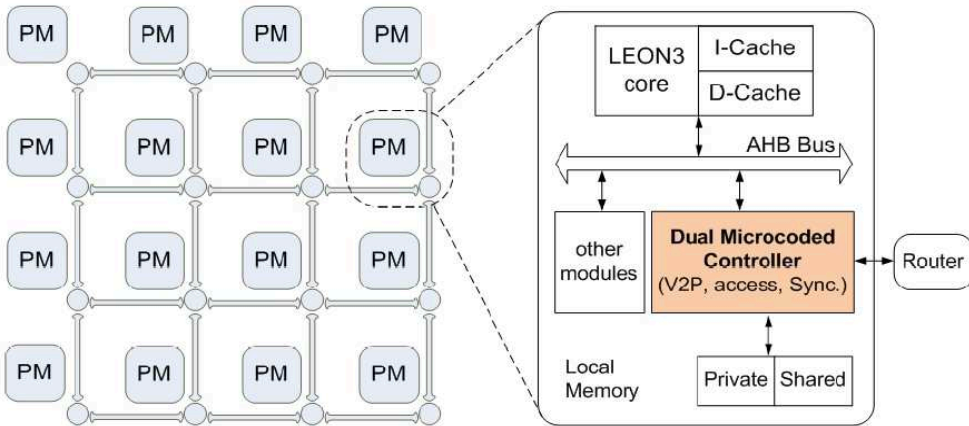


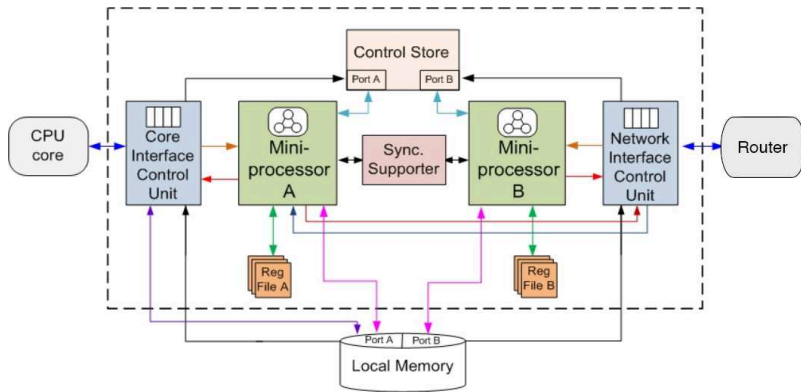
Figure 3.1: 16-node mesh McNoC; Processor-Memory (PM) node [29]

actions. *This approach offers software flexibility (due to the transparency of the services) with hardware-accelerated/improved performance.*

The used platform does not support an Operating System (OS) however; LEON3 processors are capable of running C applications using the Bare-C Compiler (BCC). Under the C programming language, dynamic memory management does not require the presence of an OS and there are no C built-in facilities for such operations. Instead, these facilities are defined in a standard library, which is compiled and linked with user applications. The developed memory management techniques are linked with C applications running on LEON3 and compiled with Bare-C Compiler 1.0.36b.

To speed up frequent memory accesses as well as to maintain a single logical addressing space, the local memory of each node is partitioned into two parts: private and shared. Accordingly, two addressing schemes are introduced: physical addressing and virtual addressing. *The local core using physical addressing can only access the private memory. All shared memories are globally visible to all nodes and organized as a single virtual addressing space using virtual addressing and virtual-to-physical (V2P) translation.* Such translation incurs overhead but makes the DSM organization transparent to the application and the other DSM services, thus facilitating programming.

Figure 3.3 shows the DSM organization and V2P translation. On the



90 nm synthesis results		
	Optimized for speed	Optimized for area
Frequency	481 MHz (2.08 ns)	472 MHz (2.12 ns)
Area	57K NAND gates	50K NAND gates

Figure 3.2: The DMC architecture and synthesis results [29]

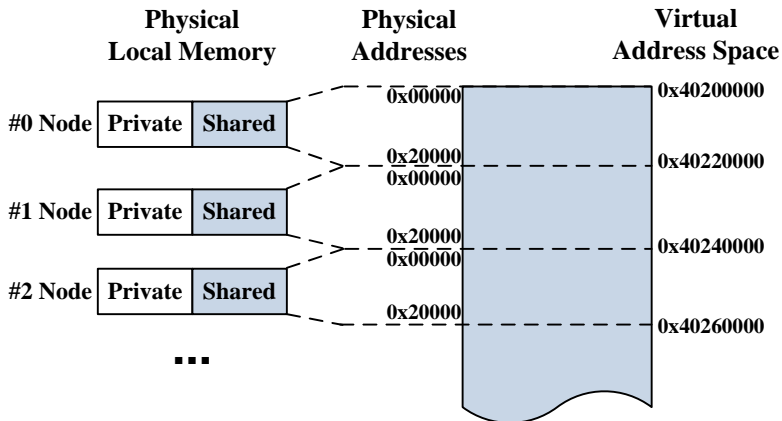


Figure 3.3: DSM organization and V2P translation

left, there are the platform's nodes each of which have their private and shared memory. The physical addresses of the shared part range from 0x00000 to 0x20000. Under V2P translation in the DSM environment, all shared memories are organized as a single virtual addressing space. The application uses the virtual addresses (0x40200000, etc.) in order to access the shared memory and the triggering of the corresponding physical node is performed but the DMC after the V2P translation.

3.2.1 Cache coherency and memory consistency

As mentioned in Section 3.2, the low-level services of the platform, are transparently used by other high-level services, such as dynamic memory management.

Maintaining coherent caches with efficiency in multi-core systems is a well-recognized challenge, especially when both caches and local memories are distributed. To be scalable, the platform supports a directory-based cache coherency where, for each shared memory address, there is a caching state (uncached, cached) and associated directory recording the sharers when shared. Rather than a flat organization, the directories are organized hierarchically in order to minimize the area overhead and to reduce network communication latency exploiting the communication locality. The cache coherency functions are realized in microcode, specifically, for cache coherent read, write and invalidation, respectively. Since only read and write misses are visible to the communication architecture, the DMCs process read/write misses and generate invalidation signals to sharers according to a particular cache coherency protocol (write back vs. write through, no-allocate vs. allocate), which can be configured in a caching protocol register in the DMCs. The microcoded coherency enables to flexibly support different protocols, and different directories with possible different hierarchical levels, depending on the system scale.

To enhance application performance, program and memory transaction, re-ordering is typically required to allow compiler, software, and hardware optimizations. To reason about the correct program behavior, a memory consistency model serves as a contract between the architecture and the application. Due to ordering restrictions imposed by the se-

quential consistency model, a relaxed consistency model is often favored for high performance computing architectures. The platform supports two relaxed memory consistency models, weak and release consistency. Both models set synchronization checkpoints before executing next code segments. While the weak consistency model uses one uniform synchronization checkpoint, the release consistency model differentiates acquire from release synchronization, allowing more re-ordering possibilities. The platform supports both models using a transaction counter based approach. With one transaction counter at each node, we realize the weak consistency model. With two transaction counters per node, we realize the release consistency model.

3.3 Memory management middleware acceleration

This section introduces the concept of accelerating dynamic memory management functions in the middleware level. We adopt the microcoded approach for supporting custom DMMs on MPSoC platforms, aiming for hardware performance but maintaining flexibility of software implementations. Specifically, we address the problem of providing customized microcode DMM on NoC platforms under a Distributed Shared Memory (DSM) environment. In order to guarantee high performance, the proposed DMM services are developed on top of a hardware dual-microcoded controller (DMC) [29] that a) works transparent to the the routing engine, b) is responsible for handling distributed memory requests and c) mitigates processor's workload. In order to further exploit DSM characteristics we developed a flexible and scalable distributed allocator, called MAD-DMM. MAD-DMM provides distributed functionality over a DSM environment while keeping the standard C-API (`malloc()/free()`). Last, we show an effective way of integrating DVFS mechanisms into any high-level DM manager transparently to the application developer.

3.3.1 Custom Microcoded Dynamic Memory Management

The methodology framework for supporting custom microcoded DMM on McNoC platforms with distributed memories is showed in Fig. 3.4

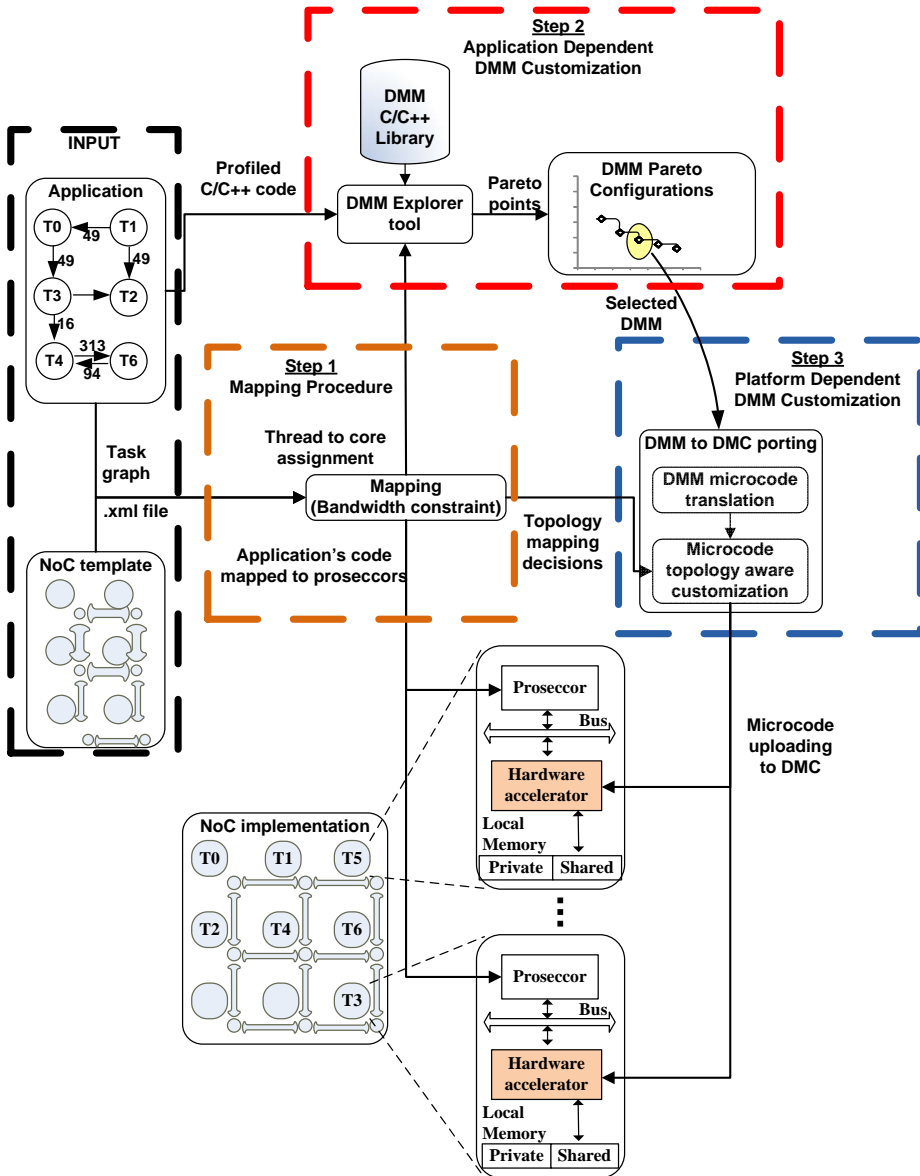


Figure 3.4: Framework for supporting custom microcoded DMM on McNoC platforms with distributed memories

Given the source code of a multi-threaded dynamic application and an NoC template, we perform three steps:

1. Application mapping to platform's cores.
2. Application dependent DMM customization.
3. Platform dependent DMM customization by refining the application-dependent DMM configuration.

3.3.1.1 Application mapping to platform's cores

The application mapping decisions have a great impact on the performance and energy consumption of the derived system, so the necessary decisions should be taken at an early design stage. The targeted application is composed of a number of tasks (nodes in the application graph). The edges among the tasks denote the communication cost and data dependencies. We formulate the mapping problem as an one-to-one mapping between two graphs. The first graph represents the application, while the second one represents the NoC platform.

- **Application Graph:** We define as application graph a directed graph $AppG(V, E)$, where each vertex $v_i \in V$ represents a kernel of the application, and the directed edge $e_{i,j} \in E$ represents the communication between the kernels v_i and v_j . The weight of the edge $e_{i,j}$ denoted as $w_{i,j}$, represents the communication load from v_i to v_j .
- **Platform Graph:** We define as platform graph a directed graph $PlatformG(N, L)$, where each vertex $n_i \in N$ represents a node in the architecture. Without loss of generality, we define $N_{Proc} \in N$ to represent the processing nodes of the NoC and $N_{Mem} \in N$ represent the memory ones. The directed edge $l_{i,j} \in L$ represents direct communication between the platform nodes n_i and n_j . The available bandwidth among these nodes is represented by $bw_{i,j}$.

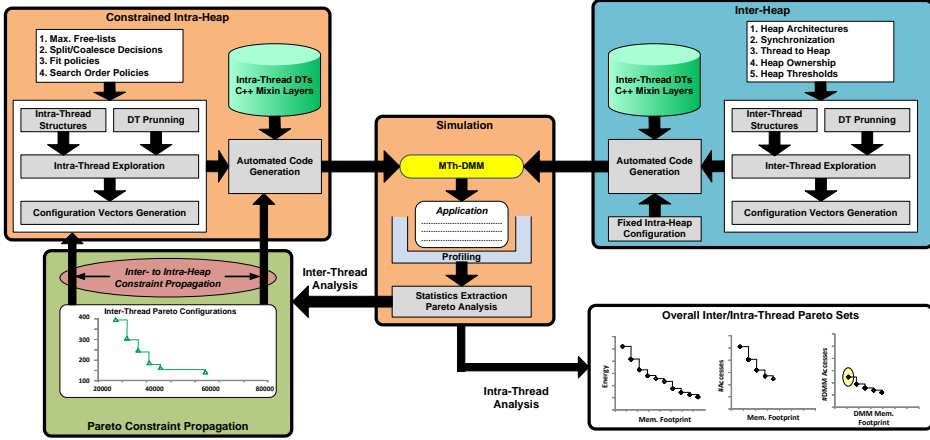


Figure 3.5: MTh-DMM Explorer tool [89]

The mapping goal is the minimization of the total communication cost (C_{total}) on the NoC, which is calculated by Equation 3.3.1.1:

$$C_{total} = \sum_i \sum_j D_{i,j} \times W_{i,j} \quad (3.1)$$

where $D_{i,j}$ is the distance (measured in hops) between the i and j nodes of the platform graph, whereas $W_{i,j}$ is the weight of the communication. For the application mapping, we use the bandwidth-constraint mapping algorithm presented in [60].

3.3.1.2 Application dependent DMM customization

In this step, we generate a set of Pareto customized Multi-Threaded Dynamic Memory Managers (MTh-DMM), tailored to the designer's constraints and the application's specific needs. MTh-DMM Explorer tool [89] was used to generate the application specific MTh-DMMs. This tool works on a platform-independent level searching among inter- and intra-thread DMM decision trees [89] as depicted in Figure 3.5.

Inter-heap level MTh-DMM exploration searches the corresponding solution space to find optimal combinations of design decisions for the dynamic application under study. The exploration procedure effectively prunes the solution space by exploiting the inter-heap inter-dependencies.

After the determination of the exploration parameters, the structures representing the examined decision trees of interest are initialized. The inter-dependency aware exploration loops are generated producing the inter-heap exploration script. The source code of each inter-heap MTh-DMM solution is linked to the dynamic application source code in order to evaluate its impact on the overall performance metrics.

A Pareto analyzer module is invoked to extract the MTh-DMM solutions presenting the most efficient trade-offs. In case that the statistics collection is performed to evaluate the inter-heap explored DMM solutions, the extracted Pareto points form the Pareto constraints to be propagated to the intra-heap level exploration tool. Intra-heap level exploration tool performs the extra refinement/customization of the inter-heap Pareto solutions. Each inter-heap Pareto solution is propagated to the intra-heap exploration tool forming its constraints. Intra-heap exploration customizes the internal heap structure considering as fixed the inter-heap decisions defined in the propagated Pareto point.

The exploration is performed based on a constraint-orthogonal partition methodology. The Pareto dimensions for extracting the application-specific memory allocators are: a) memory footprint and b) number of accesses. The MTh-DMM explorer tool generates application-specific C++ allocators.

3.3.1.3 Platform dependent DMM customization

The main goal of the third step is to move from platform-independent to platform-dependent DMM configurations in order to increase performance and exploit platform's features. This is done through porting the application-specific allocators to DMC. Two steps are required. At first, the high level C++ DMM configurations are translated to DMC's microcode. Secondly, the microcode is extended to take into consideration topology-aware features, such as memory distribution and communication cost. This extended microcode is uploaded to DMC's instruction memory.

3.3.1.3.1 DMM Microcode Translation Having as input the C++ DMM Pareto configurations, we transform the high-level code to equivalent

microcode functions. We have built generic microcode templates based on the C++ dynamic data structures and exploit platform's features. The microcode functions that are produced are fully configurable. Fig. 3.6 shows an example of the C++ to microcode translation for the First Fit [88] algorithm. More specifically, according to platform characteristics the designer can generate a large number of Heap organizations by configuring different DMM parameters, such as:

- **Number and Type of Fixed List Heaps:** This type of Heaps serve fixed sized allocation requests and in a quick manner. They can be viewed as memory caching mechanisms for dynamic data thus having a great impact on DMM's performance and memory fragmentation.
- **Heap size:** The maximum assigned memory space that DMM can use for allocation request in a specific heap. In case heap size threshold is crossed, additional memory is required from the system, otherwise allocation requests fail.
- **Heap positioning:** Refers to the mapping of the overall heap organization onto the distributed memory of the platform. With microcode functions we can generate various heap organizations mapped onto the NoC nodes. For example, we can characterize heaps according to their position either as local (heaps laying into processor nodes) or as a global heap (heaps laying into memory nodes). Heap size is closely connected with heap positioning due to different memory sizes across the NoC.

By uploading microcoded DMM to DMC instruction memory, the processors of the McNoC stop being responsible for servicing DMM requests or keeping track of the status of the heaps, thus alleviating their load.

3.3.1.3.2 Customization according to memory distribution The last step is to perform topology-aware refinement. As mentioned, the C++ DMM implementations work at a high abstraction level, thus leaving to the host operating system the decision of which (part of) physical memory is accessed during allocation requests.

However, the management of accessing physical memory becomes dominant in MPSoC architectures due to memory distribution over the plat-


C/C++ code

```

inline void * first_fit_free_list (size_t sz) {
    // Check the free list first.
    freeObject * prev = &head;
    while (prev->next != &tail) {
        if (Header::getSize(prev->next) >= sz) {
            freeObject * ptr = prev->next;
            prev->next = ptr->next;
            if (prev->next == &tail) {
                tail.next = prev;
            }
            return (void *) ptr;
        }
        prev = prev->next;
    }
    return 0
}

```

C/C++ to
DMC Microcode



```

FIRSTFIT:
set A1 {N_FREES}           ; If there are no empty blocks perform an INSERT
set A6 {G_HEAP_START}
beqz A1 INSERT
LOOP:
set A4 0                   ; Check the indexes of the single linked list,
                           ;to check where is the empty block
add A6 A6 4
lw *A6 A2
lrs A2 A4 1
beqz A4 FOUND_EMPTY_BLOCK ; Empty block found?
add A6 A6 A2
jmp LOOP
FOUND_EMPTY_BLOCK:        ;As soon an empty block is found we check
                           ;whether we can accommodate the allocation request

set A5 DATA
bleq A5 A2 ADD ;If so, INSERT (place) the element to that address
sub A1 A1 1
beqz A1 ADD               ;If not, check other free blocks (if any), else INSERT
                           ;as a new element to the end of the Allocated List
Add A6 A6 5
add A6 A6 A2
jmp LOOP

```

Figure 3.6: Code translation example. From C++ to microcode. First Fit algorithm.

form. For a given memory distribution, we further increase performance of the selected DMM by implementing microcoded functions instructing DMC: i) which (neighboring) Local Heap is more appropriate to ask for a (remote) allocation request and ii) which Global Heap is closer.

At design time, based on *topology criteria*, we build priority tables $PT_{s,d}$, ($s, d \in N$) for each node N of the MPSoC platform. $P \in N$ represents the processing nodes of the MPSoC and $M \in N$ represents the memory ones. $PT_{s,d}$ describes the priority weight of source s accessing destination d . $PT_{s,d}$ priorities are exploited at run-time guiding DMC to try allocation to (neighboring) nodes according to $PT_{s,d}$ table, starting from the node with the highest priority. The $PT_{s,d}$ value is defined in Equation 3.2.

$$PT_{s,d} = \frac{w_1 P_{s,d} + (1 - w_1)(w_2 ML_d + (1 - w_2)(w_3 MP_d + (1 - w_3 D_{s,d})))}{\sum_{\substack{i \neq d \\ \forall i}} \{w_1 P_{s,i} + (1 - w_1)(w_2 ML_i + (1 - w_2)(w_3 MP_i + (1 - w_3 D_{s,i})))\}} \quad (3.2)$$

where $i \in M$, $P_{s,d}$ and $D_{s,d}$ are the power consumption and delay of the (s, d) link respectively. ML_d and MP_d are the memory latency and memory power consumption per access of the d memory respectively. Also, $\sum_{i=1}^3 w_i = 1, w_i \geq 0$, are the weights for configuring the cost function. The microcode functions responsible for triggering remote Heaps (local or global) are totally independent and transparent to DMM's code.

DMC uses the message passing policy to propagate information to neighboring nodes. In that way, the execution of microcode to a different node is allowed even if the remote DMC has not received any signal from its own local core. What mainly affects the scalability of the DMM solution is: a) the thread to heap mapping and b) the heap to memory mapping.

Figure 3.7 shows an illustrated example of customization according to memory distribution. We assume a 3×3 NoC architecture which consists of three memory nodes ($M_{(0,1)}$, $M_{(1,2)}$ and $M_{(2,0)}$). There are three processing nodes executing threads ($Th_{(0,2)}^d$, $Th_{(1,0)}^d$, $Th_{(1,1)}^d$) with dy-

dynamic allocation operations and the rest execute code with static data ($Th_{(0,0)}^s$, $Th_{(2,1)}^s$, $Th_{(2,2)}^s$). All processing nodes have their own Local Memory (LM). Topology aware DMM customization manages threads with dynamic data. According to Equation 3.2 we build the priority access table presented in Fig. 3.7(b) (0 = highest priority, 8 = lowest priority).

Looking at the priority table (Fig. 3.7(b)) for each $Th_{(i,j)}^d$ we build a Single Linked List (SLL) structure containing microcoded memory distribution-aware functions responsible for triggering the correct physical memory when needed. The SLL structure is presented in Fig. 3.7(c). The microcode templates responsible for triggering remote memory nodes are presented in Fig. 3.7(d). We have selected the message passing policy to propagate information to neighboring nodes. The microcoded functions are totally independent and transparent to DMM's code. They are placed at the end of the code and they are automatically triggered when the local DMC asks for a remote (de)allocation request. With these functions, the execution of microcode to a different node is allowed even if the remote DMC hasn't received any signal from its own local core.

3.3.1.4 Evaluation of Custom Microcoded Dynamic Memory Management

The application we use as a test driver is a combination of several real-life kernels (performing packet processing, encryption, scheduling, etc.) that are present in network applications [13] and it is presented in Figure 3.8. The application consists of 5 kernels and each kernel is executed in its own independent thread and communicates asynchronously with the other kernels through asynchronous FIFO queues:

- Network traffic corresponding to activities like VoIP, FTP and web browsing, and which have been studied in [40]. This execution thread feeds the entire system with data packets containing the network traffic. The information available for each data packet is: time, IP addresses of the source and destination, source and destination ports and the size of the package. This thread engages the memory needed for the data packet (without including the

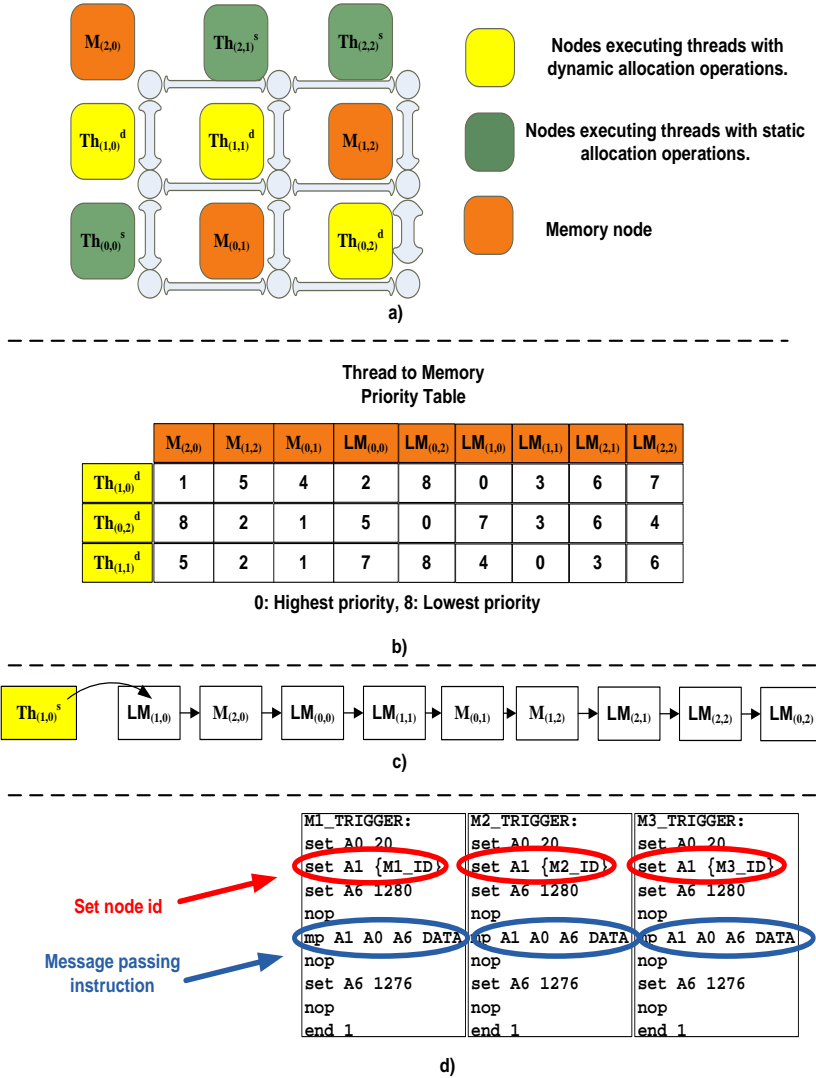


Figure 3.7: NoC memory distribution-aware DMM customization example: a) Selected topology and mapped cores, b) Thread to memory priority table, c) SLL structure. d) Microcoded topology aware function templates.

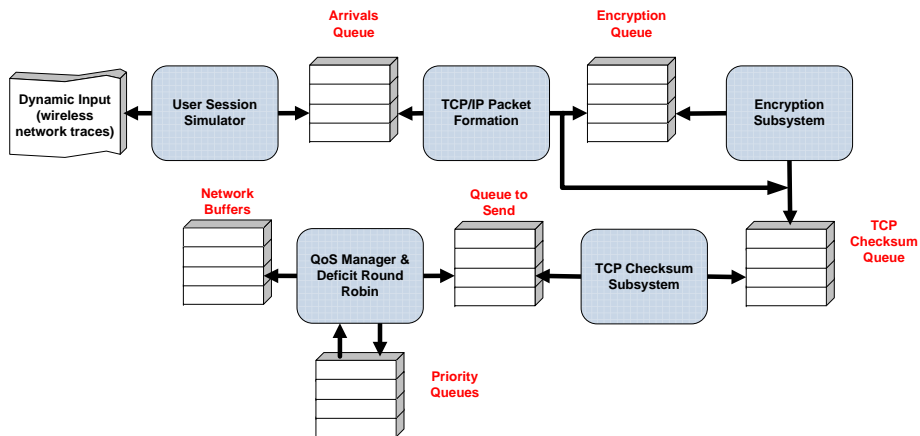


Figure 3.8: The used multi-threaded application. Squares define the different threads which communicate asynchronously through asynchronous FIFO queues

information in the header).

- Creation of a TCP/IP packet. This thread is responsible for assembling a complete TCP/IP packet by completing the information needed in the packet header. We can liken the operation of this thread by calling the `write()` (a system call), and thereby built in the entire package. The total packet size is increased by 40 Bytes. The completed packet is inserted in the corresponding queues to be encrypted or for error checking (TCP checksum) depending on whether the connection is encrypted or not.
- Encryption (packages that are part of an encrypted connection are encrypted according to the DES algorithm). This thread reads data (payload) in packet block sizes of 8 Bytes, and after the encryption it transfers the packet in the queue where packages wait for error checking.
- Creation of the TCP checksum. This thread calculates the checksum by applying the procedure described in [47]. The package contents are read by 16 bit and the thread applies to them the corresponding process. Once the checksum is created, it is written in the CRC field header and then the package is driven to the next queue and execution thread.

Table 3.1: Description of the Selected DMM Configurations

DMM	Description	Code size (# microcode instructions)			
		Conf. 1	Conf. 2	Conf. 3	Conf. 4
DMM 1	FixList ₀ (<i>block</i> = 40 <i>B</i>)	1407	485	1736	1859
	FixList ₁ (<i>block</i> = 1460 <i>B</i>)				
	FixList ₂ (<i>block</i> = 1500 <i>B</i>)				
	Generic heap				
DMM 2	FixList ₀ (<i>block</i> ∈ [0 <i>B</i> , 40 <i>B</i>])	1467	485	1796	1919
	FixList ₁ (<i>block</i> ∈ [1280 <i>B</i> , 1460 <i>B</i>])				
	FixList ₂ (<i>block</i> ∈ (1460 <i>B</i> , 1500 <i>B</i>])				
	FixList ₃ (<i>block</i> = 92 <i>B</i>)				
	Generic heap				

- The quality service manager (QoS manager) builds a list of the destinations of different packets and uses priorities for managing them. Whenever a packet enters the system, it is placed in one of the queues based on its priority type. The packets are extracted from these queues and they are forwarded to the network output according to the Deficit Round Robin (DRR) algorithm. When a packet is forwarded to the output, then the weight of the particular queue is decreased according to the packet size.

Based on the allocation behavior, the MTh-DMM Explorer tool generated the Pareto set of application specific DMMs. Table 3.1 shows the DMM configurations selected for platform dependent customization. Column 2 depicts the application-specific characteristics (Number and type of fixed size freelists) of each of the selected DMM configurations.

The topology used for the evaluation of the middleware acceleration is presented in Figure 3.9. According to mapping decisions, nodes (0, 0), (0, 1), (1, 0) are processing nodes with their own local memory. Specially each of (0, 1), (1, 0) execute 2 threads, one that handles dynamic data and another that handles only static data. Node (0, 0) executes only one thread that handles dynamic data. Node (1, 1) is a memory node that serves all requests that can't be handled by local memory. For local memories the Heap size is 4*KB* (2*KB* for fixed lists and 2*KB* for free lists). For the memory node the Heap size is 32*KB*.

For the presented topology (Fig. 3.9a) we implemented 4 different DMM configurations depending on memory distribution over the platform. The implementation of DMM configurations for the selected topology are presented in Fig. 3.9b-e. Directed edges present that an allocation request is possible to the destination from the source node while weights, based on $PT_{s,d}$, show the priority of choosing the destination node (0 = highest priority, 3 = lowest priority).

- **Configuration 1:** Pure Distributed Memory. In pure distributed memory configuration (Fig. 3.9b), each node sends allocation requests for dynamic data to its Local Heap. There is no Global Heap.
- **Configuration 2:** Centralized single Heap. In centralized single Heap configuration (Fig. 3.9c), each node sends allocation requests for dynamic data only to Global Heap (1, 1). There are no Local Heaps.
- **Configuration 3:** Distributed multiple-Heap with global Heap. In distributed multiple-Heap with global Heap configuration (Fig. 3.9d), each node first sends allocation requests to its Local Heap. If Local Heap is not able (due to lack of space) to serve any more allocation requests, the request then is sent to Global Heap (1, 1).
- **Configuration 4:** Memory distribution-aware multiple-Heap with global Heap. In memory distribution-aware multiple-Heap with global Heap configuration (Fig. 3.9e), each node first sends allocation requests to its Local Heap. If Local Heap is not able (due to lack of space) to serve any more allocation requests, then, *according to priorities*, the Global Heap or the Local Heap of another node is selected in order to serve the allocation request.

For the two selected DMMs and for each of the four aforementioned configurations, Figure 3.10 shows: i) the cycles performed until a Heap memory overflow event appears, ii) the DMM event distribution and iii) the microcode performance compared to the equivalent C implementation on the LEON3 processor. Above each bar the actual count of served DMM events (*Local Heap/ Global Heap*) is presented. Heap memory overflow is the time (counted in cycles) when Heap was unable, due to lack of space, to serve any more allocation requests. According to Figure 3.10, when DMM is aware of the memory distribution, the time

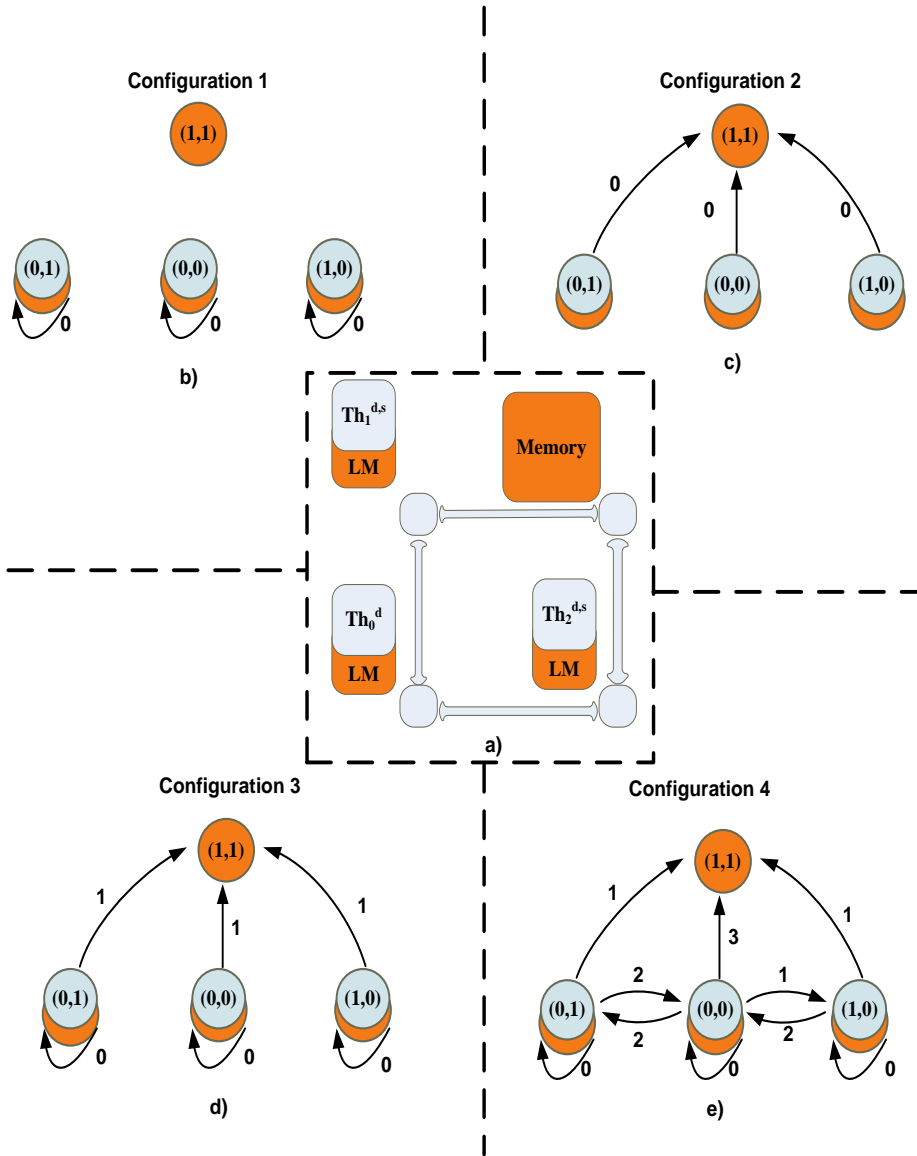


Figure 3.9: a) Topology used for evaluation 2×2 NoC with 3 processing nodes with local memory (LM) and 1 memory node. b) Pure Distributed Memory c) Centralized single Heap d) Distributed multiple-Heap with global Heap e) Memory distribution-aware multiple-Heap with global Heap

in which Heap overflow appears, increases. Specifically, configuration 4 performs $7\times$ more cycles for DMM 2 compared to configuration 1. Also, when DMM is aware of the memory distribution, the number of served DMM requests increases. For example, configuration 4 achieved to serve approximately $7\times$ more DMM events for both DMM 1 and DMM 2 compared to configuration 1 verifying also the first result of Heap's lifetime increase. Additionally, according to Figure 3.10, DMC serves the same number of DMM events in fewer cycles, performing faster than its corresponding C implementation (for the same configuration). Specifically, DMM events performed by DMC (microcode) are on average 25% faster than LEON3 (C code). This happens because DMC is responsible for handling distributed memory requests and so every time LEON3 wants to access the memory, DMC is responsible for establishing the communication.

Figure 3.11 shows: i) the average accelerator cycles, ii) the cycles spent due to memory stall and iii) the average energy consumption (pJoule) consumed per DMM event for DMM 1 and 2. Configuration 1 appears to be the fastest one, however it is the one that first exhibits Heap memory overflow. As expected, configuration 2 is the slowest among all. It needs more cycles since all processing nodes access the same global Heap for each (de)allocation and they are stalled due to memory synchronization (safe-lock) mechanisms. Configuration 3 offers good performance and additionally being more resilient in comparison to configurations 1 and 2. Configuration 4 requires a little more cycles than configuration 3 but it is a small penalty compared to the fact that it is the best solution regarding Heap memory overflow and served DMM events. We accounted energy consumed from the execution of the DMM microcode (based on post synthesis estimations at 0.09 um^2 of the DMC [29]) and the memory accessing pattern to the local and global heaps (based on Cacti [83] estimations). Configuration 2 consumes 6% more energy compared to configuration 1, since all its DMM events occur on the global Heap and the local controllers use their message passing instructions to guide the global Heap. Configuration 3, consumes approximately 18% and 19%, for DMM 1 and DMM 2 respectively, more energy in comparison to Configuration 1. This is caused by the fact that Configuration 3 consists of more microcode instructions (Table 3.1) and thus energy consumption is increased. Configuration 4, consumes approximately 25% more energy in comparison to Configuration 1. Also it consumes the highest energy amount due to the augmented code size and the often communication

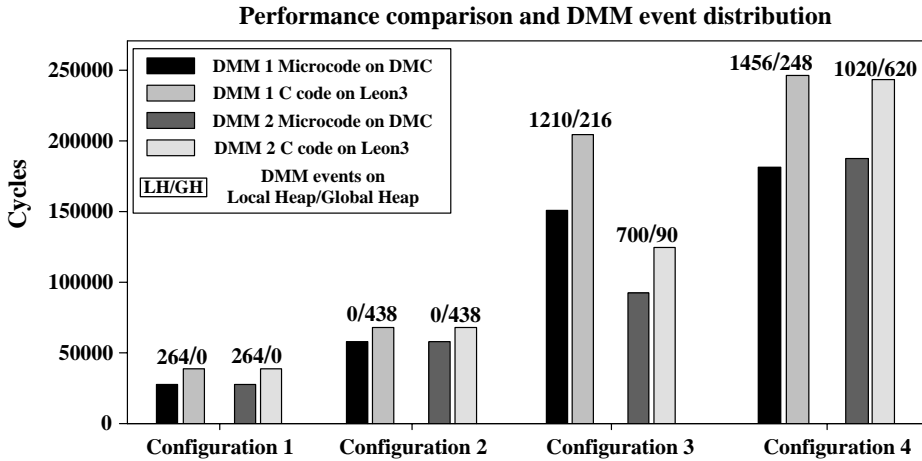


Figure 3.10: Performance comparison and DMM event distribution.

for detecting the most available Heap to use.

Experimental results show that the proposed approach for designing customized microcoded memory distribution-aware DMM (configuration 4): a) can serve more DMM events by using all available Heaps of the platform, b) increases Heap lifetime, c) is fully configurable and easy to use (offering microcoded templates), d) achieves better performance exploiting the presence of the DMC for handling distributed memory requests, thus mitigating processor's workload and e) has a negligible penalty regarding energy consumption.

3.3.2 Conclusions

The experimental results showed that in the proposed memory distribution-aware DMMs the Heap overflow chance is reduced, while the served allocation requests increase with a small penalty in average cycles and energy per DMM event. Specifically for the presented application, the gain was approximately $7\times$ for served allocation requests with a small increase of approximately 14% to average energy consumption per allocation compared to the Pure Distributed Memory organization. Also, the microcode approach is on average 25% faster than the C implementation enhancing the reason for choosing a hardware controller for handling distributed memory requests.

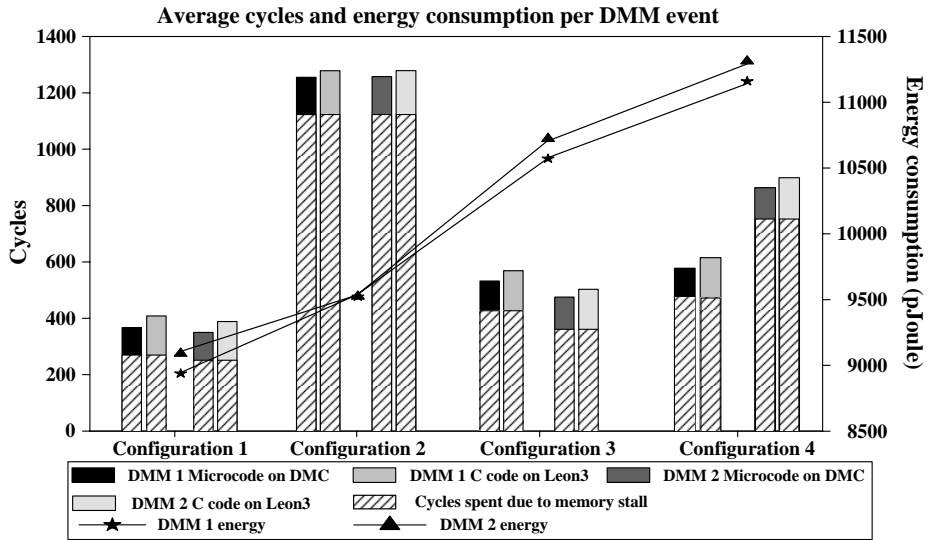


Figure 3.11: Average cycles and energy consumption per DMM event.

3.4 Microcode-accelerated distributed dynamic memory management

Concerning previous research works on the dynamic memory management field, simple DMM implementations often form the performance and scalability bottleneck in the case of multi-threaded applications [16].

The existing approaches that handle dynamic data requests rely mostly on software solutions that even though they offer flexibility, they require many processor cycles resulting in performance degradation. On the other hand, the available hardware solutions even though they are faster, they do not take full advantage of platform's characteristics and they follow a more centralized approach. However, such centralized approaches have several disadvantages. First, they create a central point of failure, which renders the whole system unusable when the central core fails. Second, a central core can hinder scalability, because it is a bottleneck for processing and communication.

In this Section, we present a flexible, distributed and scalable allocator, called MAD-DMM, developed on top of a dual-microcoded controller

(DMC) [29]. *The contributions of MAD-DMM are: (i) MAD-DMM exploits the presence of the DMC hardware accelerator, with the usage of customized microcoded functions, for accelerating dynamic data management functions; (ii) MAD-DMM provides distributed functionality over a DSM environment; (iii) The proposed allocator uses a standard C-API (`malloc()/free()`) making the microcoded actions transparent to the application; and (iv) MAD-DMM alleviates processor's workload by letting all memory management actions to DMC.*

3.4.1 Heap Space Map

As aforementioned, the local memory of each platform node is partitioned into two parts: private and shared. Accordingly, two addressing schemes are introduced for DSM functionality: physical and virtual addressing. The local core using physical addressing can only access the private memory while all shared memories are globally visible to all nodes and organized as a single virtual addressing space. The actual physical address is provided by the microcoded Virtual-to-Physical (V2P) translation service performed by the DMC (Figure 3.3). Such translation makes the DSM organization transparent to the application and the other DSM services, thus facilitating programming. In order to offer microcode-accelerated distributed dynamic memory management in the platform we introduce the *Heap Space Map (HSM)* addressing scheme, which is built on top of the V2P service.

Figure 3.12 shows how the HSM is implemented on top of the V2P. Each node offers a part of its shared memory as part of the heap. On the left, there are the platform's nodes each of which have their private and shared memory. The physical addresses of the shared part range from 0x00000 to 0x20000. Under V2P translation in the DSM environment, all shared memories are organized as a single virtual addressing space. The application uses the virtual addresses (0x40200000, etc.) in order to access the shared memory and the triggering of the corresponding physical node is performed but the DMC after the V2P translation.

In MAD-DMM, the HSM is composed of all the available heaps/pools offered by each platform's node and it is available as a continuous space to the application offering transparency to applications' `malloc()` and

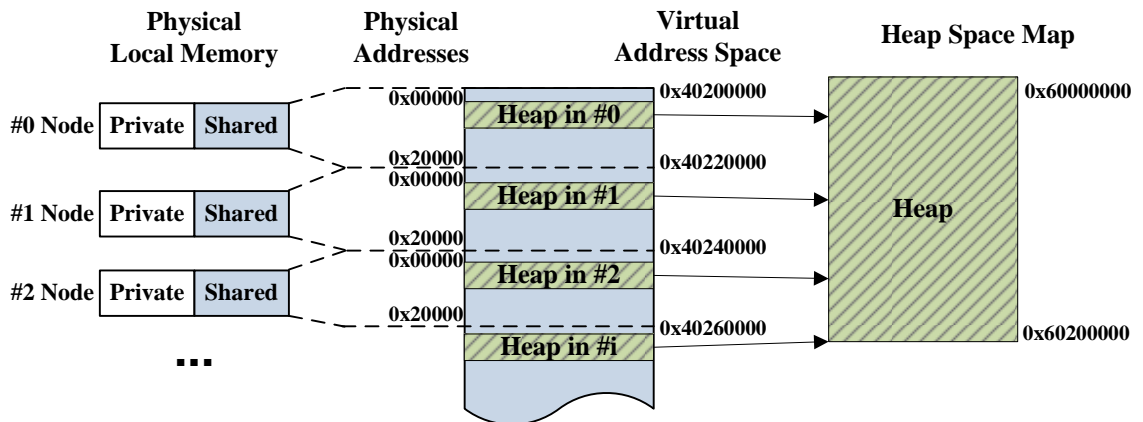


Figure 3.12: The implementation of Heap Space Map on top of V2P translation service.

`free()` calls. Each node offers a part of its shared memory as part of the heap. HSM, using the V2P service, creates another addressing scheme (e.g. starting from 0x60000000) that will be used as the heap. All nodes now can see the heap as a single address table even though the heap is actually composed of separate parts of shared memory all across the DSM environment. The heap to node translation is performed at two levels: (i) first by the HSM service (in C level) and (ii) second by the corresponding DMC (microcode V2P service).

3.4.2 MAD-DMM implementation

MAD-DMM exploits the acceleration in memory management operations, offered by the presence of DMC, by implementing at microcode all the actions for performing dynamic memory (de)allocation. In other words, DMM functions (e.g. `malloc()/free()`) have been built on DMC microcode and stored in DMC control store. Also, application interfaces have been built that allow the usage of microcode through the C application. In this way, MAD-DMM keeps software flexibility while maintaining hardware acceleration.

Figure 3.13 shows an overview of the MAD-DMM flow. The entry point is any C written application. When a `malloc()/free()` function call appears, the DMM tries to read the local heap table of the first node

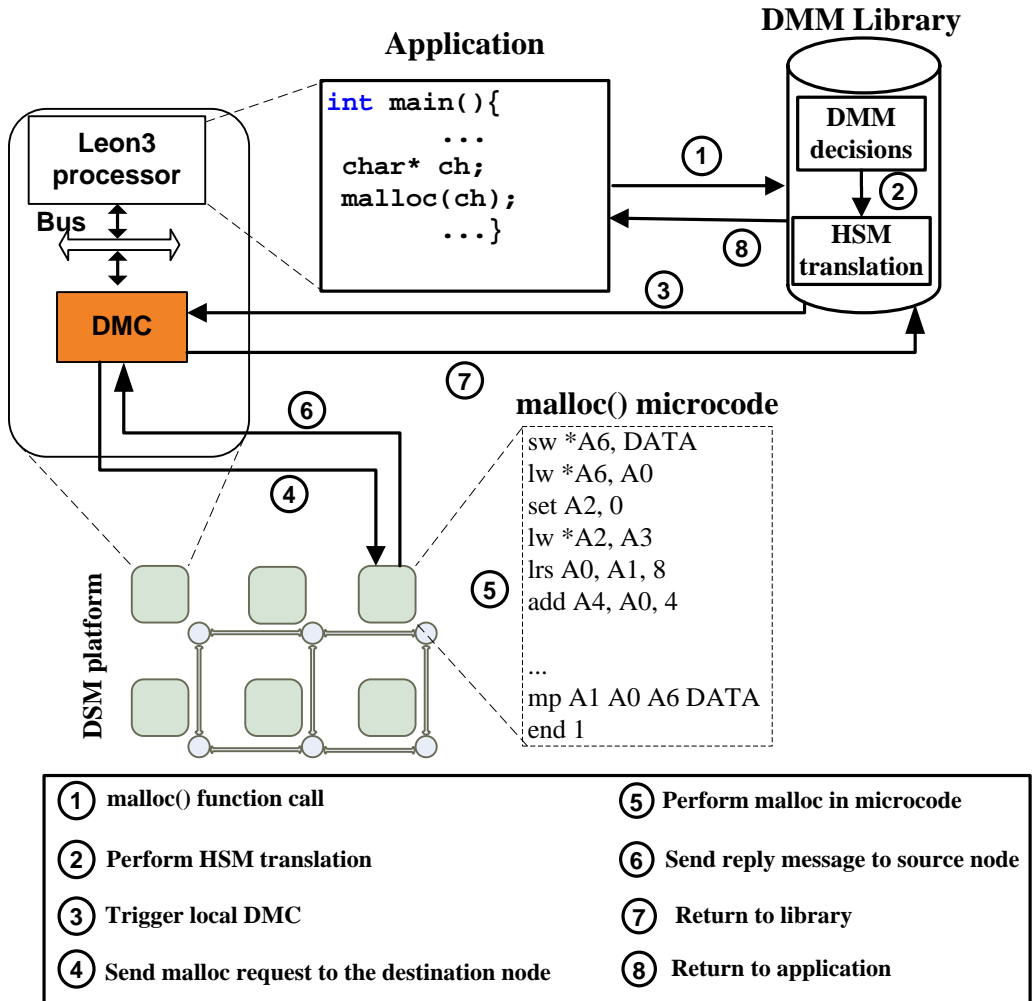


Figure 3.13: Overview of the MAD-DMM distributed allocation allocation procedure.

in HSM by acquiring a lock. If the lock is not acquired because the node is unavailable or out of space, MAD-DMM tries the allocation to the next one according to HSM. When the node is found (*destination*), the local DMC (*source*) is triggered and the following actions take place: (i) the source node sends a message with the (de)allocation request and its node id; (ii) when the destination node receives the message, it performs the (de)allocation request by executing the microcoded `malloc()/free()` functions stored in its control store and updates its local heap table; and (iii) the destination node returns a success message as a reply to the source node.

Since the C programming language does not provide built-in facilities for performing dynamic memory management operations they have been implemented at microcode level, taking advantage of the hardware acceleration, while keeping the flexibility of software implementations by offering the corresponding high-level interfaces.

MAD-DMM offers microcode-accelerated distributed dynamic memory management for shared on-chip memory systems. Similar to other allocators, MAD-DMM has a C API, offering ease of use. The main features and differentiators of MAD-DMM are three-fold: (i) support of distributed shared heap; (ii) microcode-accelerated (de)allocation operations; and (iii) scalability. The proposed allocator handles the heap continuous space under the heap space map with the help of the DMC. Additionally, the nodes on the many-core platform are seamlessly aware of the heap state. *Unlike many state-of-the-art high-level allocators [16, 57, 89], information about the allocator metadata is not stored at high-level but at microcode-level as part of each local heap table.* Each time a function call appears, the heap state information is read and the corresponding functionality is triggered. In this way, no additional internal communication between the nodes is needed.

Figure 3.14 presents an abstract presentation of the developed interfaces and the connection between C high-level language and DMC microcode. Whenever an application calls a `malloc()` function call the MAD-DMM library is triggered. One of the first functions to be triggered is the `hsm()`. The goal of this function is to check heap and find the appropriate node to serve the request. When `hsm()` function is finished, the `malloc_asm()` function performs the appropriate actions in microcode level. One of the primary things to be performed by the `malloc_asm()`

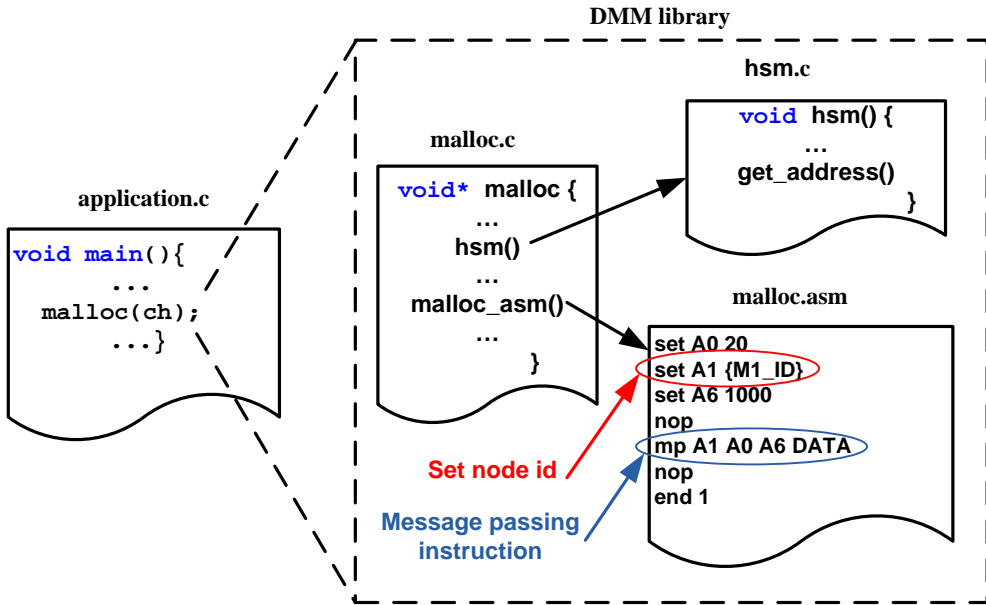


Figure 3.14: Abstract presentation of MAD-DMM interfaces

is to send the message for the allocation request to the corresponding node found by the `hsm()`. This is done using message passing instructions.

Figure 3.15 shows the communication between nodes using message passing instructions when the node is found (destination) and the lock has been acquired. In this case, a signal from (i,j) is sent to its local DMC (source). Then the source DMC, using message passing instructions, sends a message with the size of the request and its node id to the destination one. The destination DMC is triggered by its network interface (Figure 3.2) and performs the microcode actions regarding `malloc()`. Once the allocation is finished, it returns the address with a reply message to the source node and the address is propagated to the high-level layer of the DMM that returns it for usage to the C application.

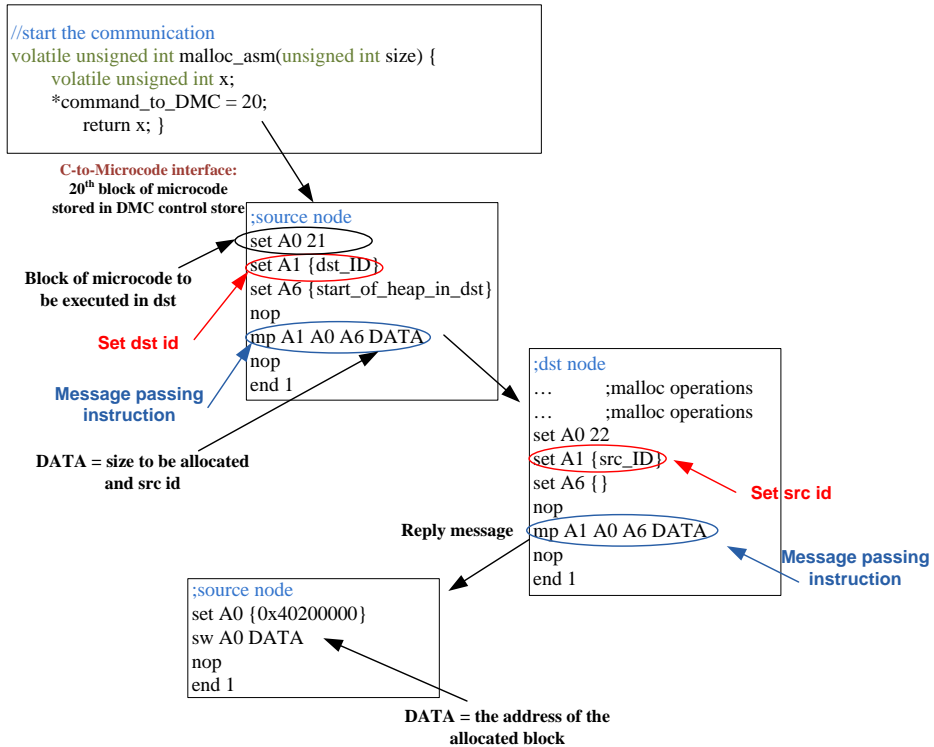


Figure 3.15: Communication between nodes using message passing instructions. The src node triggers the malloc microcode in the dst node and after the completion of the function, the dst node returns the block address back to the src DMC. Last, the block address is returned to the C application

3.4.3 MAD-DMM evaluation

For the evaluation of our approach, traces from four benchmarks were used: (i) FAST (Features from an Accelerated Segment Test), (ii) Gaussian, (iii) Integral and (iv) Matrix Multiplication. The FAST kernel is a corner detection algorithm implementation, popularly used on computer vision. The Gaussian kernel applies a Gaussian blur effect on a image whose pixels are given on a matrix. The Integral kernel calculates the integral of matrix elements and outputs the result on an output matrix. Finally, the Matrix Multiplication kernel performs matrix multiplication of an input matrix with a constant one.

All benchmarks follow the master-slave code design: One node acts as the controller of the platform responsible for handling memory management, while the rest of them are available for task execution. Before executing any task, the controller should allocate memory dynamically and then return a pointer of the allocated memory address to the executed task. Respectively, the controller is responsible for the memory deallocation after each task finishes. The cycles spent regarding dynamic data management for the used benchmarks is on average 18.8% (minimum 10.83% for the FAST application and maximum 23.12% for the Gaussian one) in respect to the overall application cycles.

In order to validate the performance of MAD-DMM, we compared it on a 2×2 NoC against (i) the memory distribution-aware microcoded DMM presented in [10] and (ii) the pure private heap high-level DMM presented in [57]. The C allocator is able to reduce the fragmentation of the heap memory taking into consideration the performance of the system.

The most common technique to prevent internal memory fragmentation is the use of free lists. The free lists are lists (i.e., double or single linked lists) of memory blocks, which were no longer needed by the application and, thus the DM allocator freed them. This technique can reduce internal fragmentation significantly and improve performance in most cases. The trade-off is that it increases external fragmentation, because the freed blocks are not returned in the main memory pool, where they can be coalesced with a neighboring free block to produce a bigger contiguous memory space.

Another technique to tweak performance and fragmentation is the use of specific fit policies. The two most popular fit policies are the first fit policy and the best fit policy. On the one hand, the first fit policy allocates the first memory block found that is bigger than the requested block. On the other hand, the best fit policy searches a part (or even 100%) of the memory pool in order to find the memory block closest to the size of the requested block. Therefore, there will be the least memory overhead per block and, thus, the least internal fragmentation. The trade-off is that the performance of the DM allocator decreases, while it spends more time trying to find the best fit for the requested block.

In the work presented in [57], the derived allocator was compared against other general-purpose ones when it was used by different applications. The results showed that the C allocator was, on average 29% faster, exhibiting also reduced fragmentation. The C allocator, that we also use as a comparison to MAD-DMM, has the blocks organized in a single-linked free list and uses the first-fit search algorithm. These features allow the allocator to find fast the blocks that are big enough to accommodate the allocation request.

Figure 3.16 shows the normalized performance of the three allocators. As expected, the microcoded allocator is the fastest of all, since all operations and decisions are performed at low-level. However, this implementation lacks a high-level API making the integration with C applications difficult. As depicted in Figure 3.16, MAD-DMM is on average 25% slower than the microcoded allocator and 10% faster than the high-level one, since most of the (de)allocation operations are performed at microcode level and only the high-level heap address manipulation is performed in C. *MAD-DMM was designed for offering distributed DMM and not using a pure private heap structure. The goal of MAD-DMM is to let all nodes be aware about the heap status the same time, having a performance penalty which in the case of pure private heaps does not exist. In other words, MAD-DMM, with the usage of DMC for accelerating memory management services and the reduction in instruction overhead, proved to be 10% faster than an allocator precisely designed for performance gain.*

In order to validate the distributed behavior and scalability of MAD-DMM, we compared (i) the average cycles per (de)allocation event (`malloc()/free()` call) and (ii) the number of microcode instructions needed for node in-

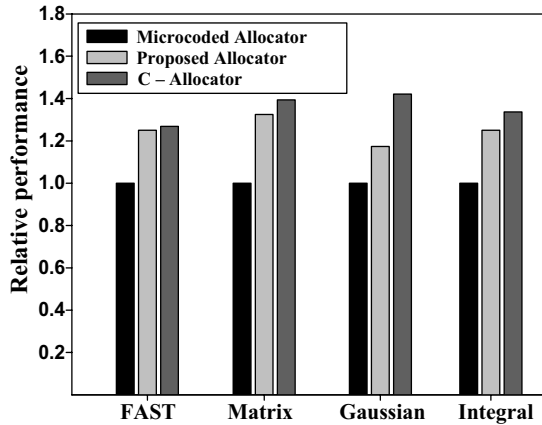


Figure 3.16: Performance comparison of MAD-DMM with [10] and [57] under a pure private heap.

tercommunication to the distributed microcoded allocator [10] for various platform sizes, ranging from 4 to 64 nodes. As shown in Figure 3.17, the microcoded DMM needs on average less cycles than MAD-DMM to complete an event when the platform is smaller than 5×5 . However, when the platform size increases beyond 25 nodes the MAD-DMM needs less cycles. This happens because the distributed microcoded allocator [10] is based on priority tables. By using the technique of priority tables, each node stores in its local memory the possible nodes to trigger in order to serve a DMM request. For a 2×2 NoC the table contains 4 records while for an 8×8 NoC the records grow up to 64 and a lot of nodes have the same nodes as targets while performing a (de)allocation request. As shown in [10] experimental results, nearly 80% of the time was "wasted" in order to acquire a lock. So, as the platform size increases the handling of these tables requires more cycles while MAD-DMM uses a lighter and more generic communication scheme between nodes, supporting arbitrary sizes of platforms and scaling well as the platform size increases. Figure 3.17 shows that the purely microcoded allocator needs on average 29% more cycles to serve an event each time the platform increases, whereas MAD-DMM needs approximately 20% more cycles. Also, MAD-DMM uses on average $1.9 \times$ less microcode instructions for node intercommunication due to the lack of the overgrowing priority tables on large platforms [10].

To sum up, MAD-DMM exploits the presence of a dual microcoded

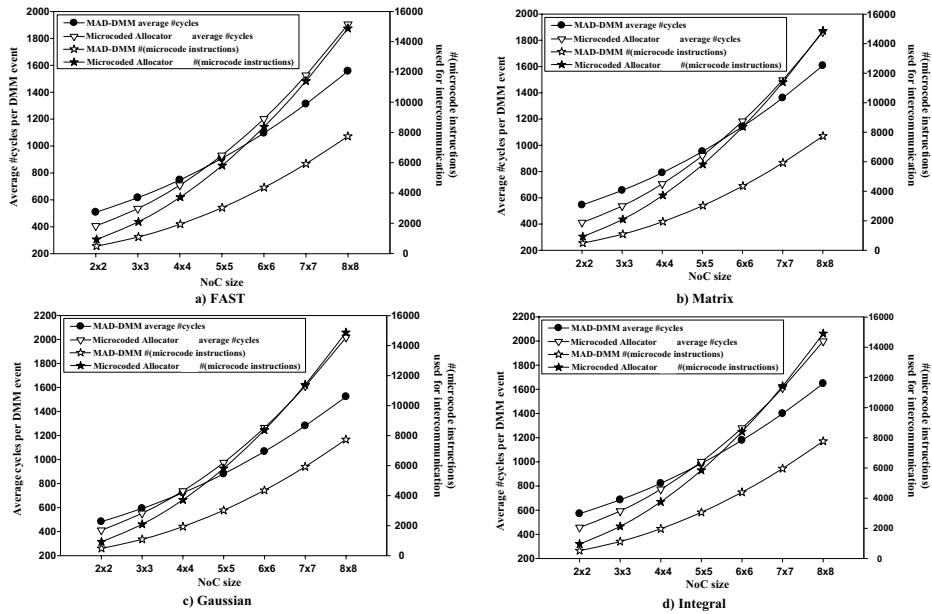


Figure 3.17: Average cycles per malloc/free call and number of microcode instructions needed for intercommunication for various NoC sizes. (a) FAST, (b) Matrix, (c) Gaussian and (d) Integral.

controller for accelerating dynamic data management functions. Experimental results showed that (i) MAD-DMM serves on average 10% faster (de)allocation requests compared to high-level managers while keeping flexibility under its C API; (ii) provides distributed functionality offering different parts of shared memory as a continuous transparent heap space; and (iii) is scalable.

3.5 Power-aware DMM on Many-core Platforms utilizing DVFS

As aforementioned, memory is an important contributor to the performance and power consumption of embedded systems. New multimedia algorithms greatly rely on dynamic memory due to the unpredictability of the input data and user interaction at compile-time. Further-

more, modern computing paradigms have set new challenges for the development of distributed applications and services, such as MapReduce [33]. As presented in [82], dynamic memory management for MapReduce algorithms plays an essential role to overall system performance and scalability. In addition the increased dynamism in data storage leads to unexpected memory footprint variations unknown at design-time.

Moreover, power consumption in embedded architectures is an important issue that system designers always try to reduce as much as possible respecting applications' performance constraints. Since DMMs are becoming prevalent components of modern systems, power consumption issues need to be taken into consideration, thus affecting the design of managers themselves. A design technique that targets power savings and it is used in modern embedded platforms is the Dynamic Voltage Frequency Scaling (DVFS) technique [44], which enables processors and other on-chip modules to operate at multiple frequencies under different supply voltages. DVFS techniques provide opportunities to reduce the energy consumption of embedded systems by scaling frequencies and voltage supply at run-time.

In this chapter, we couple the concept of dynamic memory management with state-of-art DVFS technique targeting low power consumption on many-core platforms. The goals of the presented methodology are (i) to show an effective way of integrating DVFS mechanisms into *any* high-level DMM transparently to the application developer and (ii) to give to *any* C-allocator the capability to change at run-time the voltage and frequency levels in order to reduce power consumption independently of allocators selected policies (fitting policies, coalesce and split frequency etc.). The innovations of the presented methodology are the development of a lightweight high-level window-based monitor mechanism and the integration, within the manager, of high-level DVFS interfaces responsible for changing clock frequency supply voltage at run-time. Furthermore, a DVFS decision mechanism has been used in order to make the appropriate decisions about the correct frequency and voltage level values. These techniques are completely transparent to the application developer, the application itself and the allocator selected policies and compatible with any C-allocator.

To the best of our knowledge, this is the first research work in which

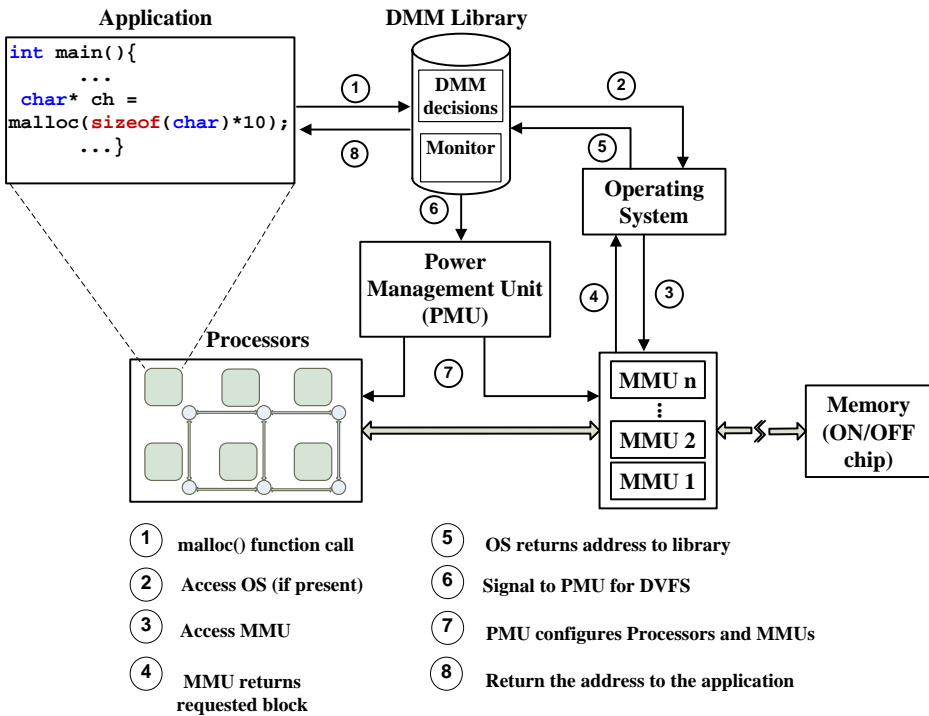


Figure 3.18: Power-aware dynamic memory management flow.

DVFS techniques are integrated as part of a DMM library, thus offering power-aware allocators, being transparent to the developer, application's code and allocator's policies.

3.5.1 Integration of DVFS mechanisms to DMM library

Figure 3.18 shows an example of how an application interacts with the enhanced DMMs and how DVFS mechanisms are triggered through the corresponding interfaces. The starting point of the flow is *any* application written in C running on a processor. The C programming language provides no built-in facilities for performing dynamic memory management operations. Instead, these facilities are defined in a standard library, which is compiled and linked with user applications. The presence of an Operating System (OS) is not a requirement and in many cases it does not even exist since DMM is an OS- and platform-independent

library. Also in small embedded systems, the available memory may not even be big enough in order to host an OS. When a DMM function call appears (`malloc()/free()`), the DMM library is triggered (STEP 1). The allocator, based on the pre-selected policies handles allocated and freed lists by employing list searching, fitting and fragmentation handling techniques. Afterwards, the DMM asks from the OS (if present) to return a virtual memory space if necessary (STEP 2). In the next step (STEP 3) the processor's Memory Management Unit (MMU), which is responsible for handling memory requests, stores the necessary information to "chunk's" headers and the requested (de)allocated block address is returned to the OS (STEP 4) and then to the DMM library (STEP 5). The developed monitor mechanisms (Section 3.5.1.1) track the required computational cost for performing the (de)allocation procedure and a corresponding signal is sent to processor's Power Management Unit (PMU) which is responsible for configuring processor's clock frequency and supply voltage at run-time (STEP 6). Regarding the signal sent to PMU suitable frequency and voltage level adjustment is applied (STEP 7). Last, the requested (de)allocated block address is returned to the application (STEP 8) and the application continues to run on the selected clock frequency and supply voltage.

In this section, we present the coupling of DMM with DVFS technique targeting low power DM managers. DMM library has been enhanced with (i) software monitor mechanisms, (ii) DVFS decision mechanism and (iii) C-supported DVFS interfaces. Figure 3.19 shows an overview of the enhanced DM managers which consist of four discrete parts.

The black rectangular boxes depicted in Figure 3.19 represent the instrumentation code used for measuring the computational cost of accesses in free-lists (Section 3.5.1.1). As soon as the allocator finishes the (de)allocation core process, the computational cost captured by the instrumentation code is forwarded to the second part, the monitor mechanism. For the monitoring we use an event-based window mechanism in order to avoid big performance overheads (Section 3.5.1.1). This part contains all the propagated computational cost for the past (de)allocation events and when this part ends the DVFS decision mechanism is triggered. The DVFS decision mechanism part (Section 3.5.1.2) is responsible for indexing the measured computational cost with the available frequency and voltage levels of the platform and finds the best

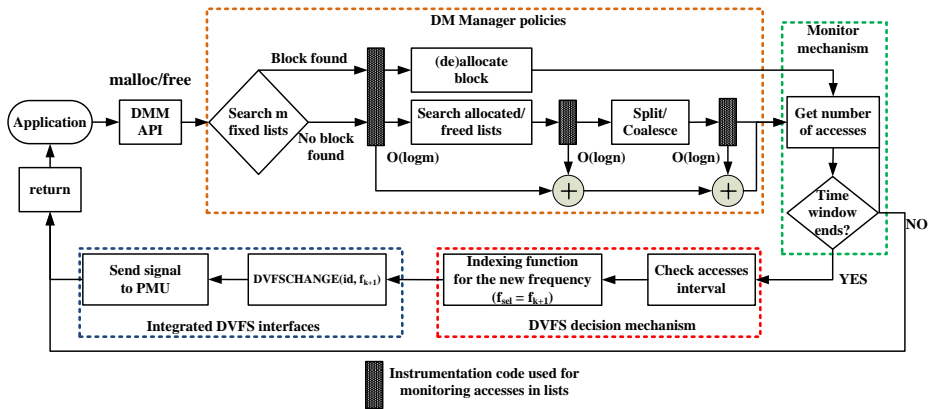


Figure 3.19: Monitoring process and DVFS decision mechanism integrated to (de)allocation process

match that meets application's needs. The frequency-voltage selected values are then forwarded to the PMU by the integrated DVFS interface/commands (Section 3.5.1.3) and the corresponding DVFS change event occurs.

3.5.1.1 Monitor mechanisms

Run-time observability in embedded system architectures is a requirement for testing, debugging, and for validating design assumptions made about the behavior of the system and its environment. The classical approach to run-time observability is to apply monitoring mechanisms to detect, collect, and report run-time information regarding the system's execution behavior. Common used techniques for monitoring include hardware probes and software instrumentation. The first ones are non-intrusive, while the latter are flexible, easy-to-implement, but with overheads. In the hardware area, monitors are usually deployed either at processor level or at system level and requires platform redesign and customization which does not offer flexibility and adaptivity. On the other hand, software monitoring mechanisms offer the cheapest and most flexible solution for gathering system's time information compared to special hardware support.

We employed software monitoring and DVFS change mechanisms that

specifically target heap management and act transparently to any other DVFS change performed by the application. The presented approach offers a lightweight high-level way of monitoring and performing DVFS targeting heap activity transparently to all other application requirements and allocator's policies. The generated DM managers have been enriched with software monitor mechanisms responsible for collecting (de)allocation procedure performance information by tracking (at high level) the list traversing time in DM manager's lists.

The advantages of the proposed monitoring mechanisms are: (i) the software monitoring can be integrated and compiled at no cost with *any* DMM library, (ii) it is platform independent, (iii) it has an easy-to-use application interface (API) and (iv) modifications can be easily done at C level. However, using this high level implementation type of integration results to a small performance overhead compared to hardware and middleware implementations that have been proved to perform faster in dynamic memory management [10] but they are still platform dependent and without a generic API.

Instrumentation Code: An important aspect of the monitor mechanisms is to minimize, or completely avoid, the intrusiveness of the monitor on the system's timing and execution properties. As aforementioned, failing to handle monitor overhead leads to probe effects which cause non-deterministic behavior in programs and big performance overheads. Software monitoring solutions for self-organized systems are based either on instrumentation code which reports at run-time performance statistics or at well-known probabilistic distributions [18, 77]. Complex self-organizing monitoring methods for DVFS are complementary to the proposed monitoring scheme.

We use an instrumentation code technique which requires insertion points in the application's code to measure the evolution of the memory management. As depicted in Figure 3.19, instrumentation code has been added after each discrete decision of the DM manager. An abstract view of the monitoring mechanism code implementation is presented in Figure 3.20. DM Managers, as aforementioned, are composed of orthogonal decision trees [89] that build in a modular way the allocator's behavior. This built-in modularity makes easy the insertion of instrumentation code for monitoring the performance of the allocator. For each discrete management policy, a new instrumentation code API

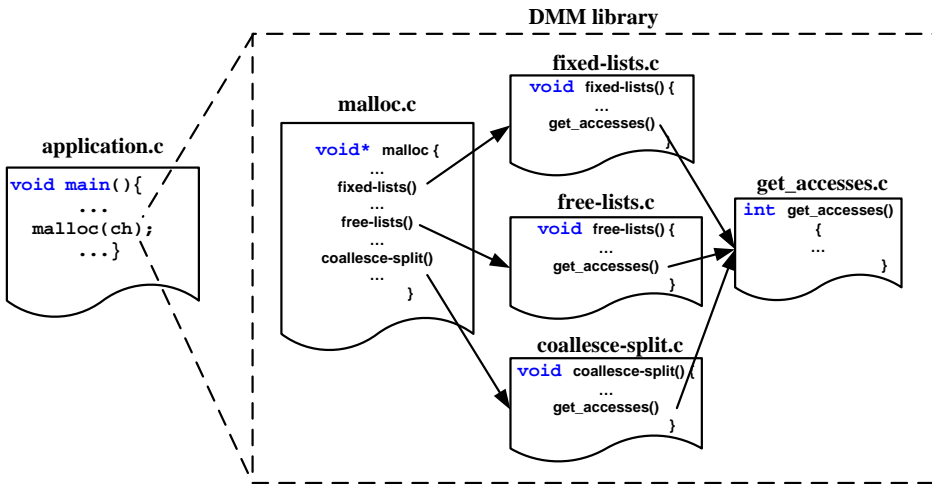


Figure 3.20: Abstract view of monitoring mechanism for gathering allocator’s accesses. The total number of accesses is propagated to the DVFS decision mechanism (Section 3.5.1.2).

(e.g., `get_accesses()`) has been implemented and inserted to the corresponding functions. *This implemented instrumentation code is responsible for reporting the number of accesses while traversing the DM manager’s lists.* At the end of the (de)allocation procedure all the necessary information regarding the total computational cost is collected and propagated to the DVFS decision mechanism (Section 3.5.1.2).

By employing this method for measuring the performance of the allocator we manage to have a fast and accurate monitoring mechanism compared to conventional system calls. An important issue to mention is that the implemented instrumentation code does not use any functions that trigger system calls (e.g., `clock()` function, in `time.h` provided by `libc`). The reason is that system calls “freeze” the execution of the application and add even more noticeable delay to the system (up to 15% more cycles compared to the implemented approach presented here) making them inappropriate for lightweight monitoring. Last, *the proposed approach does not require any changes to the application code*, since all mechanisms are integrated within the DM Manager, and it is platform independent and compatible with any C-allocator.

Window-based monitoring: In order to avoid big performance overhead, the implemented monitor mechanisms use an event-window based approach. We define as WS the window size and as $W = \{W_1, W_2, \dots, W_k\}$ the set of windows W_i . The set W is composed of mutually exclusive subsets W_i each of them containing the required monitoring information for the executed DMM events. As DMM event e_j we define either a `malloc()` or a `free()` function call. We need to mention that the number of elements in each subset W_i is the same for all subsets. For example, if $WS = 4$, then $W_1 = \{e_1, e_2, e_3, e_4\}$, $W_2 = \{e_5, e_6, e_7, e_8\}$ etc. We define as c_{e_j} and as c_{W_i} the computational cost on list traversing by the e_j event and the W_i window respectively. c_{W_i} and c_{e_j} are connected through Equation 3.3. Without loss of generality and based on the DMM exploration [89] we consider that the actual time needed by a DMM event is proportional to the time needed to traverse allocator's lists.

$$c_{W_i} = \sum_{\forall e_j \in W_i} c_{e_j} \quad (3.3)$$

At the end of each window, the number of accesses performed in the current window while traversing the free-lists of the allocator are captured by the instrumentation code and they are propagated to the DVFS decision mechanism in order to check if there is a need to change the operating frequency-voltage levels or not. According to the allocation behavior and structure (the existence of fixed lists or not, fitting policies, block splitting and coalescing, etc.) the cost for traversing lists and returning the (de)allocated block address varies and it is not known a priori. Also, it is difficult to predict beforehand the number of accesses since it changes at run-time and it depends on the following:

- $\{L_{Fix}\}$: The set L_{Fix} contains all the available fixed lists of the DM manager. The fixed lists are sorted by size inside L_{Fix} and the complexity for (de)allocating a memory block in a fixed-list is $T_{S_{szm}} = O(1)$. Thus, the cost for (de)allocating a memory block while having m fixed lists is $T_{L_{Fix}} = O(\log m)$.
- $\{L_{Dynamic}\}$: The set $L_{Dynamic}$ contains the allocated and freed free-lists. The complexity for (de)allocating a memory block in

$L_{Dynamic}$ is $T_{L_{Dynamic}} = O(\log n)$ where n is the size of the lists.

- For coalescing and splitting the complexity is based on the size of $L_{Dynamic}$ and so the complexity for performing these actions is $T_{coal||split} = O(\log n)$

So, the computational cost complexity of each window W_i is the sum of all the aforementioned costs. The total complexity is described by Equation 3.4.

$$T_{c_{W_i}} = T_{L_{Fix_{W_i}}} + T_{L_{Dynamic_{W_i}}} + T_{coal||split_{W_i}} \quad (3.4)$$

3.5.1.2 DVFS decision mechanism

The two main questions that need to be answered by each DVFS decision mechanism are: (i) when to change the frequency and voltage levels and (ii) which appropriate values should be chosen. An early or late decision for frequency/voltage level changing may result in power consumption overhead or performance degradation respectively. In general, DVFS algorithms try to compute the total CPU utilization for dedicated monitored tasks and consequently adjust clock frequency and supply voltage level maximizing this utilization. However, overestimating the utilization can result in inefficient power consumption. In this work, we choose the target clock frequency based on the aforementioned monitor mechanism reports.

At the end of each window W_i , the DVFS decision mechanism checks for the c_{W_i} value and matches this propagated cost to predefined intervals. The number of these predefined intervals depend on the number of available platform frequencies and its generic type is $[0, a^K WS]_{K_0, \dots, l-1}$ where l is the number of available frequencies and a is a variable, defined by the designer, that is related to the number of performed accesses while traversing in allocator's lists in order. Let $F = \{f_1, f_2, \dots, f_l\}$, $f_l > \dots > f_2 > f_1$, the set of the l available clock frequencies provided by the platform and I an one-to-one indexing function $I : c_{W_i} \rightarrow F$ that maps the computational cost accessing DM manager's lists to the appropriate clock frequency ($K_0 \rightarrow f_1, K_1 \rightarrow f_2, \dots, K_{l-1} \rightarrow f_l$).

For example, suppose we have a platform that supports four frequencies $F = \{f_1, f_2, f_3, f_4\}$, $f_4 > f_3 > f_2 > f_1$. At the end of a window with

size WS the DVFS decision mechanism checks the c_{W_i} and decides the new frequency f_{new} according to Equation 3.5:

$$f_{new} = \begin{cases} f_1, & c_{W_i} \in [0, a^0WS] \\ f_2, & c_{W_i} \in (a^0WS, a^1WS] \\ f_3, & c_{W_i} \in (a^1WS, a^2WS] \\ f_4, & c_{W_i} > a^2WS \end{cases} \quad (3.5)$$

The variable a can be considered as a way of how sensitive our system will be to the reported number of accesses in allocator's lists and thus how frequent the DVFS changes will be. Small value of a means small c_{W_i} intervals and so the probability to move from one interval to another and change the frequency according to the I function is bigger and DVFS changes will appear more times. On the other hand, if the designer chooses big values for a , then for the same WS the intervals are getting bigger and the probability to move from one interval to another is smaller leading to no DVFS changes. Concerning the voltage level, if the supply voltage of a module is V , the maximal frequency f_{max} at which the module can run is given by an approximation of the Alpha model [70].

3.5.1.3 Integrated DVFS interfaces

In this section we describe the developed interfaces for accessing the provided DVFS features. Based on the monitor and DVFS decision mechanisms, we need to send the signal for the DVFS change. The developed interfaces have been integrated to the DMM library and can be triggered when needed. The API calls take the form: `COMMAND(id, opt)` where `id` is the processor's id and `opt` are command-specific options. The API translates the command and the `opt` options into command code corresponding to the PMU of the processor indicated by the parameter `id`. The developed power management API provides four power management commands which can be directly accessed by the PMU through the corresponding power management APIs inside the C code. The four commands that can be accessed are: `SETOPTION`, `DVFSCHANGE`, `POWERDOWN`, and `WAKEUP`. Table 3.2 presents in more detail the integrated power management commands.

Table 3.2: Power management commands and description [26]

Command	Description
SETOPTION	used to set one configuration option of the power management unit. The command takes as a parameter a value indicating the code of the configuration option, and the value to which the configuration option should be set.
DVFSCHANGE	used to change the DVFS point of one region. The command takes as a parameter the global clock which the Clock Generation Unit (CGU) should select and the new clock divider ratio.
POWERDOWN	used to send to a power-down mode (clock gating, hibernation or shutdown) a region. The command takes as argument a parameter indicating the type of power-down mode (clock gating, hibernation or shutdown).
WAKEUP	used to wake up a region from a power-down mode. It takes as argument the port on which the node receiving the command neighbors the region that should be woken up.

The presented framework enables the easy integration of *any* any high-level allocator running on any platform (x86-based systems, McNoCs, embedded processors, NUMA machines, etc.). Of course the developed C-to-DVFS API is specific for the employed platform and its memory hierarchy. However, in the case of a different platform the only thing that is needed to change is the implementation of the C-to-DVFS API.

3.5.2 Experimental set-up

3.5.2.1 DVFS overview

A power management system has been built on top of the platform by introducing a GRLS wrapper around every originally synchronous node. The wrapper is used to ensure safe communication between nodes and to enable dynamic frequency and voltage scaling. The access point to provide the power services is given by the Power Management Unit (PMU), which controls a Voltage Control Unit (VCU) and a Clock Gen-

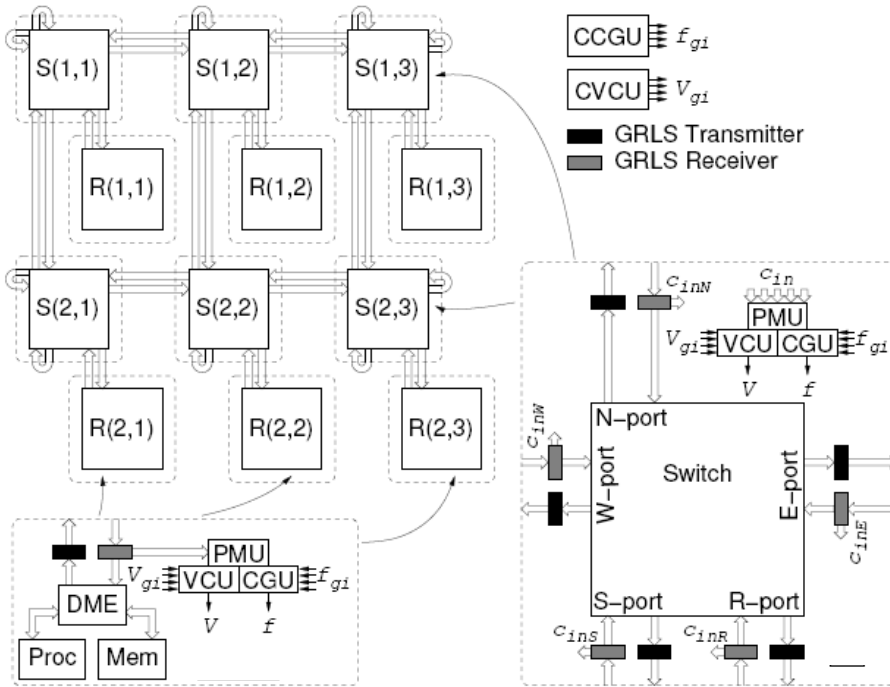


Figure 3.21: GRLS Overview [26]

eration Unit (CGU), used to control the voltage and the clock frequency respectively in the node. The PMU is accessed through the GRLS synchronizers, which detect the aforementioned special power commands (Table 3.2) and forward them to the PMU. The global structure of the McNoC platform is shown in Figure 3.21.

In a central Clock Generation Unit, up to 4 global clocks running at frequencies $f_{G0}, f_{G1}, f_{G2}, f_{G3}$, all submultiples of a frequency f_H are generated. The value for f_H can be chosen arbitrarily, but the GRLS interfaces introduce an upper bound on f_H which depends on the technological characteristics. Up to four global supply voltages $V_{G0}, V_{G1}, V_{G2}, V_{G3}$ ($V_{G0} > V_{G1} > V_{G2} > V_{G3}$) are generated in a central Voltage Control Unit, and distributed throughout the chip using up to four parallel voltage distribution grids. The global structure of the McNoC platform from the point of view of power management is shown in Figure 3.22.

Once this configuration phase is finished, the PMU can change the DVFS

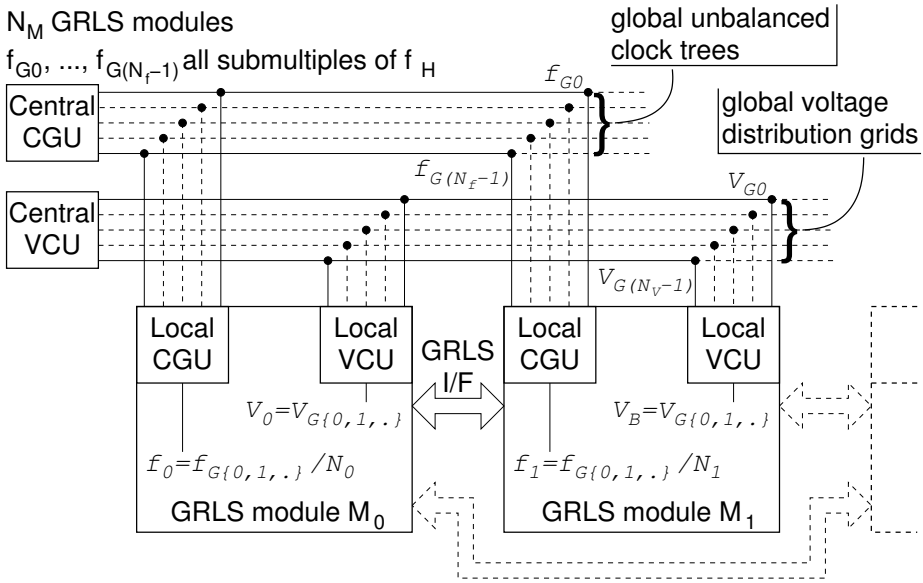


Figure 3.22: Power management architecture of the McNoC platform [26]

point of every node by issuing the DVFSCHANGE command (Table 3.2) to a node through the API. The nodes receiving a DVFSCHANGE command automatically change their frequency by locking to the new global clock and changing their division ratio, and their supply voltage by setting it to the minimal possible value. They also go through a handshake phase with neighboring nodes to inform them about the change in their operating frequency. This is necessary for the GRLS synchronizers to operate correctly but further discussion about this lies outside the scope of this work [26].

3.5.2.2 Benchmarks and execution model

For the evaluation of our approach, four benchmarks were used: (i) FAST (Features from an Accelerated Segment Test), (ii) Gaussian, (iii) Integral and (iv) Matrix Multiplication. The *FAST* kernel is a corner detection algorithm implementation, popularly used on computer vision. The *Gaussian* kernel applies a Gaussian blur effect on a image whose pixels are given on a matrix. The *Integral* kernel calculates the integral

of matrix elements and outputs the result on an output matrix. Finally, the *Matrix Multiplication* kernel performs matrix multiplication of an input matrix with a constant one. All benchmarks follow the master-slave code design: One node acts as the controller of the platform, while the rest of them are available for task execution. Resources for each task are assigned solely by the controller, so that each node can perform tasks independently from the rest without worrying for a resource conflict unless sharing is explicitly requested. In this case, synchronization mechanisms apply in terms of mutexes, as well as forked tasks or tasks that could be joined.

In this parallelization approach only the controller is responsible for handling memory management. Before executing any task, the controller should allocate memory dynamically and then return a pointer of the allocated memory address to the executed task. Respectively, the controller is responsible for the memory deallocation after each task finishes. In order to further stress out the memory allocation and deallocation actions we employed, for each of the aforementioned benchmarks, a randomization pattern for the order of memory requests. In this way we managed to have a more unpredicted behavior of each benchmark concerning the size and the time of a DMM event arrival.

3.5.2.3 Selected DM managers

In order to validate the behavior of the implemented DVFS changing system, we have used five different DM managers which have been enhanced with the presented monitor and DVFS decision mechanisms and the corresponding DVFS interfaces. Each DM manager differs from each other in the pool organization, block (de)allocation policy and whether the split/coalesce mechanisms are active or not. The variety of DM managers results in having different heap handling leading to different frequency and voltage changes, power consumption and heap fragmentation. Specifically, the tested DM managers are:

- **DM manager 1:** The power aware DM manager presented in [57].
- **DM manager 2:** The power aware DM manager presented in [57] with enabled coalesce and split policies.

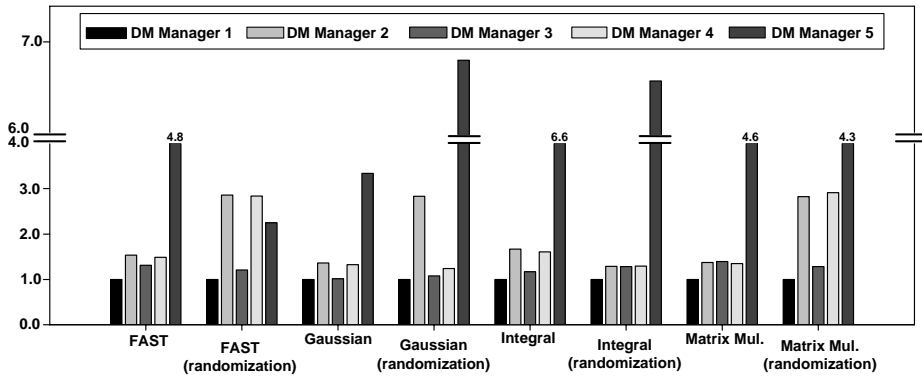


Figure 3.23: Normalized power consumption comparison of the selected DM managers without any monitoring or DVFS changing mechanism.

- **DM manager 3:** A freelist based memory allocator with best fit fitting policy and without coalesce and split policies.
- **DM manager 4:** A freelist based memory allocator with best fit fitting policy and enabled coalescing and splitting policies.
- **DM manager 5:** A fixed-sized list based allocator.

3.5.3 Evaluation

In order to validate our approach, we have performed extensive simulations of the proposed framework on the aforementioned benchmarks and platform targeting power consumption, heap fragmentation and performance overhead for each of the aforementioned DM managers.

3.5.3.1 Power consumption and heap fragmentation of the selected DM managers

The power model used to estimate power is based on the fact that power consumption is proportional to the frequency and to the square of the supply voltage at which a block runs. The model, according to the synthesis results of processors and switches (including the interconnec-

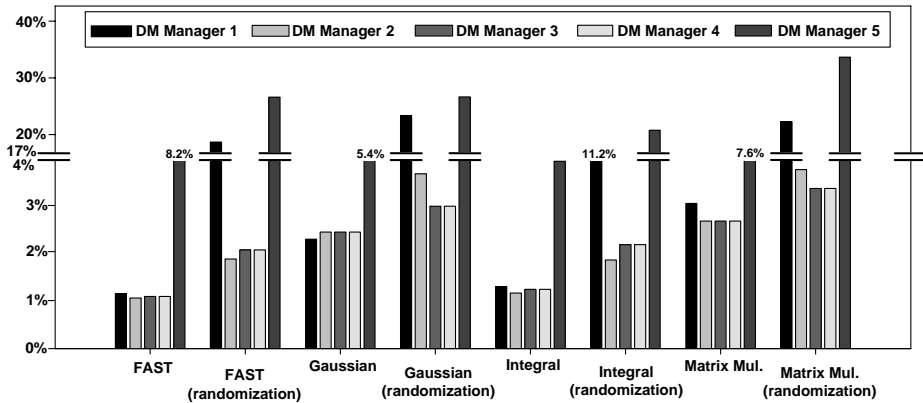


Figure 3.24: Heap fragmentation of the selected DM managers.

tion network), obtained from Synopsys Design Compiler can establish what is the power consumption of a block when running at a certain frequency and voltage point.

Figures 3.23 and 3.24 present the normalized power consumption and the heap fragmentation results for the five selected DM managers respectively without any monitoring of DVFS change mechanism. Values on top of the bars represent the values that are within the break. We define as heap fragmentation the percentage of the maximum amount of memory allocated from the system divided by the maximum amount of memory required by the application [16]. Heap Fragmentation is an important metric in the field of dynamic memory management and excessive fragmentation can degrade performance by causing poor data locality, leading to paging problems.

Figure 3.23 presents the normalized power consumption of the five selected DM managers for the used benchmarks. Power-aware DM manager 1 [57] is the baseline metric for the presented results. As expected, DM manager 1 has the lowest power consumption because it does not have any fixed lists, it uses first-fit fitting policy and there are no coalescing/splitting mechanisms available. These configurations make the (de)allocation procedure fast and minimize memory accesses leading to lower power consumption. DM manager 2 consumes on average about $1.9\times$ more power compared to DM manager 1 due to coalescing and splitting. Coalescing and splitting policies search (de)allocated memory blocks in order to reduce fragmentation resulting to an increased

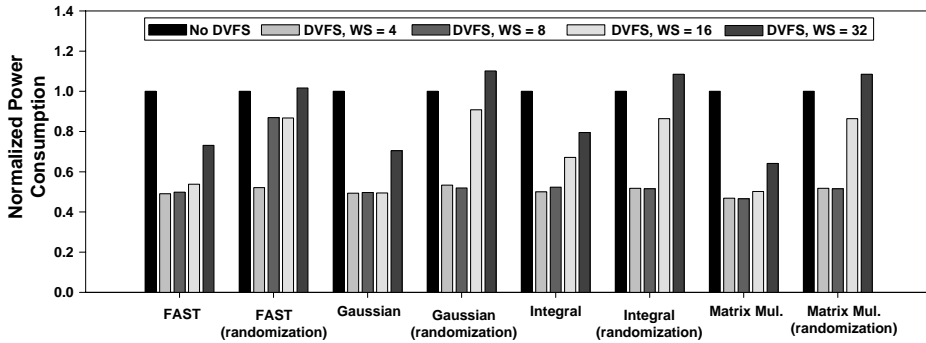


Figure 3.25: Normalized power consumption for DM manager 1 compared with the integration of DVFS mechanisms and different window sizes (WS).

number of memory accesses thus leading to more power consumption. DM manager 3 consumes on average $1.2\times$ more power compared to DM manager 1. The only difference between these two managers is the different fitting policy they use. By using best-fit fitting policy, DM manager 3 extensively searches freed list in order to return the best memory block that serves application needs and not the first one (first-fit). DM manager 4 consumes on average $1.7\times$ more power compared to DM manager 1. As aforementioned coalescing and splitting is the main reason for high power consumption. Last, DM manager 5 is the worst of all having on average $4.6\times$ more power consumption compared to DM manager 1. This happens because DM manager 5 searches in fixed-size lists first and in freed lists after in order to find the appropriate memory block.

In addition, Figure 3.24 presents the heap fragmentation of the selected DM managers. As aforementioned heap fragmentation is the maximum amount of memory allocated from the system divided by the maximum amount of memory required by the application and high values of heap fragmentation result to performance degradation. DM managers 1, 2, 3, 4 and 5 have on average 10%, 2.3%, 2.3%, 2.3% and 17% heap fragmentation respectively. As expected, DM manager 1 results to high heap fragmentation due to its first-fit fitting policy and the absence of any coalescing/splitting techniques. DM managers 2, 3 and 4 have the lowest fragmentation values because both best-fit fitting policy and coalesce and split techniques contribute to low heap fragmentation. Last,

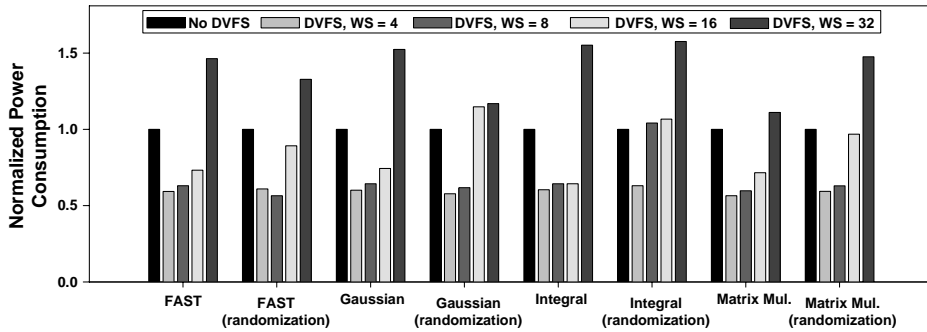


Figure 3.26: Normalized power consumption for DM manager 2 compared with the integration of DVFS mechanisms and different window sizes (WS).

DM manager 5 has the worst heap fragmentation due to the fact that it handles specific sizes it is more likely to create holes between memory blocks while trying to fit the fixed-size blocks.

3.5.3.2 Power consumptions

Figures 3.25-3.29 present the normalized power consumption for the five selected DM managers respectively with the integration of the proposed framework. The power overhead of both monitoring and DVFS change mechanisms have been estimated and they are part of the overall presented DMM power consumption for every window-size. For each DM manager we extracted the power consumption regarding (i) no frequency or voltage regulation system (no DVFS mechanism) and (ii) GRLS frequency-voltage regulation system with different window size (WS) values (from $WS = 4$ to $WS = 32$). It was estimated that the power consumption is directly proportional to the frequency at which a block runs and directly proportional to the square of the supply voltage. The model, according to the synthesis results of processors and switches, obtained from Synopsys Design Compiler can establish what is the power consumption of a block when running at a certain frequency and voltage point

As aforementioned, the proposed framework works transparently to all functionalities of the DM manager leaving untouched all decisions

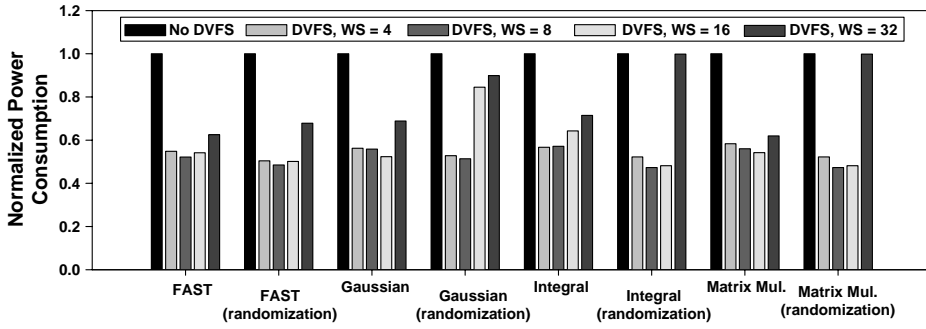


Figure 3.27: Normalized power consumption for DM 3 manager compared with the integration of DVFS mechanisms and different window sizes (WS).

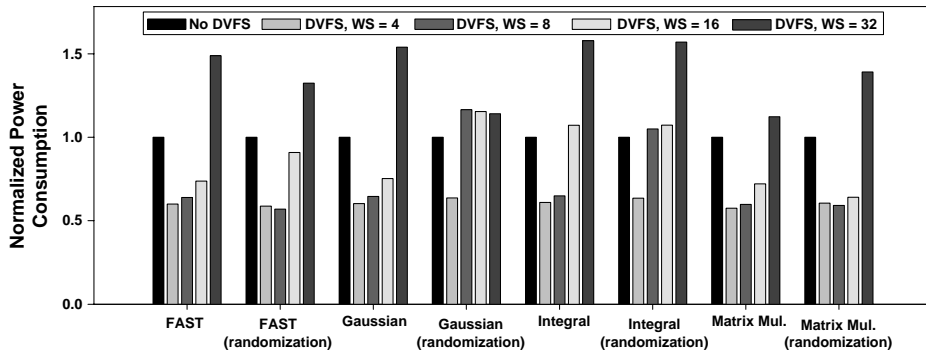


Figure 3.28: Normalized power consumption for DM manager 4 compared with the integration of DVFS mechanisms and different window sizes (WS).

such as fit policies, heap management, fragmentation, block splitting and coalescing, etc. Since these decisions/policies are the ones that affect the performance of the manager, the proposed framework focuses on the reduction of power consumption without affecting these performance metrics.

Experimental results show that by using the proposed method for monitoring and applying DVFS mechanisms, the power consumption concerning heap management was reduced by approximately 37%. Specifically, when the window size is small (e.g. $WS = 4$) the gain is even bigger reaching approximately 45% compared to the case where no

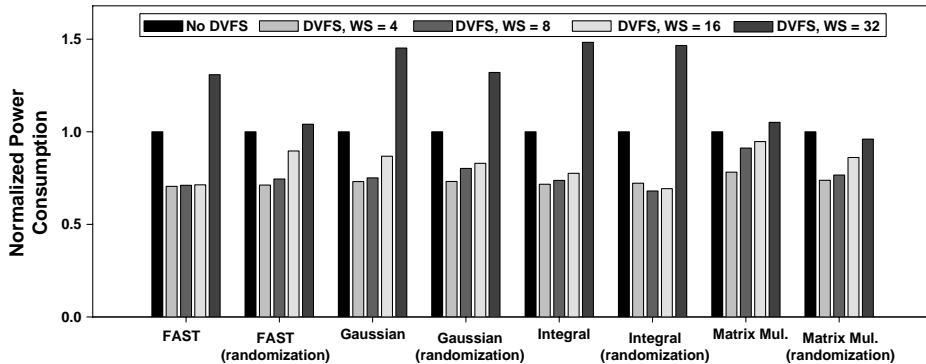


Figure 3.29: Normalized power consumption for DM manager 5 compared with the integration of DVFS mechanisms and different window sizes (WS).

DVFS mechanism is present. However, when the window size increases the power consumption gain is reduced and varies from 33% ($WS = 8$) to 21% ($WS = 16$) compared to the case where no DVFS mechanism is applied. In the case where window size is equal to 32 ($WS = 32$) there is a power overhead of about 16%. This can be explained by the fact that when the window size increases, DVFS decision mechanisms cannot take the right decision on time and time spent on heap can not be captured correctly by the monitor mechanisms.

Figure 3.25 presents the normalized power consumption for DM manager 1 with the integration of the proposed framework. By using the proposed framework, power consumption can be reduced at maximum 50% ($WS = 4$) compared to the case where no DVFS mechanism is applied. As aforementioned, when the window size increases, the power consumption gains are reduced and in some benchmarks, when $WS = 32$, there is a small power consumption overhead. In Figure 3.23 DM manager 1 was shown to be the most power efficient manager. *With the proposed additions we managed to further reduce the power consumption, making it ideal for systems targeting low power consumption.*

Figures 3.26-3.28 present the normalized power consumption for DM managers 2, 3 and 4 respectively with the integration of the proposed framework. The approximate maximum power gain for all the three managers is approximately 48% and it decreases as window size in-

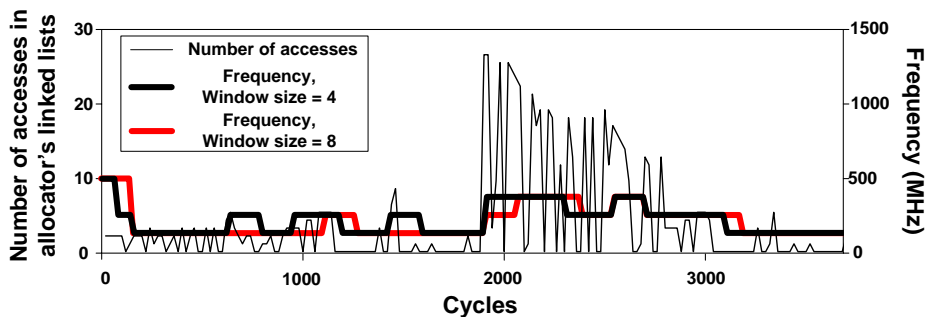


Figure 3.30: Number of accesses in DM manager's linked lists for FAST benchmark in comparison with DM manager's decisions for frequency changes for $WS = 4$ and $WS = 8$

creases (compared to the case where no DVFS mechanism is applied). In Figure 3.24, DM managers 2, 3 and 4 have been shown to be the most efficient in terms of heap fragmentation. By applying the proposed monitoring and DVFS changing mechanisms, we managed to achieve such power reduction that the final consumption is comparable with the DM manager 1, the most power efficient of all. *By combining the proposed framework, we can achieve low power consumption with low heap fragmentation making the DM managers power-aware for systems targeting low heap fragmentation.*

Last, Figure 3.29 presents the normalized power consumption for DM manager 5 with the integration of the proposed framework. And in this case, as the window size is small there is a significant gain in power consumption. But as window size increases the power gain is reduced resulting to power overhead in some cases.

Figures 3.30 and 3.31 show the number of accesses in the DM managers linked lists in comparison with the DM manager's decision for DVFS changes. As expected, when the window size is small, monitor mechanisms can follow application's needs and trigger the appropriate DVFS mechanisms. On the other hand, as the window size becomes bigger, monitor mechanisms find it difficult to follow application's needs and wrong DVFS decisions are taken or the correct decisions are taken at wrong time.

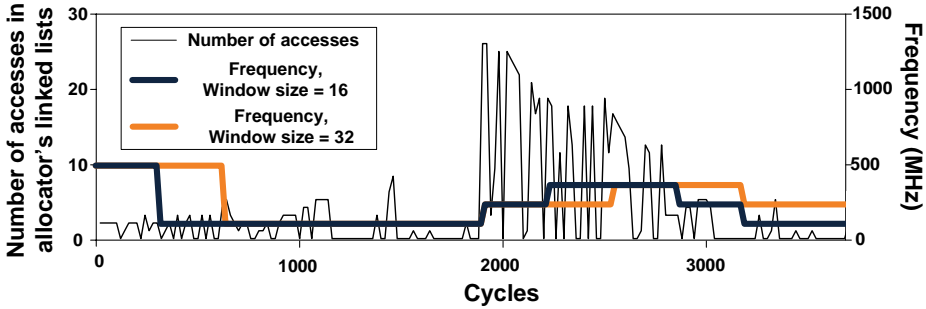


Figure 3.31: Number of accesses in DM manager’s linked lists for FAST benchmark in comparison with DM manager’s decisions for clock frequency changes for $WS = 16$ and $WS = 32$

3.5.3.3 Performance overhead

So far, we have shown that the proposed method can achieve power reduction up to 37% approximately compared to the case where no DVFS mechanism is applied. Also, we have showed that when the window size is small monitor mechanisms can follow application’s needs and trigger the appropriate DVFS mechanisms when needed. However, continuous monitor mechanisms result in performance overhead in terms of cycles. In Table 3.3 the performance overhead in terms of cycles, for each of the used benchmark (with and without randomization pattern), of the implemented monitoring and DVFS mechanisms is presented. Particularly performance overhead has been split into two parts according to whether there was a decision for DVFS change or not: (i) the overhead added only by the monitor mechanisms in cases where no DVFS change was decided (Idle) and (ii) the overhead added by the monitor mechanisms including the DVFS changing procedure overhead (switching time t_T) (DVFS). As aforementioned, it is not necessary to change frequency-voltage levels in every window if not necessary.

As expected, when the window size is small the performance overhead increases at an average of 14% more execution cycles due to the fact that monitor mechanisms are activated more often and stall application’s execution. On the other hand, when the window size increases,

Window size	Cycles	FAST	FAST (random.)	Gaussian	Gaussian (random.)	Integral	Integral (random.)	Matrix Mul.	Matrix Mul. (random.)
4	Idle	10.16%	9.45%	11.52%	8.26%	12.02%	5.47%	6.12%	5.99%
	DVFS	3.56%	4.74%	3.63%	9.38%	3.79%	10.51%	2.36%	3.79%
	Total	13.72%	14.19%	15.15%	17.65%	15.82%	15.98%	8.48%	9.78%
8	Idle	3.88%	3.88%	4.27%	4.51%	4.59%	3.92%	2.32%	2.63%
	DVFS	3.52%	3.52%	3.39%	4.64%	4.17%	4.56%	2.39%	2.71%
	Total	7.40%	7.40%	7.66%	9.14%	8.76%	8.49%	4.72%	5.35%
16	Idle	1.10%	1.42%	1.75%	2.58%	1.55%	1.67%	0.69%	1.07%
	DVFS	2.62%	2.10%	2.59%	2.34%	2.90%	3.12%	1.63%	2.01%
	Total	3.73%	3.52%	4.34%	4.91%	4.45%	4.78%	2.31%	3.08%
32	Idle	0.16%	0.33%	0.19%	1.01%	0.43%	0.95%	0.35%	0.50%
	DVFS	1.90%	1.63%	2.17%	1.69%	2.14%	1.57%	1.40%	1.83%
	Total	2.06%	1.95%	2.36%	2.70%	2.56%	2.52%	1.75%	2.33%

Table 3.3: Performance overhead in terms of cycles

the monitor mechanisms are not triggered so often and the performance overhead can be considered negligible in some cases (below 5%). Another noticeable remark is the fact that when the window size increases the overhead of the DVFS changing procedure remains almost the same while the idle monitoring cycles decreases. This is explained by the fact that in these cases there is a DVFS change command every time a monitor window ends. DM manager tries to follow benchmark's needs and makes the appropriate frequency-voltage changes in order to adapt to benchmark's needs.

According to Table 3.3 the FAST benchmark follows the aforementioned remark about the window size. The total performance overhead is approximately 14% and decreases to 2% as the window size increases. The Gaussian has approximately an average total performance overhead of 16% more execution cycles and decreases to 2% too as the window size increases. Moreover, The total performance overhead of Integral benchmark is approximately 14.5% and decreases to 2.5% as the window size increases. Last, the Matrix Multiplication benchmark has approximately an average total performance overhead of approximately 8% and decreases to 1.3% as the window size increases.

3.5.3.4 Power consumption and performance overhead trade-off

As shown in Figures 3.30 and 3.31, when the window size is small the monitor mechanisms can keep up with applications' dynamic memory needs resulting to lower voltage-frequency levels when necessary and thus reducing the overall power consumption. However, according to Table 3.3, continuous monitoring results in performance overhead in terms of cycles. On the other hand, as the window size increases wrong DVFS decisions are taken.

In order to examine the trade-off between power consumption reduction, performance overhead and window size, we define $P(e_i)$ as the probability of serving the e_i DMM event at the correct frequency and voltage operation levels. Figure 3.32 shows the $P(e_i)$, for the different DM Managers (Figures 3.32a-e), in comparison to normalized power consumption (case in which no DVFS mechanism is applied), performance overhead and window size (referenced as WS in Figure). As expected, when the window size is small ($WS = 4$) the $P(e_i)$ is on

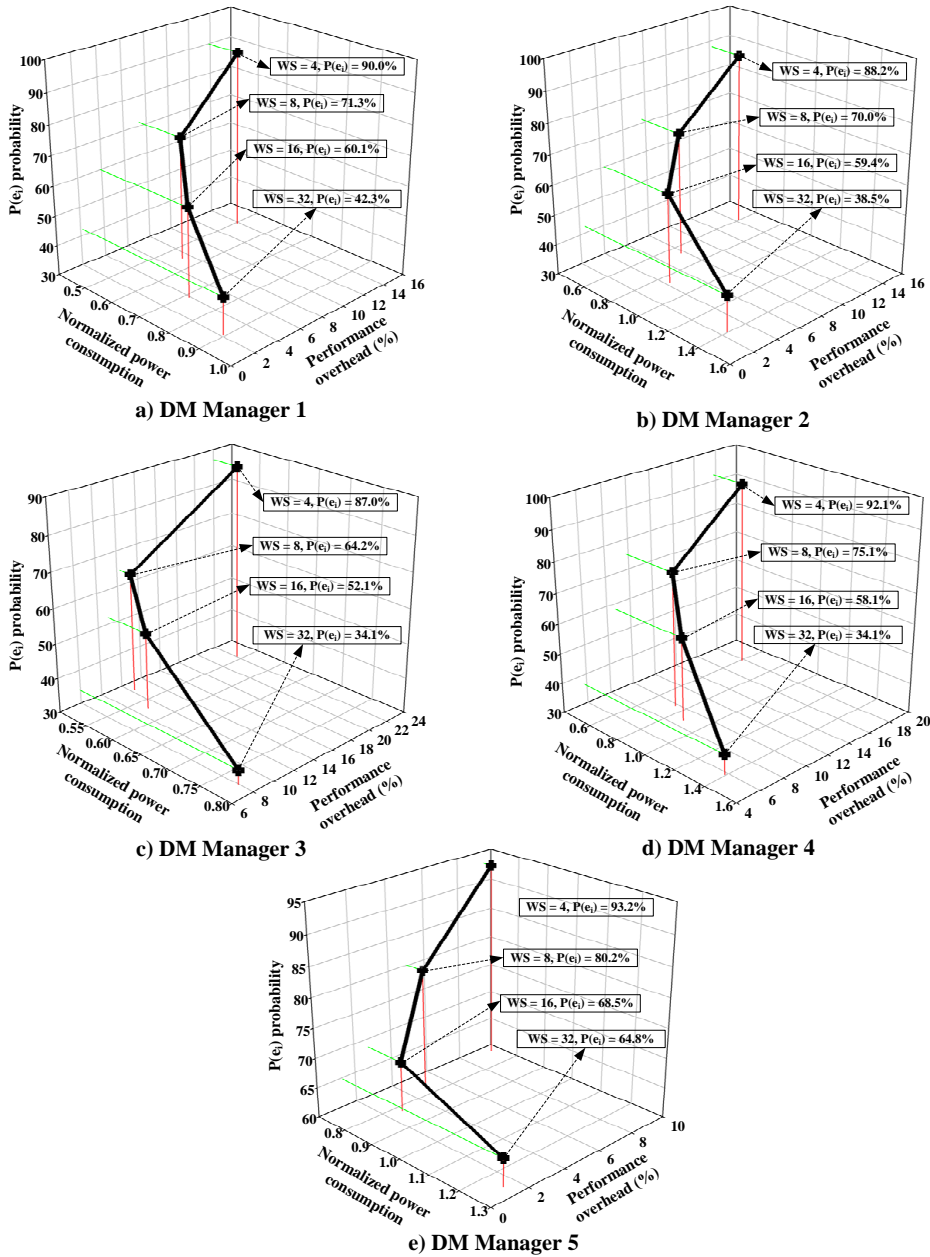


Figure 3.32: $P(e_i)$ values, for all DM Managers, in comparison to normalized power consumption (compared to no DVFS), performance overhead and window size (WS).

average 90.1%, whereas when the window size is big ($WS = 32$) this percentage falls down to 43.3%. The different features of the allocators affect their behavior and this is reflected on the different $P(e_i)$ values for the same window size as shown in Figure 3.32. For example, $P(e_i)$ values range from 34.1% (DM Manager 3) to 64.8% (DM Manager 5), when $WS = 32$, while $P(e_i)$ values range from 87% (DM Manager 3) to 93.2% (DM Manager 5), when $WS = 4$.

3.5.4 Conclusions

To sum up, we couple the concept of dynamic memory management with state-of-art DVFS technique targeting low power consumption. The proposed framework consists of high-level window-based lightweight monitor mechanisms and integrated high-level DVFS interfaces responsible for changing clock frequency and supply voltage levels. To the best of our knowledge, this is the first attempt in which DVFS techniques are integrated as part of a library responsible for handling dynamic data, thus offering power-aware DM managers. This framework is transparent to the application code and can be integrated to *any* allocator written in C. Experimental results show that by using the proposed method for monitoring and applying DVFS mechanisms, the power consumption concerning heap management was reduced by approximately 37%. Furthermore, the designer can use the $P(e_i)$ metric to explore the trade-off among the power consumption, performance overhead and monitoring granularity. In addition, by combining this method with heap fragmentation-aware DM managers, we can achieve low power consumption with low heap fragmentation values.

Chapter 4

Distributed Run-time resource management

4.1 Introduction

The concept of many-core platforms poses a new problem for the designer in a higher level of abstraction, the one of allocation of resources among the cores. Until now, we faced the problem of scheduling the tasks in one core or a very small set of cores. In a many-core platform, resource allocation has a spatial aspect as well. An application comprises of tasks being executed in parallel and every task has to be assigned to a core. The choice of core has to take into account its position as well. This is the reason why we refer to this process as mapping. An idea of what mapping refers to can be seen in Figure 4.1.

Calculating a cost efficient mapping for a given application in a short

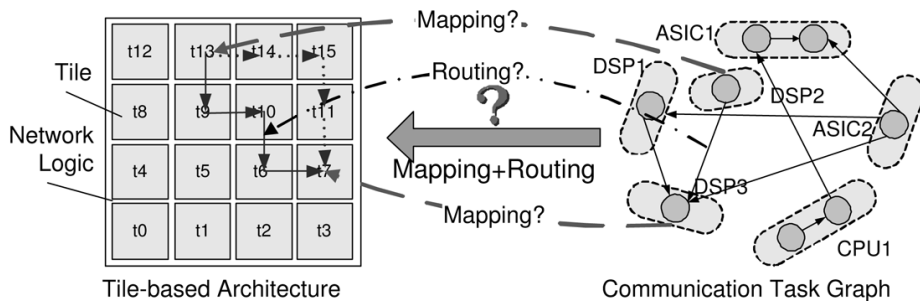


Figure 4.1: Illustration of the mapping problem [46]

amount of time is crucial. The design may have many aspects. Some common goals are to maximize performance, minimize distance between tasks or minimize energy consumption in case of heterogeneous platform. Since the mapping process is an intermediate step between the request for an application to start and the actual initialization of the application, the process must be as fast as possible.

Resource management is a key technology for the successful use of computing platforms. The run-time resource management paradigm has become prominent recently because it can deal with the run-time dynamics of applications and platforms. Thus, the efficient run-time application mapping enables the efficient usage of the platform resources, minimizing mapping time, interconnection network communication load and energy budget.

In the case of design-time mapping, the decisions are taken at the design phase having explored all the appropriate capabilities and solutions. However, this designation cannot be altered during run-time and in many cases a lot of time is required to recalculate the mapping. So, this technique can be employed only when small amount of application can be run on a man-core platform and only for specific scenarios. Apparently this is not the case in modern systems.

In run-time mapping, the resources are allocated dynamically as applications enter and exit the system, providing the necessary flexibility for the platform to function properly. An interesting fact is that run-time mapping is not only the only way to tackle the unpredictability of the incoming applications but has a number of other abilities as well: (i) *It adapts to applications' needs by exploiting the available resources. These resources may vary over time, due to different applications running simultaneously;* (ii) *it enables unforeseeable upgrades after first product release time, e.g. new application and new or changing standards;* (iii) *it avoids defective parts of a SoC. Larger chips mean lower yield. The yield can be improved when the mapping algorithm is able to avoid faulty parts of the chip and* (iv) *it is be used with reconfigurable hardware where the type of available processing elements can vary over time.*

Existing approaches to run-time mapping algorithms on many-core platforms, even if they expose some autonomic properties, are typically centralized [23]. Traditionally, a central core periodically analyzes applications' malleability and platform resources and tries to

find the best match between them. However, such centralized approaches [68] limit scalability due to bottlenecks appeared from processing and communication functions, especially in environments that require frequent configuration changes. Also, the large number of cores in modern systems, increase the failure rate of single processors resulting in system errors when parallel applications are executing [14]. Last, centralized run-time managers lack the concept of self-adaptation and self-organization, actions that trend to be a solution to modern platforms [51].

More specifically, centralized mapping utilizes one or a small set of cores in order to perform the mapping for every application that arrives. These cores are burdened with the responsibility to calculate the mapping for the whole system resulting in the following problems [6!]: *(i) Monitoring traffic is increased in volume. During the run-time mapping the centralized approach needs to collect data from the whole chip causing unnecessary traffic on the interconnection resulting in performance degradation; (ii) a high computational cost is required to calculate the mapping for the whole chip at once; (iii) there is a single point of failure. If the Centralized Manager fails for some reason, the mapping can't be performed at all; (iv) the Centralized Manager becomes a point of hot-spot as every tile sends information to it increasing the chance of bottleneck issues around the manager and (v) the system lacks scalability. As modern many-core platforms grow in size, more and more Processing Elements will be added, thus exponentially increasing the computational effort of mapping and the on-chip load.*

On the other hand, in the distributed mapping scheme, the effort of the computation is distributed on several tiles across the chip [81]. In this way, the problems of the Centralized mapping are solved as following: (i) Monitoring traffic is decreased in volume. The Processing Elements only need to send the data to their closest manager reducing interconnection load; (ii) the distributed managers only need to perform the mapping computation for the dedicated area they control. In this way, the computation demanding problem is divided in less demanding ones; (iii) There are no issues of single point of failure or hot-spots, since the smaller portions of the computation can be performed on any tile and (iv) distributed mapping scales very well with larger many-core platforms, since all that is needed is some more light-weight distributed managers, whose individual computation effort isn't

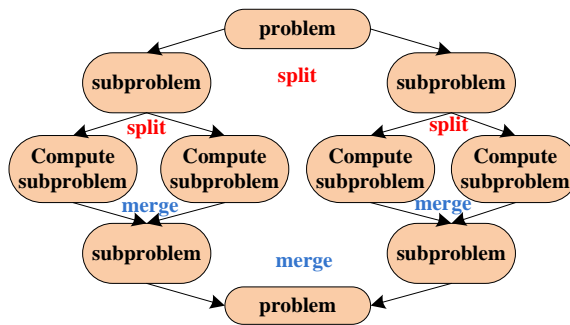


Figure 4.2: Divide and Conquer in computer science.

increased.

4.2 Divide and Conquer based Distributed Run-time Mapping on many-core platforms

Divide-and-Conquer (D&C, derived from Latin: divide et impera) is a combination of political, military and economic strategy. Its goal is to maintain control by breaking up larger concentrations of power into smaller ones. These smaller concentrations of power individually have less power than the one implementing the strategy. Historically, it has been used by great empires such as ancient Rome, by dividing the land to smaller regions and setting local Kings.

In computer science, the D&C strategy consists in breaking a problem into simpler subproblems of the same type, next to solve these subproblems, finally to amalgamate the obtained results into a solution to the problem. Thus, it is primarily a recursive method for finding solutions to a big problem as depicted in Figure 4.2. The algorithms of this type display two parts; the first one breaks the problem into subproblems, the second one merges the partial results into the final result.

The developed methodology adopts the concept of the D&C method and use it so as to perform *distributed run-time mapping* on both homogeneous and heterogeneous many-core platforms. An abstract example of our approach implementing the D&C concept is presented in Fig-

ure 4.3.

The mapping is carried out in a distributed manner. In order to achieve that, the platform is partitioned in regions, i.e. subsets of the set of all the tiles on the NoC. These regions have no fixed boundaries, and can be reshaped, created or abolished when necessary. Every new application mapping request is processed firstly by a designated tile, where the System-Wide Controller task is being executed. This task is a lightweight piece of code, implemented for every type of Processing Element on the many-core platform in order to keep the system protected from any single point of failure problems. This task's purpose is to find a region suitable to execute the new application, or take actions if the application can't be mapped for any reason. It holds easily transferable data for the whole platform, based on which the resulting region is found. The collection of that data doesn't burden the whole platform, but only specific tiles as shown later. In addition to the System-Wide Controller, there are some more designated tiles, one for each region, called Regional Controllers. As the name suggests, these tiles are responsible for any action involving the mapping in their respective region. More specifically they are responsible for:

- Computing the mapping for the region for which the controller is responsible for.
- Collecting data for the region.
- Communicating and exchanging data with the System-Wide Controller.

In the same manner as the System-Wide Controller, the Regional Controllers are meant to be executable on any tile of the region, so that the platform's functionality doesn't depend on any single tile. Once a region has been selected by the System-Wide Controller for the mapping of a new application, its Regional Controller is triggered and data describing the application is sent to it. Then the mapping is performed and its results are reported back to the System-Wide Controller.

The novel ideas of the presented methodology are:

- We propose a flexible distributed method for run-time mapping

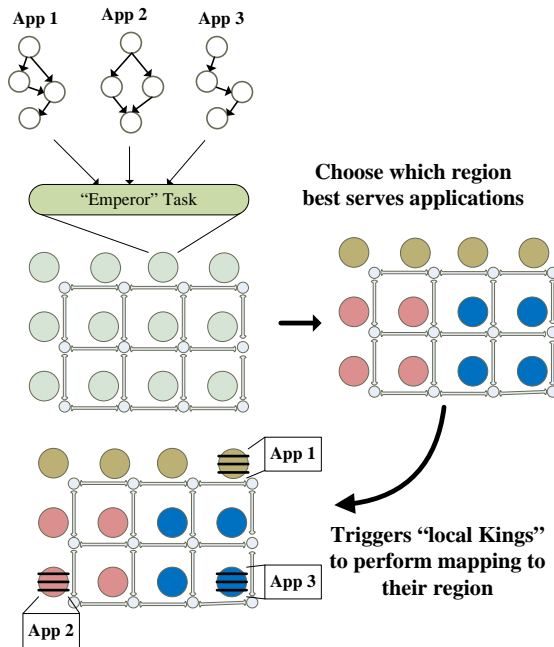


Figure 4.3: Divide and Conquer on many-core platform

on *both* homogeneous and heterogeneous many-core platforms

- The flexibility of our approach is based on the fact that our run-time mapping framework can achieve different levels of platform's resources utilization depending on application's needs in comparison with other state-of-the-art distributed algorithms [9].
- We employ a fast node swapping procedure at the final step of the run-time mapping producing even better results in terms of on-chip communication cost.

4.2.1 Proposed run-time mapping methodology framework

The goal of the proposed methodology framework is to perform bandwidth-aware run-time mapping of application(s) described by their application task graphs in many-core network-on-chip architectures described by their application graphs.

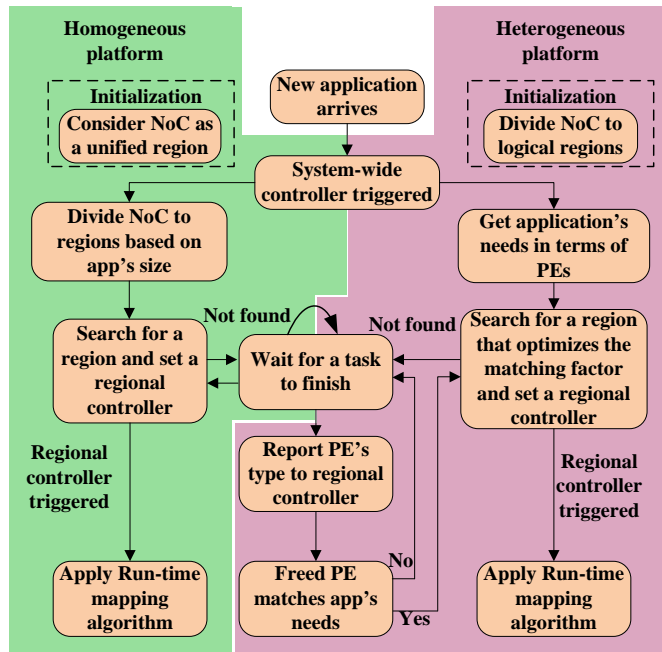


Figure 4.4: Flow of our D&C methodology.

An overview of our methodology framework is presented in Figure 4.4. Once a new application arrives the system-wide controller is invoked, the so called “Emperor” task is triggered, and according to the kind of many-core platform, the Homogeneous (Section 4.2.1.2) or the Heterogeneous (Section 4.2.1.3) flow is followed. The system-wide controller is a light-weight task that performs the initial steps of the run-time mapping. It is responsible for:

1. getting the application's requirements,
2. selecting an appropriate region to map that application on and
3. triggering other cores in the region (the regional controllers, the so called “Local Kings”, so as to perform the run-time mapping algorithm.

4.2.1.1 Definitions

An application task graph (ATG) is used to capture the traffic flow characteristics. The ATG $G(T, D)$ is a directed acyclic graph, where each vertex t_i represents a computational module in the application. $K[t] \forall t_i \in T$ specifies task's class type (e.g. logic task, computational task, memory task etc). Each directed arc $d_{i,j} \in D$ between tasks t_i and t_j characterizes data and communication dependencies. Each $d_{i,j}$ has an associated value $b(d_{i,j})$, which stands for the communication volume exchanged between tasks t_i and t_j .

A many-core platform topology and its communication infrastructure can be uniquely described by a strongly connected directed graph $A(I, N)$. The set of vertices N is composed of two mutually exclusive subsets N_{PE} and N_C containing the available platform's Processing Elements (PEs) and the platform's on-chip interconnection elements (such as routers in Network-on-Chip technology). $C[pe_i] \forall pe_i \in N_{PE}$ specifies the class of the PE pe_i . The set of edges I contains the interconnection information (both physical and virtual) for the N set.

The mapped cores define the M_{PE} set. We also define a mapping function $map : T \rightarrow N_{PE}$ that maps the application's task (T set) to the available PEs (N_{PE} set). Let the set of unmapped nodes $\overline{M_{PE}}$ such as $pe \in \overline{M_{PE}}$ if $pe \notin M_{PE}$. From our definition it follows that $M_{PE} \cap \overline{M_{PE}} = \emptyset$.

We define the set R which describes the logical regions on the platform.

R is composed of k ($k \geq 1$) subsets $R_1, R_2, \dots, R_i, \dots, R_k$ such that $\bigcap_{i=1}^k R_i = \emptyset$ and $\bigcup_{i=1}^k R_i = R$. $M_{R_i}[]$ is a list that defines the one to one result of the map mapping function in the R_i region. A region R_i is considered occupied iff $\exists pe_i \in R_i : pe_i \in M_{PE}$.

In an heterogeneous platform, the $C[pe_i]$ varies for each PE. Also, each t_i may require a special class of PE to run on, e.g. a DSP PE class. In order to comply with application's requirements and platform's resources and have the best $T \rightarrow N_{PE}$ correspondence, we define $\forall t_i \in T$ the parameter Matching Factor (MF) such that:

$$1 \geq \frac{C[pe_i]}{K[t_i]} \geq MF_{t_i}, \forall pe_i \in N_{PE} \quad (4.1)$$

MF is a designer specified parameter and defines the classes of PEs that the t_i can sit on. MF implies how good the class $C[pe_i]$ of pe_i element matches the specific t_i task and it defines a priority type on which core the task should be mapped on first. Different values and different decisions for MF result to different $M_{R_i}[]$ lists. In a homogeneous platform $MF_{t_1} = \dots = MF_{t_N}, \forall t_i \in T$ because $C[pe_1] = \dots = C[pe_N], \forall pe \in N_{pe}$.

4.2.1.2 Homogeneous Platform

The starting point of the methodology is the annotated graphs of the many-core platform $A(I, N)$ and of the application(s) $G(T, D)$ to be mapped. The mapping algorithm utilizes this information and proposes the solution without violating the bandwidth constraints of the platform and the requirements of the application(s). The mapping procedure is presented in Algorithm 1.

- Step 1 (lines 1-7): In the first step, we check the total number of tasks $|T|$. If platform is capable for serving the application a new region R_i is created and a signal is sent to a core inside the R_i region. This core plays the role of the regional controller (“Local King”) and it performs the run-time mapping algorithm. If this is not the case, we wait for a task to finish and free its PE. Due to fact that the set R is composed of mutually exclusive subsets, pe_i cannot belong to other subsets of R . If the new application fits in the platform, the System-Wide Controller appoints a tile on the NoC to be a Regional Controller, and sends the request to it. This Regional Controller is responsible for the region consisting of the unoccupied tiles around it, whose Manhattan distance from the controller is less or equal to the *search_distance* value, which is defined in Equation 4.2. The *search_distance* is not a fixed value and may be increased to include more unoccupied tiles.

$$search_distance = \begin{cases} \sqrt{|T|}, & |T| < 9 \\ \sqrt{|T|} - 1, & |T| \geq 9 \end{cases} \quad (4.2)$$

Algorithm 1 Homogeneous platform

```

    // Step 1: Check availability
1: If  $|T| \leq |\overline{M_{PE}}|$ 
2:   define new  $R_i \in R | \forall pe_i \in R_i, pe_i \in \overline{M_{PE}}$ 
3:   signal( $R_i$ )
4:   jump(Step 2)
5: Else
6:   wait() // for a task to release its PE
7:   jump(Stage 1)
    // Step 2: Run time mapping procedure
8:  $\forall d_{i,j} \in D$ 
9:    $\forall pe_i \in R_i$ 
10:     $src = \min\{F_{HOM}(d_i, pe_i)\}$  // equation 4.3
11:     $dst = \min\{F_{HOM}(d_j, pe_i)\}$ 
12:     $M_{PE}, M_{R_i}[] \leftarrow src$ 
13:     $M_{PE}, M_{R_i}[] \leftarrow dst$ 
    // Step 3: Swapping procedure
14:  $bestCost = bwCost\{M_{R_i}[]\}$  // equation 4.4
15:  $\forall t_i \in R_i$ 
16:    $\forall t_j \in R_i, t_j \neq t_i$ 
17:   If  $(MD(t_i, t_j) \leq MAX\_MANH\_DST)$ 
18:      $swap(t_i, t_j)$ 
19:      $tmpCost = bwCost\{M_{R_i}[]\}$ 
20:     If  $tmpCost < bestCost$ 
21:        $bestCost = tmpCost$ 
22:        $M_{R_i}[] \leftarrow \text{new } M_{R_i}[]$ 
23:     Else
24:        $swap(t_i, t_j)$ 

```

- Step 2 (lines 8-13): For every communication flow $(d_{i,j})$ in G , we find for the source (i) and the destination (j) the *min* value of cost function 4.3 for the selected PEs.

$$F_{HOM} = \sum_j (b(d_{i,j}) + MD_{i,j}) + \sum_i \sum_j b(d_{i,j}) \times MD_{i,j} \quad (4.3)$$

where $MD_{i,j}$ is the distance (measured in hops) between pe_i and pe_j . This cost function combines the communication cost of the neighborhood of pe_i (first term) and the total communication cost of the platform (second term).

- Step 3 (lines 14-24): After the initial mapping has been performed we employ an iterative application node swapping process (similar to the one used in [61]) trying to further reduce the total communication cost. During this process a pair of application nodes (mapped on platform nodes) is chosen and their position on the platform is swapped. After each swap the total communication cost (equation 4.4) is evaluated and if it is smaller than the previous value the swap is kept, otherwise the swap is not valid. The number of iterations of this swapping procedure is defined by the value MAX_MANH_DST .

$$bwCost = \sum_{M_{R_i}} b(d_{i,j}) * (MD_{i,j}) \quad (4.4)$$

4.2.1.3 Heterogeneous Platform

Computing a mapping for a heterogeneous platform is more complex than the one for homogeneous platforms. This is due to the fact that the calculation of the most efficient Processing Element type for each task is needed, followed by the computation of a mapping that respects these preferences to types and in the same time minimizes the communication energy of any application's execution.

The starting point of the methodology for heterogeneous platforms is the annotated graphs of the many-core platform $A(I, N)$, of the application(s) $G(T, D)$ to be mapped on and the designer specified parameter MF . The algorithm searches for regions able to serve application's task

Algorithm 2 Heterogeneous platform

```

// Step 1: Region selection step
1:  $\forall t_i \in T$ 
2:    $\forall pe_i \in N_{PE}$ 
3:   sort $\{MF_{t_i}\}$ 
4:  $\forall R_i \in R$ 
5:   If  $(|T| \leq |\overline{M_{PE}}|)$  &&  $(\forall t_i \in T, \exists pe_i \in \overline{M_{R_i}} : \frac{C[pe_i]}{K[t_i]} = 1)$ 
6:     select $(R_i)$ 
7:     jump(Step 5)
8:
// Step 2: if the first matching doesn't yield a result
9:  $\forall R_i \in R$ 
10:  If  $(|T| \leq |\overline{M_{PE}}|)$  &&  $(\forall t_i \in T, \exists pe_i \in \overline{M_{R_i}} : 1 > \frac{C[pe_i]}{K[t_i]} \geq MF_{t_i})$ 
11:    select $(R_i)$ 
12:    jump(Step 5)
13:
// Step 3: no matching region has been found
14:  $\forall$  unoccupied  $R_i \in R$ 
15:    $\forall pe_i \in \overline{M_{PE}}, pe_i \notin R_i$ 
16:    $\{R_i\} = \{R_i\} + pe_i$ 
17:   repeat(Steps 1-2) for  $R_i$ 
18:   If  $R_i$  not selected
19:      $\{R_i\} = \{R_i\} - pe_i$ , restore $(R_i)$ 
20:
// Step 4: no region was found, or all regions are occupied
21: define new  $R_i = \emptyset \in R$ 
22:  $\forall pe_i \in \overline{M_{PE}}$ 
23:    $\{R_i\} = \{R_i\} + pe_i$ 
24: repeat(Steps 1-2) for  $R_i$ 
25: If  $R_i$  not selected
26:    $\{R_i\} = \{R_i\} - pe_i$ , restore $(R_i)$ 
27: wait() // for a task to release its PE
28: jump(Step 1)
// Step 5: Run time mapping procedure
29:  $\forall K[t_k] \in G$ 
30:    $\{S\} = d_{i,j}$  iff  $(K[t_k] = K[t_i]) \vee (K[t_k] = T[t_j])$ 
31: sort $(S)$  //by  $b(d_{i,j})$ 
32:  $\forall d_{i,j} \in S$ 
33:    $\forall pe_i \in R_i$ 
34:    $src = \min\{F_{HET}(d_i, pe_i)\}$  // equation 4.5
35:    $dst = \min\{F_{HET}(d_j, pe_i)\}$ 
36:    $M_{PE}, M_{R_i}[] \leftarrow src$ 
37:    $M_{PE}, M_{R_i}[] \leftarrow dst$ 
// Step 6: Swapping procedure
38:  $bestCost = bwCost\{M_{R_i}[]\}$  // equation 4.4
39:  $\forall t_i \in R_i$ 
40:    $\forall t_j \in R_i, t_j \neq t_i$ 
41:   If  $(MD(t_i, t_j) \leq MAX\_MANH\_DST)$ 
42:     swap $(t_i, t_j)$ 
43:      $tmpCost = bwCost\{M_{R_i}[]\}$ 
44:     If  $tmpCost < bestCost$ 
45:        $bestCost = tmpCost$ 
46:        $M_{R_i}[] \leftarrow new M_{R_i}[]$ 
47:     Else
48:       swap $(t_i, t_j)$ 

```

as best as possible in terms of available classes. The algorithm for the heterogeneous platforms is presented in Algorithm 2.

- Step 1 (lines 1-8): In the first step, we try to find a region in which all cores' classes match perfect with all application's tasks ($\frac{C[pe_i]}{K[t_i]} = 1$). If such region exists, a signal is sent to a core inside the region in order to perform the distributed run-time mapping algorithm.
- Step 2 (lines 9-13): If the first matching does not yield a result and application's requirements are not so strict ($MF < 1$) we change the binding according to MF_{t_i} and try to find regions that are as close as possible to the required $MF_{t_i} \forall t_i \in T$. If such a region exists this region is selected and the mapping algorithm is performed.
- Step 3 (lines 14-20): If still no matching region has been found, we search for any unoccupied R_i and we add to that R_i any $pe_i \in \overline{M_{PE}}$. If the new region R'_i is not able to serve the application, all PEs that were previously attached, they are now restored to their previous regions.
- Step 4 (lines 21-28): If still no matching region has been found or all regions are occupied, a new region R_i is created and any $pe_i \in \overline{M_{PE}}$ is added to that region. And in this case if the new region R_i is not able to serve the application, all PEs that were attached, they are now restored to their previous regions and we wait for a task to finish and free its PE.
- Step 5 (lines 29-37): In this step, we define the set S that contains all the flows $d_{i,j}$ whose either source's class ($K[t_i]$) or destination's class ($K[t_j]$) is bound to $K[t_k]$. This set is then sorted by $b(d_{i,j})$. We sort flows by bandwidth requirements as it helps in reducing bandwidth fragmentation and it important from a resource conservation perspective since the benefits of a shorter path grows with communication demands. Then, for every communication flow ($d_{i,j}$) in S , we find for the source (i) and the destination (j) the *min* value of cost function 4.5 for the selected pe_i .

$$F_{HET} = F_{HOM} + Q(C[pe_i]) \quad (4.5)$$

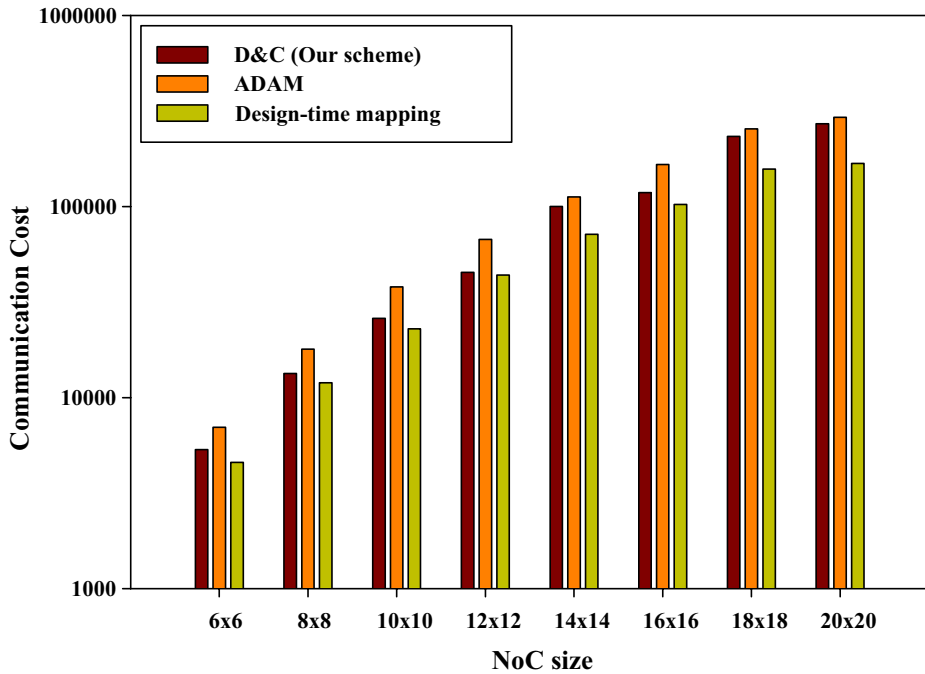


Figure 4.5: Communication Cost comparison in homogeneous platforms to ADAM [9] and design-time mapping [46].

where $Q(C[pe_i])$ is the requirements of class $C[pe_i]$ in terms of execution utilization and defines the PE's utilization by the task to be mapped on.

- Step 6 (lines 38-48): After the initial mapping has been performed we try to further reduce the total communication cost by employing the swapping technique. We swap a pair of mapped nodes and after each swap the total communication cost (equation 4.4) is evaluated. If it is better than the current communication cost the swap remains, otherwise, we restore it back.

4.2.2 Experimental results

We have performed extensive simulations of the behavior of several applications (a) MPEG-4, (b) Multi-Window Display (MWD) [17], (b) Picture-In-Picture (PIP) [17] (d) MultiMedia System (MMS) [46], (e)

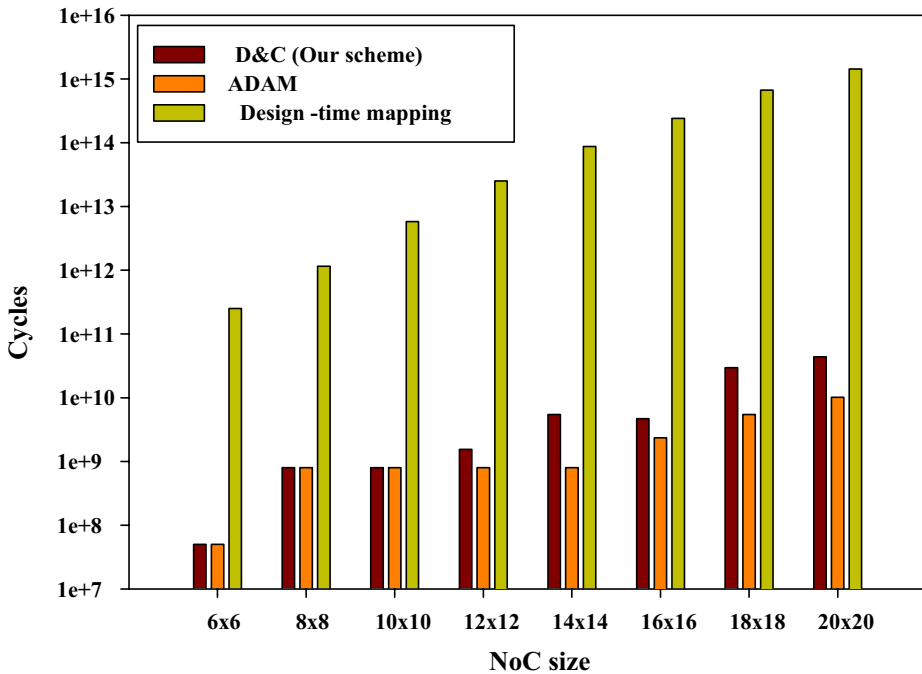


Figure 4.6: Mapping computational effort in homogeneous platforms in comparison with ADAM [9] and design-time mapping [46].

Digitale Radio Mondiale (DRM) [79] and (f) applications from TGFF [35] to validate our approach. An architecture that is able to accommodate a high number of cores, satisfying the need for high on-chip communication and data transfers, is the Network-on-Chip (NoC) architecture. The Network-on-Chip (NoC) model is emerging as a revolutionary architecture in solving the performance limitations arising out of long interconnects, outperforming more mainstream bus architectures and as a very good architecture template for many-core platforms.

In Figures 4.5 and 4.6 we compare for various homogeneous platforms, using TGFF application graphs, our D&C approach to the state-of-the-art distributed run-time mapping algorithm [9] and to an exhaustive design-time mapping [46], in terms of on-chip communication cost. The on-chip communication cost is a part of the mapping cost function for all these three mapping algorithms. Figure 4.5 shows that our D&C method achieves on average 21% better result in terms of final on-

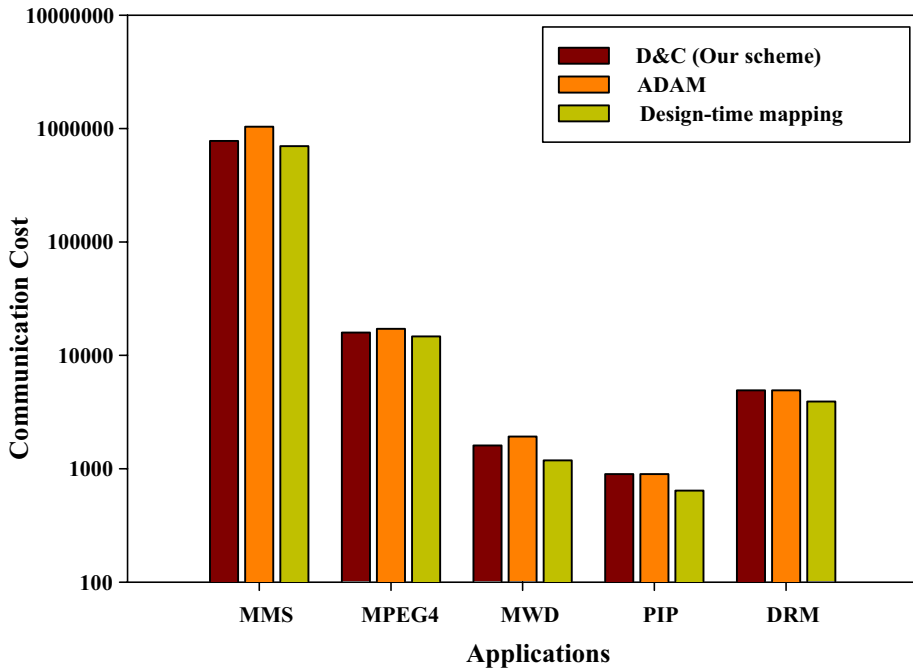


Figure 4.7: Communication cost comparison for the five selected applications with ADAM [9] and design-time mapping [46].

chip communication cost compared to the run-time algorithm presented in [9]. Although the optimal result is achieved by the exhaustive design-time mapping algorithm, our D&C method requires significant less cycles (improved scalability) in order to make the mapping decisions, as shown in Figure 4.6.

For the validation of our approach on heterogeneous platforms we used the platform presented in Section 3.2. The platform is composed of Processor-Memory (PM) nodes interconnected via a packet-switched mesh network. A node can also be a memory node without a processor, pure logic or an interface node to off-chip memory. Each PM node contains a LEON3 processor, hardware modules connected to the local bus, and a local memory. The system uses a virtual-to-physical translation and all shared memories are globally visible to all nodes and organized as a single virtual addressing space. The communication of cores inside the platform is done using message-passing instructions and by using the shared memory interface. Whenever there is a need

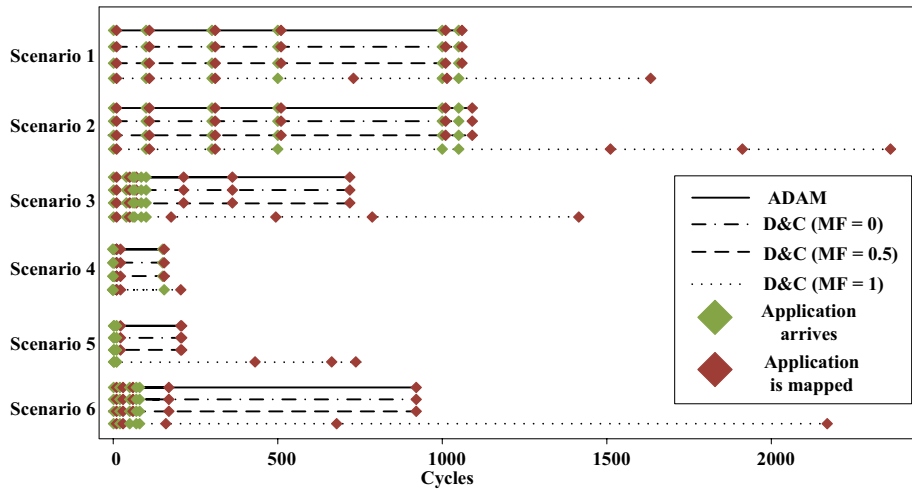


Figure 4.8: Run-time mapping scenarios on an heterogeneous many-core platform compared with ADAM [9]

for the system-wide controller to trigger another core, the hardware’s synchronization safe-lock memory mechanism is used. Shared memory environment allows the ease use of such mechanisms. The lock is acquired by the system-wide controller and it propagates information to shared memory. Then the lock is freed and the region controller loads the data from the memory and performs the required mapping operations. The execution of code on a regional controller is also possible with the usage of message passing instructions.

Figure 4.7 presents a comparison of the on-chip communication cost for the five selected applications. We compare the on-chip communication cost of our D&C approach ADAM and with the exhaustive design-time mapping algorithm [46]. As Figure 4.7 depicts, our proposed algorithm has on average 10% better result in terms of on-chip communication that the ADAM run-time mapping algorithm. However, the best result is achieved, as expected, by the exhaustive design-time mapping algorithm. The cycles required for the result extraction are the same for both the run-time algorithms but for the exhaustive one is $100\times$ bigger.

Several run-time scenarios were built on the NoC platform. The difference between the different scenarios is the number of applications and

Table 4.1: Utilization of platform's resources

	D&C ($MF = 1$)	D&C ($MF = 0.5$)	D&C ($MF = 0$)	ADAM [9]
Scenario 1	100%	92%	92%	91%
Scenario 2	100%	88%	88%	87%
Scenario 3	100%	87%	87%	88%
Scenario 4	100%	86%	86%	84%
Scenario 5	100%	78%	78%	79%
Scenario 6	100%	87%	87%	88%

the arrival time of each one of them. The arrival time was randomly generated. We compared our D&C scheme, using different matching factor values, with the ADAM [9] one. Figure 4.8 depicts all the implemented scenarios. The green diamond represents the arrival time of the application while the red one represents the time that the mapping result was taken. Three MF values are chosen to match what is provided by the many-core platform [10, 29]. The picture shows that both our D&C scheme (with $MF = 0$ and $MF = 0.5$) has the same run-time behavior with the ADAM approach. D&C scheme with $MF = 1$ has a different behavior because under the $MF = 1$ restriction a task can be mapped only on a core that has the same class type with the task. In this case, the algorithm takes more time because it waits for desired cores to be freed after finishing their tasks. D&C scheme with $MF = 1$ has the best task to core mapping decision resulting to best platform's resources utilization as depicted in Table 4.1. Table 4.1 shows that with $MF = 1$, we can have 100% utilization of platform resources at run-time with a penalty cost at performance. If our application needs are not so strict we can choose other values of matching factor, thus relaxing the strictness of the matching.

4.2.3 Conclusions

In this chapter, a D&C based distributed run-time application mapping framework for both homogeneous and heterogeneous many-core platforms is presented. The framework adapts to application's needs and application's execution restrictions by using the matching factor parameter and produces on average 21% and 10% better on-chip com-

munication cost for homogeneous and heterogeneous platforms respectively, compared to the ADAM [9] scheme with almost the same computational effort. The random implemented run-time scenarios showed that our algorithm can have different behavior according to the selected matching factor and resulting to different platform's resources utilization.

4.3 Distributed run-time resource management for malleable applications

The run-time resource management paradigm has been revealed as a key challenge to modern multi-core systems and it has become prominent due to the run-time dynamicity of modern parallel applications and platforms. In modern execution environments run-time resource availability may vary due to system dynamism as resources can be added or removed from such environments at any time. According to the parallel job classification scheme presented in [37] we can separate parallel applications into three categories based on their characteristics:

- **Moldable applications:** Parallel applications that can be stopped at any point but the number of processors for such jobs cannot be changed during run-time.
- **Malleable applications:** These are parallel applications that can be stopped at any point of execution and have flexibility to change the number of assigned processors during run-time. These applications are also called reconfigurable applications.
- **Migratable applications:** These are parallel applications that can be stopped at any point of execution and can be restarted on processors in a different site, cluster or domain.

As described in [50] malleability is used for autonomous application reconfiguration in response to dynamic changes in platform's resource availability, thus allowing applications to optimize the use of platform's features (e.g., number of processors). In other words, malleable applications use varying amounts of platform resources during their execution

and they may specify the minimum and maximum number of processors they require. As minimum can be considered the minimum number of processors a malleable job needs while maximum describes the maximum one. Any allocation of cores more than the maximum number is a waste of platform resources.

4.3.1 Methodology Framework

The goal of the proposed methodology framework is to perform run-time resource management on many-core platforms, both homogeneous and heterogeneous, for parallel applications in a distributed way. The proposed framework is designed for malleable applications. Even though it can support both moldable and migratable ones, it best utilizes the platform available resources for malleable applications. In this work each core can have one of the following roles concerning the resource management of the platform: (i) *initial core*; (ii) *controller core*; and (iii) *manager core*.

The *initial core* is randomly chosen and triggered when a new application arrives in the system. Its purpose is to find the initial set of cores that the application will start running on. It requests cores from controller and manager cores, gathers their offers and determines if at least one core had been offered. An initial core cannot handle two applications at the same time. Therefore, if it is chosen to initialize an application while it is already initializing one, the second one will be stalled until the initialization of the first one is over.

The *controller core* is responsible for handling all the unoccupied cores of a predefined region of the platform. It is defined at the initialization of the platform and cannot be changed at run-time. At the same time, the controller core also maintains a list of all manager cores that occupies a core in its region. This is an essential part of our distributed design since the information of the occupied cores inside a region is not maintained in a central point but is scattered throughout the platform. This information is provided to initial or manager cores if and when it is asked for.

Last, the *manager core* manages an application searching for new cores and instructing the resizing of the application whenever it has more

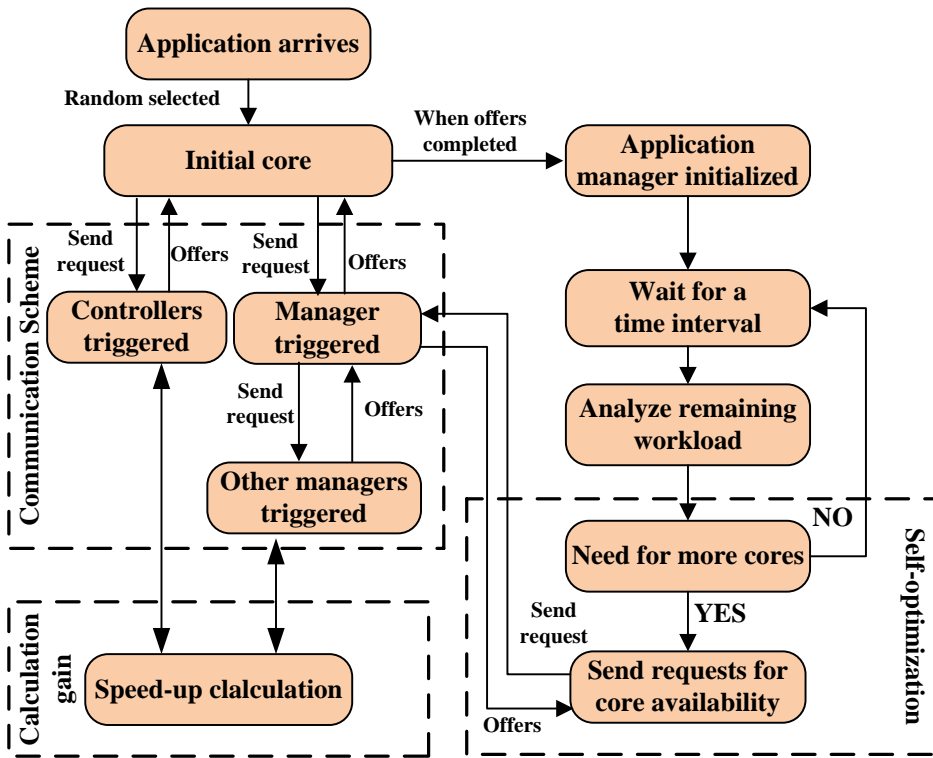


Figure 4.9: Overall flow of the proposed methodology.

or less cores to run on. This core does not execute any part of the application. If it does not possess any other cores for the actual application to run on, a self optimization is necessary in order to for the manager to acquire at least one working node. Although an initial core can be a manager one and vice versa, a controller core cannot change its functionality.

An overview of our methodology framework in terms of node intercommunication is presented in Figure 4.9. Once a new application arrives on a random core (this is the initial core), this core sends messages to controller cores found near it and asks for an available core to serve as the actual application manager. Then the controller cores search into their area for any unoccupied cores and also send requests to any managers into this area. According to the application's characteristics, cores with the appropriate type respond, provided that the speed-up

gain of the requesting application is greater than the speed-up loss of the offering manager. After that, the initial core receives all offers and determines the new manager which is initialized by a signal. Then, the new manager distributes evenly the workload to the working node he manages. After a predefined time interval the manager, if there is a need and the application is not close to its finish, requests for more cores in order to increase application's performance giving to the framework a self-optimization aspect.

4.3.1.1 Definitions

The application and the speed-up model used in this work is the common malleable application model described in [31, 36] and used by other distributed approaches [50]. Each application is described by four parameters W , var , A and Q , where W is the workload, var is the parallelism variance, A is the average parallelism and Q is most preferred Processing Element (PE) type that the application is supposed to be executed on. A many-core platform topology and its communication infrastructure can be uniquely described by a strongly connected directed graph $P(I, N)$. The set of vertices N is composed of two mutually exclusive subsets N_{PE} and N_C containing the available platform's PEs and the platform's on-chip interconnection elements (C), such as routers in an NoC technology. Each platform's PE can be of a specific type and differ from the other platform types (supporting heterogeneous platforms) or all PEs can be of the same type and thus have the same functionality (homogeneous platform). In our framework $T_{pe_i} \forall pe_i \in N_{PE}$ specifies the type of the PE pe_i . In an heterogeneous platform, the T_{pe_i} varies for each PE while in an homogeneous platform T_{pe_i} is the same for all PEs. In order to comply with the application's requirements in terms of required PE classes and have the best $Q \rightarrow T_{pe_i}$ utilization, we define $Util[pe_i] \in [0, 1]$ that implies how good the $app(W, var, A, Q)$ is served by the pe_i with T_{pe_i} type. $Util[pe_i]$ can also be considered as a priority factor when an application is asking for additional cores, meaning that when $Util[pe_i] = 1$ we want the best match while when $Util[pe_i] = 0$ the application is not requesting any specific T_{pe_i} . Last, we define the sets F and $offers[]$, which describe all the nodes that can appropriately serve an application based on their type and all the cores offered to the application respectively.

4.3.1.2 Communication Scheme

In order to be consistent with the rest of the document and with Algorithms 3 and 4 we declare that the index dst specifies the manager core that is requesting more resources while src is the manager core that is offering them. Also, we define the set R which contains, for each controller core, the PEs in a manhattan distance specified by the equation 4.6 where $size$ is the size of the platform and $num_controllers$ is the number of controllers cores on it on the X dimension.

$$distance = size/num_controllers_X \quad (4.6)$$

Algorithm 3 Communication scheme algorithm

```

// Initial core actions
1: analyze(W, var, A, Q)
2: req_send(control[], core_id, app, R)
3: start_timer()
4: offers[] = receive_offers
5: end_timer()
6: sel_pe = best{offers[]}
7: initialize_manager(sel_pe, offer, R)

// Controller core actions
8: analyze(W, var, A, Q, R)
9: for each (NPE ∈  $\bar{F}$  && NPE ∈ R)
10:   If calculate(gain(app)) > 0 // Algorithm 4
11:     offers[] = offers[] + new_offer
12:   send_offers(offers, core_id)

// Manager core actions
13: // Actions for offering cores
14: analyze(W, var, A, Q, R)
15: If calculate(gain(app)) > 0 // Algorithm 4
16:   offers[] = offers[] + new_offer
17:   send_offers(offers, core_id)
18: // Actions for self-optimization
19: while (app(W, var, A, Q) != finished) {
20:   analyze(W, var, A, Q, offers)
21:   timer()
22:   If ((app.threshold = max) || (app.left_time < timer()))
23:     continue
24:   else
25:     req_send(control[], R)
26:     start_timer()
27:     offers[] = offers[] + receive_offers
28:     end_timer()
29: end while

```

As aforementioned, when a new application arrives on a core, the initial core task is executed and the communication inside the platform takes place in order to establish a manager core for the application. The communication between initial, controller and manager cores is described in Algorithm 3.

Initial core (lines 1-7): When a new application arrives on an initial core (Figure 4.10a), this core analyzes the application's characteristics, sends a message to the controllers and managers (Figure 4.10b) that are inside its region R and fires a timer in order to check for their responses. After the end of the timer, the initial core selects the best offer and sends a signal, according to the offer, and starts the initialization of the manager that will handle the application (Figure 4.10c).

Controller core (lines 8-12): A controller cores has a variety of responsibilities. Besides maintaining regional information about the managers existing on its region, it has to provide this information to any cores requesting it. It also informs these cores about the position of controller cores in other regions. When the controller receives the signal form the initial core, it analyzes the application and starts to find cores to offer. The controller core can offer any unoccupied core he owns, inside the region R of the initial core, provided that it serves the application's characteristics.

Manager core (lines 13-29): The manager core has two tasks. During the first one, the manager checks if it can offer a core to the new application without loosing more in terms of application speed-up than the gain that the new application will have with the new addition. The second task has to do with the self-optimization process of the already running application. Specifically, there is a time threshold in which the manager checks if the application has taken all the necessary resources it needs or it is near to its completion. If the application has maximized its speed-up [36] there is no need to search for more cores. The same happens if the application is almost finished. In other words, if the remaining time of the application is less than the time interval there is no need to search for more cores. Otherwise, the application enters a self-optimization phase and the manager follows the same communication scheme and sends a message to the controllers (Figure 4.10d) that are inside its region R and fires a timer in order to check for their responses. After the end of the timer, the manager core checks the offers

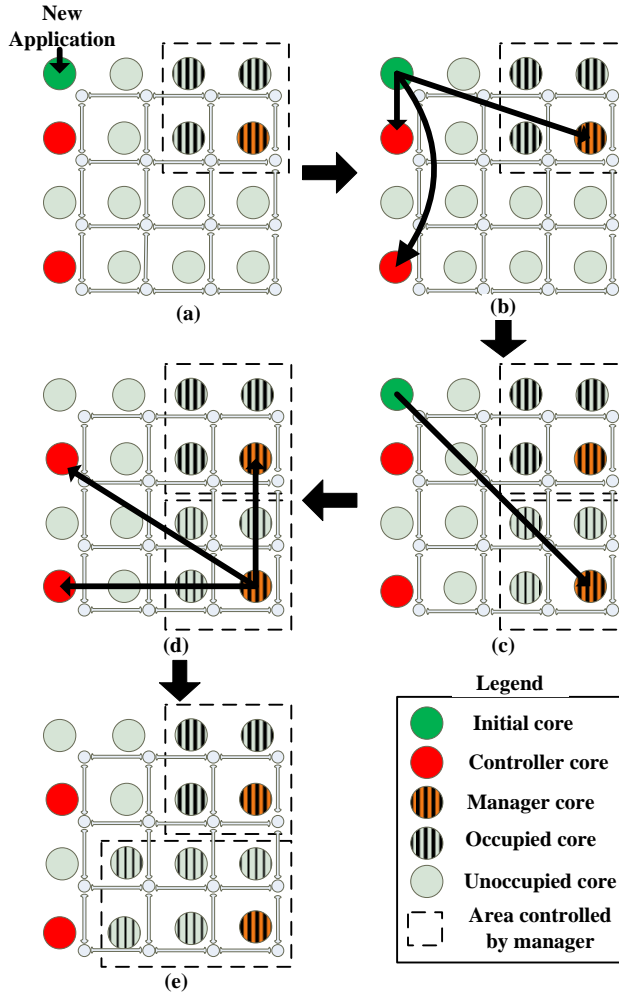


Figure 4.10: Example of the communication scheme.

form the controller cores or from any other manager cores and starts the resize of the application (Figure 4.10e). Both the controller and manager cores use a function (lines 10 and 15 respectively) in order to calculate the gain or the loss to the application speed-up when offering a core (Algorithm 4).

4.3.1.3 Gain calculation

Algorithm 4 describes the required steps that both the controller and manager cores take in order to decide which cores should be offered when an application starts its self-optimization process asking for cores. It has three discrete parts: (i) the actions regarding the calculation of the speedup of the destination node requesting application (lines 4-8); (ii) the actions regarding the calculation of the speedup of the source node offering application (lines 9-13); (iii) and the calculation of the final total gain of this core trade (lines 14-21). During the actions regarding the destination node, for each $N_{PE} \in ((PE_{src} \cap R \cap F) - offers[])$ we calculate the speed-up of the application taking into account the core utilization ($Util[N_{PE}]$) in order to offer to the application the best choices in terms of PE type. Once the speed-up is calculated we check whether the gain of adding the new core to our working set results to an overall gain for the application. On the other side, the actions regarding the source node check whether the loss of a core results in a bigger performance degradation on an already running application. In order to verify it, we calculate the loss speed-up both in terms of performance power and in terms of the $Util[N_{PE}]$ that are occupied and are needed by the application. Since both destination and source actions are greedy, the source offers cores to the destination only when the gain of the destination is bigger than the loss of the source.

4.3.1.4 Self-optimization process

An important part of our framework is that the manager core can decide to look for more cores in order to increase the number of his working cores. This procedure is called self-optimization process and its goal is to increase further the speedup of an already running application. All manager cores can initiate this process resulting in a better sharing be-

Algorithm 4 Gain calculation algorithm

```

1:  $offers[] = \emptyset$ 
2: while ( $gain > 0$ ) {
3:   for each  $N_{PE} \in ((PE_{src} \cap R \cap F) - offers[])$  {
4:     // Actions regarding the destination node
5:      $PE_{dst} = PE_{dst} \cup N_{PE} \cup offers[]$ 
6:      $ord\_PE_{dst} = order\{PE_{dst}\}$ 
7:     for  $pos = 1$  to  $ord\_PE_{dst}.length()$  {
8:        $SP_{dst} = Util\{ord\_PE_{dst}[pos]\} * (SP[pos] - SP[pos - 1])$ 
9:        $gain_{dst} = SP_{dst} - previous\_SP_{dst}$ 
10:    // Actions regarding the source node
11:     $PE_{src} = PE_{src} - offers[] - N_{PE}$ 
12:     $ord\_PE_{src} = order\{PE_{src}\}$ 
13:    for  $pos = 1$  to  $ord\_PE_{src}.length()$  {
14:       $SP_{src} = Util\{ord\_PE_{src}[pos]\} * (SP[pos] - SP[pos - 1])$ 
15:       $loss_{src} = previous\_SP_{src} - SP_{src}$ 
16:    // Calculate total gain
17:     $gain\_temp = gain_{dst} - loss_{src}$ 
18:    if ( $(gain\_temp > gain) \parallel ((gain\_temp = gain) \ \&\& \ D(manger, N_{PE}) <$ 
19:       $D(manger, prev\_N_{PE}))$ )
20:       $gain = gain\_temp$ 
21:       $prev\_N_{PE} = N_{PE}$ 
22:    end for
23:    if ( $gain > 0$ )
24:       $offers[] = offers[] \cup N_{PE}$ 
25:    end While

```

tween them. For example, suppose that the first application that arrived in the system received a great amount of cores at an area because at that moment no other applications were active. However, when new applications arrive only a small subset of unoccupied cores will be available for them resulting in resource starvation. By using the self-optimization process, the newly arrived application will acquire some cores of the already running application in that way so as the gain of acquiring new cores will be greater than the loss in the already running application. Additionally, the resource availability of the platform changes dynamically as some applications come to an end. Since there is no central resource management scheme, the existing applications have to claim those new cores without having original information about their availability. This self-optimization process has also been used in other distributed approaches [50].

Finally, even if an application meets all the criteria for self-optimization it is not wise to constantly burden the platform with such a process due to the performance overhead. Thus, between two such processes, a time interval must pass by. The only case a self-optimization is imperative is

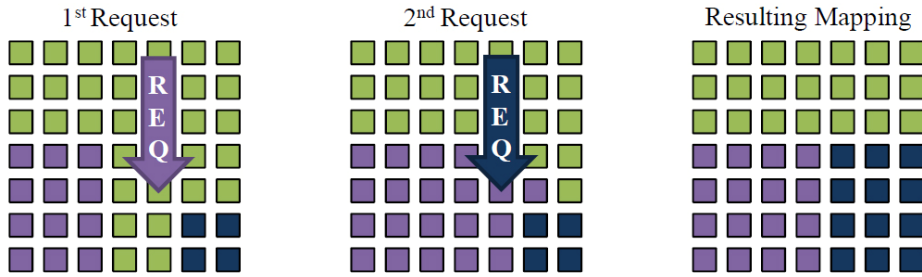


Figure 4.11: More fair resource allocation through self-optimization process [50]

when a manager has no working nodes.

4.3.2 Experimental Results

In order to validate our framework we have performed extensive simulation experiments in two steps: (i) using a C-based simulator (Chapter 4.3.2.1) and (ii) integrating the framework on the Intel Single Cloud Chip (SCC) many-core platform [45] (Chapter 4.3.3). In both cases, we compared the performance of the presented framework to the state-of-art distributed run-time resource manager DistRM [50]. As malleable applications input we have used the benchmarks provided by the parallel workload archive [37] and produced a file representing scenarios of applications arriving to our system. Each scenario consists of the time the application arrives on the system, its parameters as defined in section 4.3.1.1 and a random workload.

4.3.2.1 Evaluation on C simulator

Firstly, as aforementioned, we have developed a C-based simulator capable of simulating the behavior of malleable applications and all the necessary actions required by the on-chip communication scheme. For more accurate simulation, every node was represented by a different process. The inter-node communication is implemented using traditional Linux signals, while messages are passed using a pipe between the sender and the receiver. For synchronization semaphores

Table 4.2: Comparison of the proposed technique to the DistRM [50] in the C simulator.

	Msg. cnt.		Msg. size		Avg. sp.		Comp. eff.	
Plat. sizes	Number of applications							
	32	64	32	64	32	64	32	64
6x6	72.3	71.4	73.2	72.4	3.8	13.8	86.8	86.6
8x8	64.3	63.4	64.8	63.6	4.5	9.3	83.3	83.0
12x12	49.1	52.3	44.8	48.7	-1.4	1.5	66.0	68.3
16x16	42.6	45.8	40.0	41.4	0.5	3.1	61.3	64.9
20x20	32.6	36.1	27.7	30.5	3.1	2.7	53.4	57.8
24x24	25.1	27.2	18.5	19.2	2.0	2.4	50.1	51.7
28x28	20.6	22.3	13.5	14.1	1.5	2.8	42.7	48.7
32x32	17.9	14.6	9.9	2.5	1.6	2.8	41.9	42.4
Average	41%		37%		3%		62%	

are used. The goal of this step was to have a quick and abstract view of our methodology and estimate the cost of the developed distributed on-chip communication scheme. Also, the simulation method offers the capability of functional error correction and fast debugging in temps of possible communication deadlocks. The simulator supports big topologies (up to 32×32 core system), numerous application inputs.

Table 4.2 presents the results of the comparison of the proposed technique against the DistRM [50] distributed run-time manager. We evaluated the two run-time managers for various platform platform sizes, from 6×6 up to 32×32 , and for 32 and 64 applications. The comparison metrics are: (i) message count (*Msg. cnt.*), which is the total number of messages sent by all nodes during the whole duration of the simulation both for resource management and application execution; (ii) message size (*Msg. size*), which is the total size of sent messages; (iii) application average speed-up (*Avg. sp.*); and (iv) computational effort (*Comp. eff.*), which is the total number of speed-up function calls. Table 4.2 presents the *percentage gains* of the presented framework in comparison to DistRM. Concerning the message count, simulation results showed an average gain of 41% for the presented methodology due to the fact that the core request messages are sent only inside the area R while DistRM sends messages in many areas, smaller than R and probably overlap-

ping, thus increasing the number of messages used for sending requests and receiving answers. Also, the total size of these messages, measured in *bytes*, for our framework is on average 37% smaller than the size needed by DistRM. Thus, the network burden of our framework is on average 38% smaller. In terms of average application speed-up, the proposed framework achieves on average 3% better results than DistRM. The speed-up function used for this metric is the application speed-up function presented in [50]. Last, the gain of our methodology regarding the total computational effort is on average 62% compared to the DistRM.

4.3.3 Evaluation on Intel SCC platform

After the first evaluation of our framework in the C simulator, we integrated the whole framework as a run-time service on the many-core Intel SCC platform [45] in order to test our framework on a real platform and not only at the simulation level. The Single-Chip Cloud Computer (SCC) experimental processor [45] is a 48-core “concept vehicle” created by Intel Labs as a platform for many-core software research.

Intel SCC platform employs 48 cores interconnected using a NoC infrastructure. It features a well-known x86 processing element employed in each core. This is a significant feature, as explained in [58], since the Linux operating system and C, C++, FORTRAN compilers can be run on this platform, providing a run-time and programming environment which can be used by most programmers and giving the opportunity for other well-known and useful programs to be imported into the platform. An overview of the Intel SCC platform is depicted in Figure 4.12. The platform consists of:

- Two blocks, each with a P54C core (second generation Intel Pentium processor), 16 KB instruction and data L1 caches plus a unified 256 KB L2 cache.
- Mesh Interface Unit (MIU) with circuitry to allow the mesh and the interface to run at different frequencies.
- 16 KB Message Passing Buffer.

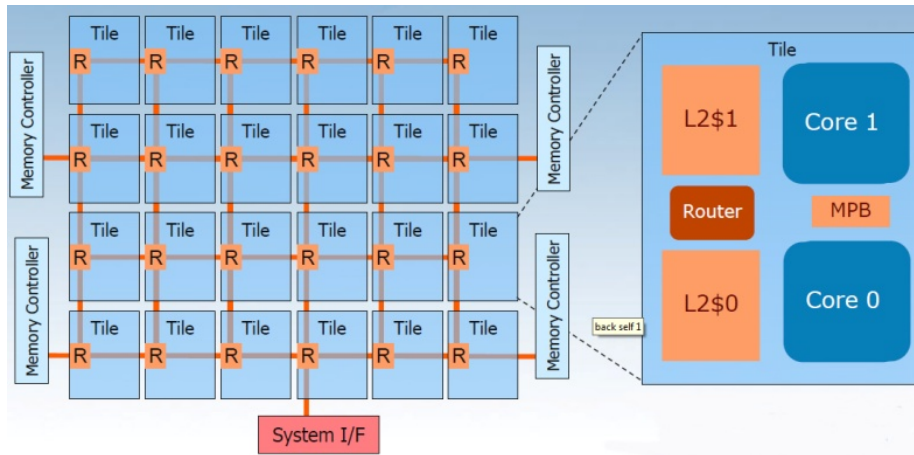


Figure 4.12: Overview of the Inter SCC Platform [81]

- Two test-and-set registers.

Each tile connects to a router. This router works with the Mesh Interface Unit (MIU) to integrate the tiles into a mesh. The MIU creates packets of data into the mesh and collects them from the mesh using a round-robin scheme to arbitrate between the two cores on the tile. The off-chip memory varies from 16 to 64 GB of DDR3 RAM, controlled by four memory controllers. Finally, router is connected to an off-package FPGA to translate the mesh protocol into the PCI express protocol, allowing the chip to interact with a PC serving as a management console. The entire memory architecture of the Intel SCC platform is illustrated in Figure 4.13

Each tile includes a message passing buffer (MPB) which provides a fast, on-die shared SRAM, as opposed to the bulk memory accessed through four DDR3 channels. While the processor does not offer any hardware-managed memory coherence, it features a new memory type to enable efficient communication between cores. This new memory type is called the Message Passing Buffer Type (MPBT). The size of the MPB is 16KB in each tile and its memory is shared among all the cores on the chip. With 24 tiles, the SCC provides 384KB of message passing buffer. When a program sends a message from one core to another, the MPB is used to propagate the message on the chip. However, data coherence and synchronization between cores is programmer's responsibility. In order

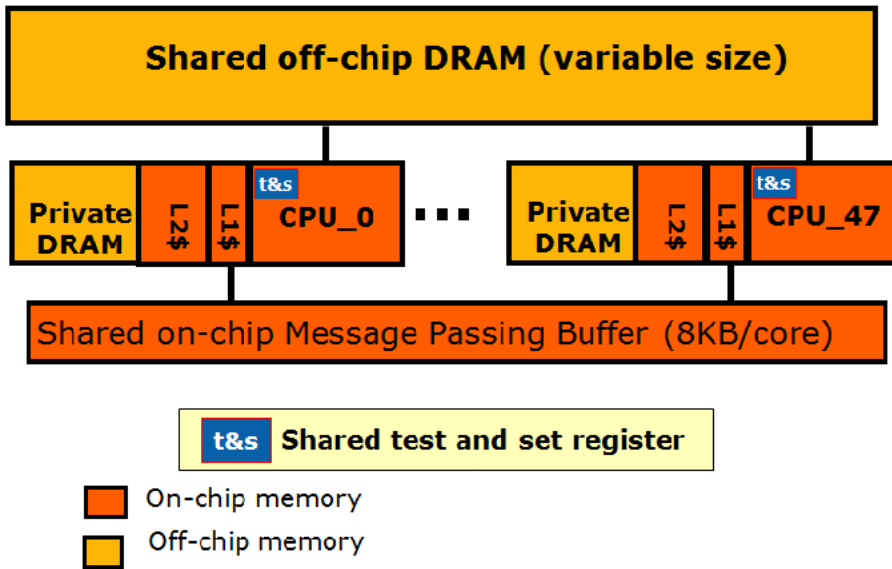


Figure 4.13: Memory architecture of the SCC processor [58]

to achieve mutually exclusive memory operations, a test&set register is built in every core.

In order to make programming easier and to increase the portability and scalability of programs written for the SCC platform, Intel provides a communication environment known as the RCCE. As a matter of fact, SCC and RCCE were developed side by side [81]. RCCE distributes evenly the MPB address space to the 48-cores, designating that each core will have 8KB of memory in this buffer for itself. It provides two basic interfaces for inter-node communication. The first is the *gory* one which is a low level design and offers the programmer greater control over the SCC in exchange of manual explicit synchronization. The first step of every memory operation is to allocate a memory space in MPB of MPBT data type. This is done with the `RCCE_malloc` function (Figure 4.14). The second interface is the *basic* one which employs `send` - `recv` functions for synchronization issues.

Since the platform size is fixed (48 cores) we compared the performance of our distributed run-time manager to the DistRM [50] for various number of applications running on the platform (from 8 to 64).

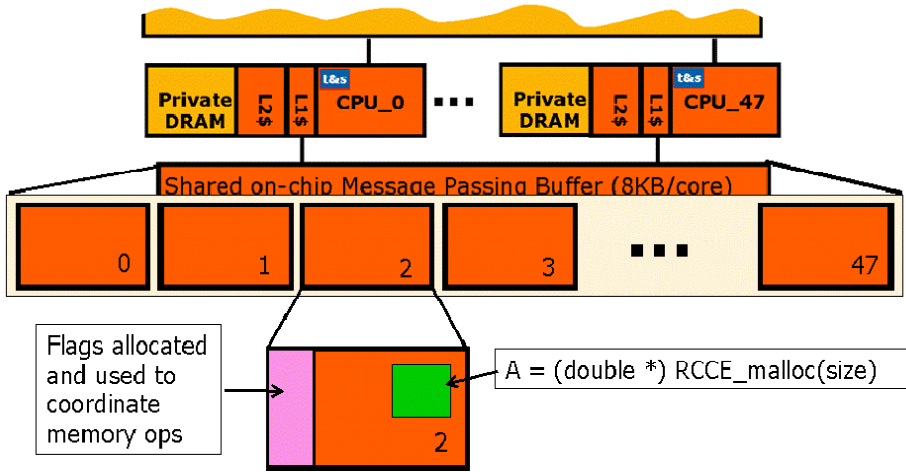


Figure 4.14: Symmetric name space model for the MPB as designed for RCCE library [58]

Figures 4.15 and 4.16 present the total number of messages sent by all nodes during the whole duration of the simulation and the total size of these messages in *bytes* respectively. The messages are used for application initialization, self-optimization and application resizing by manager cores and application execution. The proposed framework sends on average 70% less messages in order to perform all the necessary actions since the messages are sent only inside the area R . Whereas, DistRM searches for available cores in more areas, smaller than R , thus increasing the number of messages used for sending requests and receiving answers. Another reason that the presented algorithm has less messages than DistRM, is the fact that the framework performs application self-optimization under very specific criteria, only when the application has not maximized its speed-up or it is not near to its completion. Also, the size of the messages sent are on average 64% less than the ones used by the DistRM. The size is not proportional to the number of the messages since each message varies in size. For example, an offer message about four cores can have up to 20 *bytes* while the answer to this offer is 1 *byte*. Last, the proposed framework burdens network resources 66% less than DistRM.

Figure 4.17 presents the average application speed-up using the speed-

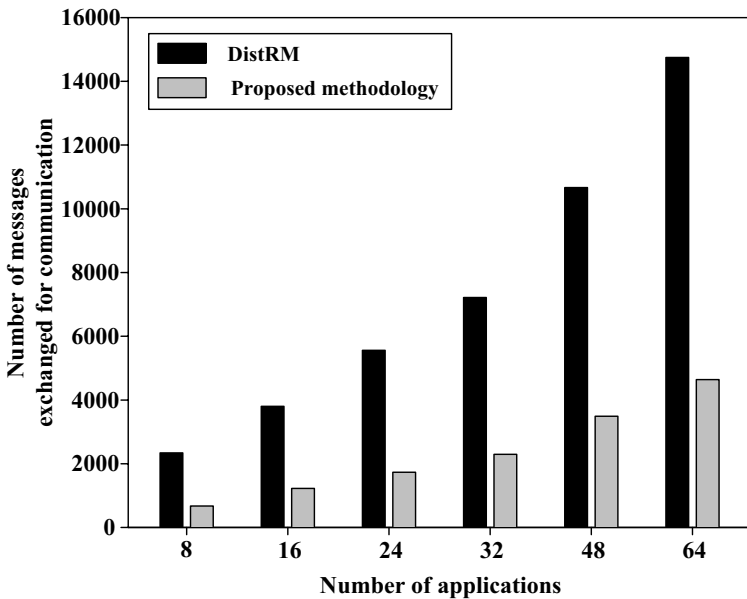


Figure 4.15: Total number of messages sent for intercommunication by all nodes for various applications compared with DistRM [50]

up function presented in [50]. Speed-up is defined as the ratio of the total number of turnarounds performed for all applications divided by the total workload. The presented framework achieves on average 20% better application speed-up than DistRM. This can be explained by the fact that in the presented framework cores are not disturbed so often by messages during their application execution and thus completing the applications faster. On the other hand, due to the large number of messages sent by the application agents in DistRM, cores stop their functionality more frequently in order to answer to these messages, thus delaying the execution.

Figure 4.18 shows the comparison of the computational effort between the presented framework and DistRM. Computational effort is defined as the total number of speed-up function calls during the whole simulation. The presented framework has on average 85% less speed-up function calls than DistRM. This can be explained by the fact that, as aforementioned, the presented framework performs less application self-optimizations due to specific criteria. So the manager cores calculate the

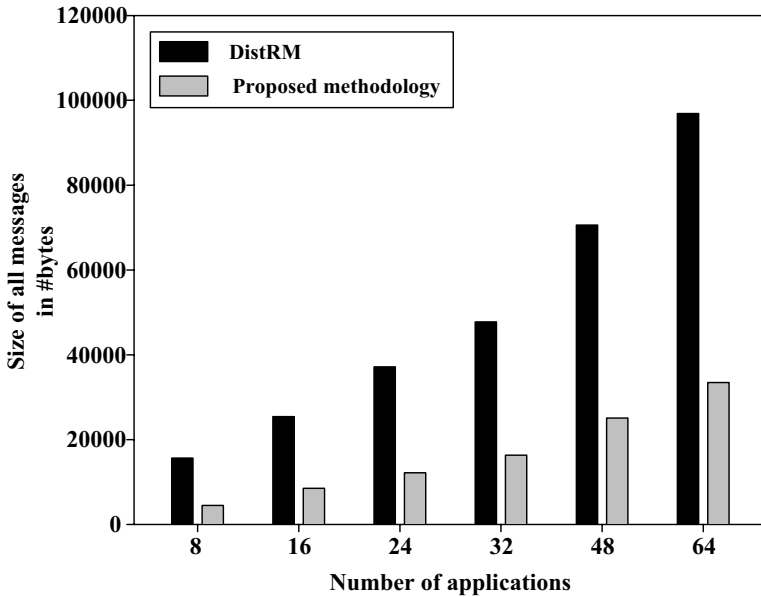


Figure 4.16: Total size of sent messages in *bytes* compared with DistRM [50]

speed-up function less frequently.

4.3.4 Conclusions

To sum up, we presented a distributed run-time manager for malleable applications. We coupled the concept of distributed computing with parallel applications and we present a workload-aware distributed run-time framework for malleable applications running on many-core platforms. The proposed framework is based on the idea of local controllers and managers while an on-chip intercommunication scheme ensures decision distribution. *The presented framework is responsible (i) for serving, at run-time, the needs of malleable applications, in terms of processing cores; (ii) makes sure that the application will get the optimum number of cores avoiding dominating effects; (iii) it takes into account the type of processors best utilizing any platform's heterogeneity; and (iv) it has a small overhead in overall core intercommunication.* The presented framework has a small communication overhead, takes into account platform's heterogeneity

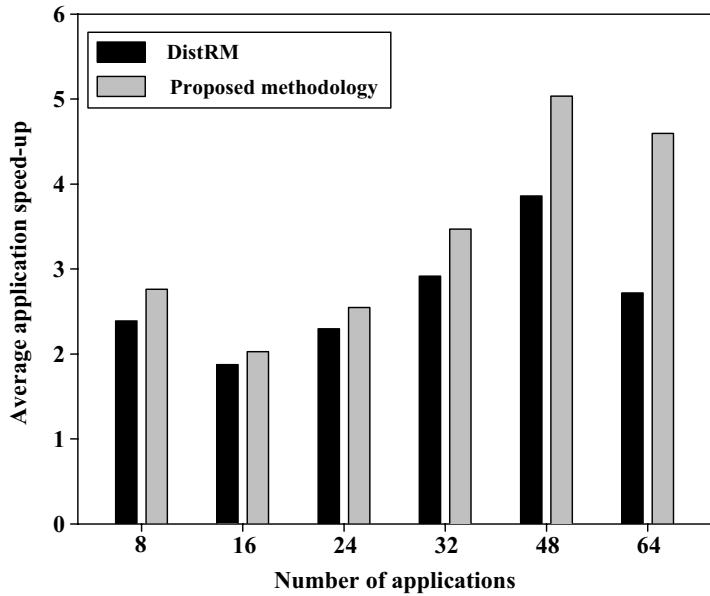


Figure 4.17: Average application speed-up using the speed-up function presented in [50].

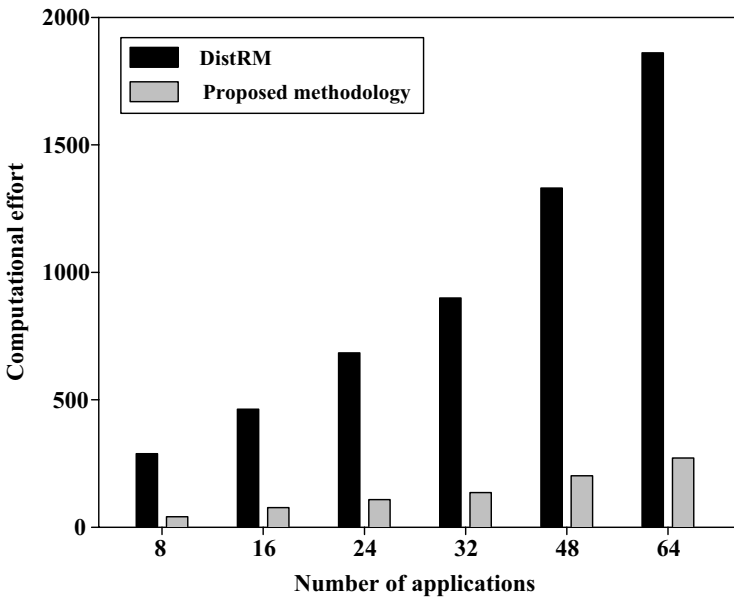


Figure 4.18: Computational effort comparison with DistRM [50]

and makes sure that the application will maximize its speed-up function. Our framework has been implemented as part of a C simulator and additionally as a run-time service on a real many-core platform and we compared it against the DistRM [50] state-of-art run-time resource manager. Experimental results showed that our framework has on average 70% less messages, 64% smaller message size and 20% application speed-up gain.

Chapter 5

High-level customization framework for resource management on NoC architectures

5.1 Introduction

As aforementioned, future integrated systems will contain billion of transistors [73], composing tens to hundreds of IP cores implementing emerging complex and demanding applications. An architecture able to accommodate such a high number of cores, satisfying the need for on-chip communication and data transfers, is the Network-on-Chip (NoC) architecture [15, 49].

Resource management has been explored in modern multi-processor architectures both in the embedded community (in MPSoC and NoC systems) and in the distributed/cloud computing communities. Efficient management usually translates to optimizing job real-time constraints and power envelopes. Representative examples of such work include [9, 19, 62, 66, 93]. More recently a framework for distributed resource management has also been proposed [50], but it stills assumes available resources.

In the distributed computing community the constraints are seldom as tight. Resources, entire computers in this case, can become available or unavailable at any moment [22]. Marketplace mechanisms have been extensively explored as a way for independent agents to interact with the central “authority” using a simple and efficient procedure [24, 53]. The aforementioned articles treat computing resources

as agents that strive to manage the system's workload and resources as efficiently as possible, either in isolation or in cooperation with each other.

Future consumer electronics devices are expected to move towards the complexity of previous generation distributed computing systems due to inability to increase processing power by other means. Their system level constraints, on the other hand, will not be relaxed. Intel has already created platforms with 48 general-purpose processing cores [45], while ST-NXP is also advocating the need for many core platforms for upcoming and future mobile telephony standards [84]. The industrial vision goes as far as thousand core chips [20].

NoCs have been recognized as the new paradigm to interconnect and organize a high number of cores. NoCs address global communication issues in System-on-Chips (SoC) involving communication-centric design and implementation of scalable communication structures evolving application-specific NoC design as a key challenge to modern SoC design.

Although the concept of NoC has been derived from traditional interconnection networks, they have some special features that make them unique during design time. As presented in [64], the NoC design key challenges are (i) the communication infrastructure, (ii) the communication paradigm selection and (iii) the application mapping optimization. The choice of network topology, mapping and routing are important design issues which can dramatically affect network's performance such as network's average delay and power consumption.

There are two main architecture templates concerning the NoC design space: (i) Regular and (ii) Irregular design. The most common regular architecture is the mesh topology in which each router is connected to its four neighboring routers via a bi-directional channel and an embedded core is attached to the router. Regular NoC topologies are more suitable for general-purpose on-chip multiprocessors, which consist mostly of homogeneous set of processing and storage arrays and can benefit from spatial locality to achieve higher performance [32].

However, applications running on regular NoCs do not fully exploit interconnection network's capabilities. The diversity of communication

in the network is affected by architectural issues such as system composition and clustering. Irregular NoC design serve better the application requirements since they are application specific and they maximize application's utilization factor in terms of power and network's delay. However, the irregular NoC design is more complex and depends on a variety of design parameters compared to the regular NoC design.

In this section we present a high-level customization framework and methodology for resource management on NoC architectures, both regular and irregular, based on application needs. The whole optimization framework is automated and evaluated on a SystemC simulator. *The goal of this framework is to provide to the designer a variety of choices in resource management and application customizations in order for him to test and evaluate a variety of different configurations.*

5.2 NoC framework overview for resource management

The presented framework for supporting and evaluating resource management on NoC topologies, regarding network's performance and power consumption, is based on a modified version of the high-level Noxim [2] SystemC NoC simulator. The simulator has been heavily modified in order to support the following services: (i) The simulator supports custom irregular topologies with custom routers build as separate SystemC modules; (ii) a configuration file can be given as input to the simulator describing each time the irregular NoC topology; (iii) the designer can specify the size of each buffer in every router used in the simulation resulting to different power consumption and network's delay. This number can be different for different routers in the same simulation; (iv) the Ebit energy model [90] has been added in order to make an evaluation of network's power consumption; and (v) a custom table-based routing can be defined by the designer in a separate file order to have custom routing. If not the default in irregular design is Dijkstra shortest path algorithm while for regular design is the conventional XY routing. To clarify, with the term irregular NoC topology we define any custom 2D NoC topology using routers with multiple ports other than the ones used for the standard mesh and torus topologies. Figure 5.1

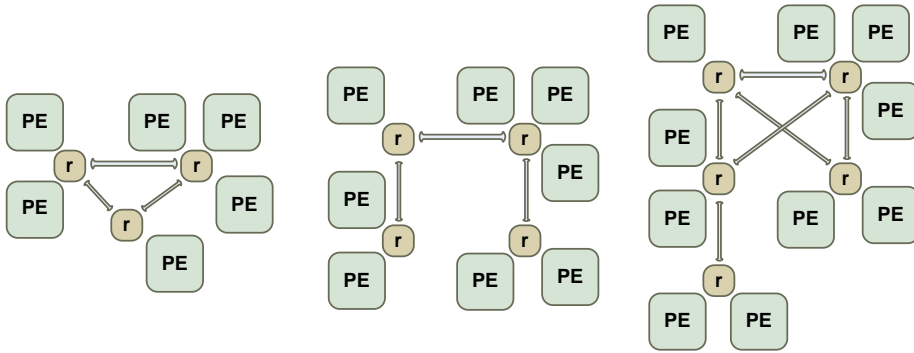


Figure 5.1: Examples of irregular NoC topologies. Each router (r) can serve more than one Processing Element (PE) by having multiple ports.

shows an example of such irregular topologies.

An overview of the developed framework for designing application-specific regular and irregular NoC architectures is presented in Figure 5.2. Having as input a high-level description of the target application, we choose whether to follow the regular or the irregular NoC design flow for resource management. The decision for choosing the appropriate design path depends on designer's goals and application's constraints and requirements.

In the regular NoC design flow, the first step is to select the appropriate generic NoC architecture template (mesh, torus) and then apply a bandwidth-constraint mapping algorithm so as to efficiently utilize platform's characteristics. The mapping step is composed of the mapping core algorithm procedure as well as of an iterative swapping step for better refining of the final result. Then, in order to further adjust to application's constraints and requirements and give certain classes of traffic preferential treatment, a priority assignment scheme follows.

In the irregular NoC design flow, the first step is to perform application partitioning followed by the clustering procedure. In the partitioning step, application's tasks are split into different logical partitions based on bandwidth-aware criteria. Next, in the clustering procedure, the designer can attach an appropriate router to each partition by linking a SystemC library with configurable components and finally generate

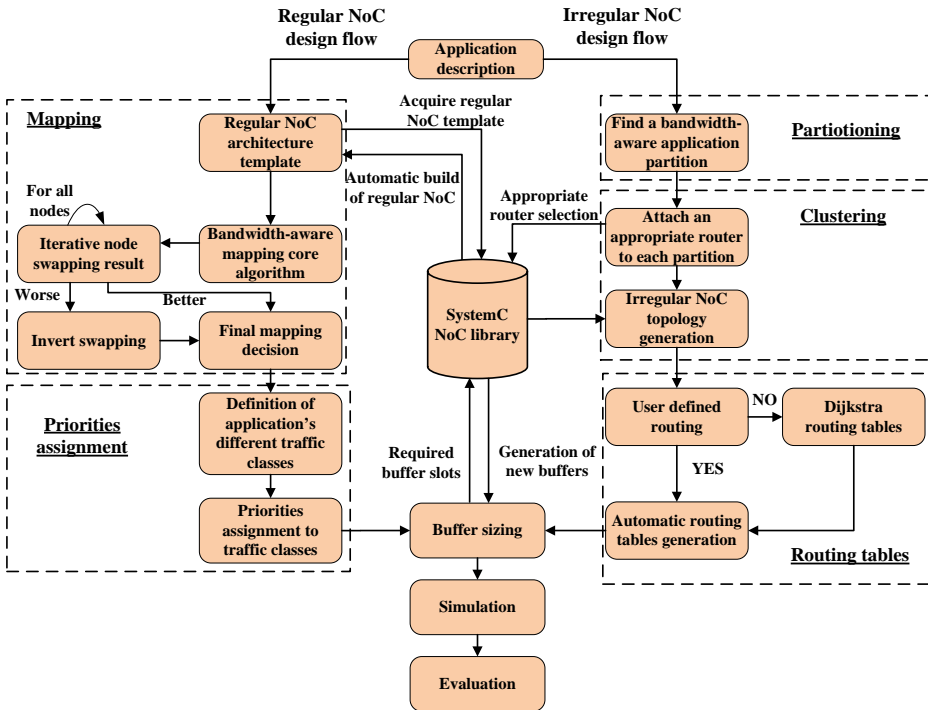


Figure 5.2: Proposed high-level simulation framework for automatic generation of application-specific NoC architectures

the irregular NoC topology. Moreover, the designer can specify a custom routing algorithm, by defining its own routing tables, or let the proposed framework automatically build them with Dijkstra's shortest path algorithm.

Last, in both regular and irregular design, additional power consumption reduction can be achieved by reducing the number of unnecessary input buffer slots in the network. The developed framework uses a buffer sizing algorithm in to order to calculate the appropriate buffer slot number for the generated architectures based on application's needs.

As aforementioned, an application task graph (ATG) is used to capture the traffic flow characteristics. ATG is a common way to abstractly present application's constraints and requirements and it has been widely used in NoC research field as benchmarks for NoC design tools and algorithms for both design- [17, 46, 61] and run-time [9, 50]

NoC optimizations. The ATG $G(T, D)$ is a directed acyclic graph, where each vertex t_i represents a computational module in the application. Each task t_i is annotated with relevant information, such as execution characteristics on each Processing Element (PE) type on the network. Each directed arc $d_{i,j} \in D$ between tasks t_i and t_j characterizes data and communication dependencies. Each $d_{i,j}$ has an associated value $b(d_{i,j})$, which stands for the communication volume exchanged between tasks t_i and t_j .

An NoC platform topology and its communication infrastructure can be uniquely described by a strongly connected directed graph $A(I, N)$. The set of vertices N is composed of two mutually exclusive subsets N_{PE} and N_C containing the available platform's PEs and the platform's on-chip interconnection elements. The set of edges I contains the interconnection information (both physical and virtual) for the N set.

5.2.1 Resource management in regular NoC design

In regular NoC design path, the presented framework performs application-specific customization by employing a priorities assignment scheme.

By providing priorities to different applications, data flows or traffic classes we can guarantee a certain level of system's performance. In many applications it is desirable to give certain classes of traffic preferential treatment in order to meet application's deadlines and QoS standards. In a priority queuing system transfers are divided into $K \geq 2$ classes numbered $1, 2, \dots, K$ where the lower the priority, the higher the class number. In other words, priority i transfers are given preference over priority j transfers if $i < j$.

In the context of this work, four priority classes are provided. Two more bits have been added to each flit as header information. These two bits declare the priority type of each flit (00 is highest and 11 is lowest) and indicate the way the flit will be served in the buffer of each router. The implementation of priorities assignment on NoC architectures is affected by (i) the wormhole switching and (ii) the in-order packet delivery constraints. The presented framework uses a *modified non-preemptive* priority policy in which a high priority packet

can move ahead of all the low priority ones waiting in the queue, but low priority packets already in service are not interrupted by high priority ones.

The router in wormhole switching can be described by a two-state variable $g \in \{1, 2\}$, where $g(1)$ represents the idle state (no packet is being served) and $g(2)$ the busy state (a packet is being served). In wormhole switching the router does not tear down the connection if it has not served the whole packet. Let W_{k_m} be the time spent waiting (not in service) by the m flit of packet with priority k , T_s be the time spent in service by a flit and W_s presents the waiting time of a flit in router that it is in the busy state (a packet is being served) This time is the same for all flits of all classes. n be the number of flits per packet ($n > 1$) and k be the priority class id, ($k \in [1, 4]$). The waiting time of the m flit for the two states is presented in Equations 5.1 and 5.2.

$$\begin{aligned}
 W_{k_m}^{g(1)} = & \underbrace{\sum_{i=1}^{k-1} n(W_i + T_s)}_{\text{Flits with higher priorities}} + \underbrace{\sum_{l=1}^{m-1} l(W_k + T_s)}_{\text{Ahead flits of the same packet}} + \\
 & + \underbrace{A \times (W_k + T_s)}_{\text{Flits of other packets with the same priority that arrived first}} \quad (5.1)
 \end{aligned}$$

$$\begin{aligned}
 W_{k_m}^{g(2)} = & \underbrace{T_s}_{\text{Flit that is being served}} + \underbrace{j(W_s + T_s)}_{\text{The remaining flits of the packet}} + \\
 & + \underbrace{\sum_{i=1}^{k-1} n(W_i + T_s)}_{\text{Flits with higher priorities}} + \underbrace{\sum_{l=1}^{m-1} l(W_k + T_s)}_{\text{Ahead flits of the same packet}} + \\
 & + \underbrace{A \times (W_k + T_s)}_{\text{Flits of other packets with the same priority that arrived first}} \quad (5.2)
 \end{aligned}$$

where $j \in [0, n - 1]$ and $A \in \mathbb{Z}$.

Each buffer has a dedicated size of $|B|$ buffer slots in flits that can serve. According to Equations 5.1 and 5.2 we can extract, in Equation 5.3, the maximum number of flits per packet in order for the priorities to have effect leading to the result that the buffer size of input channels must be bigger than the number of flits per packet in order for the priorities to have an actual role in the network.

$$j + n + \sum_{m=1}^{l=1} l + N < |B| \Rightarrow 0 < n < |B| \quad (5.3)$$

When $n \geq |B|$, $W_{k_m}^{g(1)}$ and $W_{k_m}^{g(2)}$ follow the form described in Equation 5.4.

$$W_{k_m}^{g(1)} = W_{k_m}^{g(2)} = T_s + j(W_s + T_s) \quad (5.4)$$

which does not depend on variable K . In that way even though priorities exist, they are not actually used.

While implementing priorities in NoCs, elimination effects can occur during which packets with low priorities are always stuck in buffers without being processed due to the arrival of high priority packets. In order to avoid these elimination effects we have added to each flit a field called *waiting_time* which starts from zero and increases every that it is not being processed by the router. Whenever *waiting_time* exceeds a threshold, then the priority type of this specific flit decreases until it reaches zero which is the highest priority type. Thus, new incoming flits are put behind it and no elimination effects take place.

The flit management and placement procedure in buffers based on priorities and wormhole switching is presented in Algorithm 5. First the algorithm checks if the flit has the lowest priority (line 2). If so, the flit is placed in the last position of the buffer queue (line 3). If this is not the case, then first check if another flit is being processed (line 4). If this is the case (line 5), the buffer queue is searched (and assuming that the flits of the packet that is being processed have the highest priority) the incoming flit is placed in the first appropriate slot based on its priority. This action is performed because as aforementioned, wormhole switching does not allow breaking the processing of a packet by inserting and processing flits of another one. In line 8 we check whether we have the

same priority with the first flit in buffer. After that, in line 9, we search the buffer based on flits sequence number and we confirm its position by its priority. In all other cases (lines 11-13) we search the entire buffer and put correctly the flit based on its priority. After the flit is put in the buffer, we search all flits for the *wating_time* field and according to each value we update either the priority or the *wating_time* field in order to avoid elimination effects (lines 17-24).

Algorithm 5 Priority-based flit manipulation algorithm

```
1: if flit.priority == LOW then
2:   buffer.put_back(flit)
3: else
4:   if buffer.is_processing() then
5:     buffer.search_by_priority()
6:     buffer.insert(flit)
7:   else
8:     if first == flit.priority then
9:       buffer.search_by_position()
10:      buffer.insert(flit)
11:     else
12:       buffer.search()
13:       buffer.insert(flit)
14:     end if
15:   end if
16: end if
17: p=buffer.head
18: while p!=NULL do
19:   if p.wating_time > threshold then
20:     p.priotiry--
21:   else
22:     p.wating_time ++
23:   end if
24:   p=p.next
25: end while
```

5.2.2 Resource management in irregular NoC design

The irregular NoC design path provided by the presented framework offers to the designer the opportunity to fully exploit interconnection network's capabilities by the automatically generation of customized NoC topologies. In contrast to regular NoC design path irregular topologies are application dependent and the customization process involves complex steps such as (i) application partitioning, (ii) clustering and (iii) routing tables generation. As aforementioned, the goal of this work is not to present a design methodology for selecting the best application-specific NoC topology. On the contrary, the presented framework gives

the designer a variety of choices and application customizations in order for him to test and evaluate a variety of different configurations thus, selecting the one topology that best fits his needs.

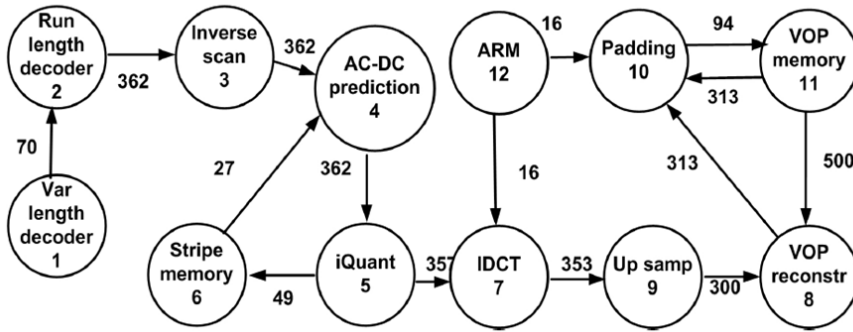
5.2.2.1 Application partitioning

The first step is a bandwidth-aware application partitioning process. A careful and precise analysis of application's characteristics and particularly tasks' communication bandwidth is essential in order to define the design space and the components that is composed of.

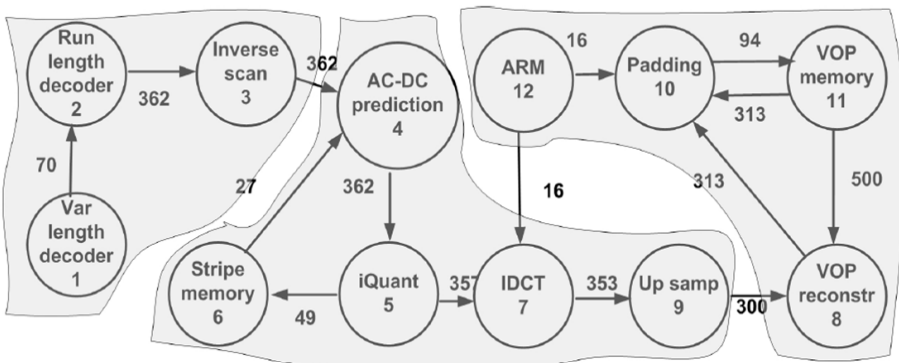
We define the set R which describes the logical application's partitions. R is composed of m ($m \geq 1$) subsets $R_1, R_2, \dots, R_i, \dots, R_m$ such that $\bigcap_{i=1}^m R_i = \emptyset$ and $\bigcup_{i=1}^m R_i = R$. A partition R_i is considered valid if $\exists t_i \in R_i : t_i \in T$. We define as $K[R_i] = \sum b(d_{i,j}) \forall t_i, t_j \in R_i$ the communication cost inside the R_i partition and as $Q[R_i R_j] = \sum b(d_{i,j}) \forall t_i \in R_i, t_j \in R_j$ the communication cost between the R_i and R_j partitions. The goal of application partitioning step is to compose the appropriate R set so as (i) to minimize the communication needed between the partitions and (ii) balance the load inside partitions as described in Equation 5.5.

$$R = \{R_1, R_2, \dots, R_m\} : \{K[R_1] \simeq K[R_2] \simeq \dots \simeq K[R_m]\} \wedge \wedge \min\{Q[R_1 R_2], Q[R_1 R_3], \dots, Q[R_1 R_m], \dots, Q[R_{m-1} R_m]\} \quad (5.5)$$

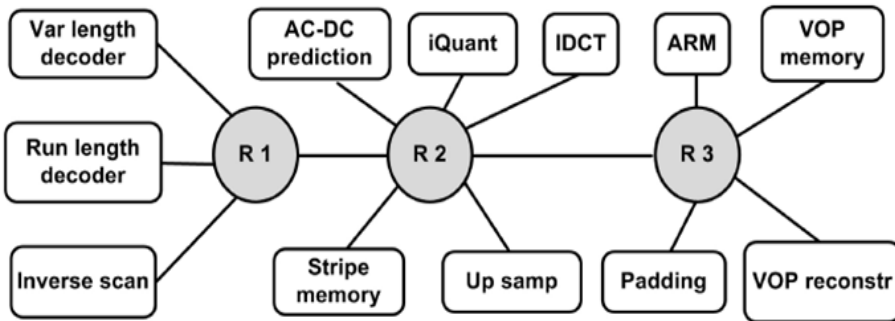
The graph partitioning problem is NP-complete, and we therefore should not expect to solve it in polynomial time. The partition algorithm used is the *Multilevel - KL* algorithm [41]. The input to the Multilevel-KL algorithm is an application task graph, like the VOPD benchmark shown in Figure 5.3(a). *Multilevel - KL* method works by creating a sequence of increasingly smaller graphs approximating the original graph, partitioning the smallest graph, and projecting this partition back through the intermediate. The algorithm for constructing smaller approximations to the graph relies upon finding a maximal matching in the graph, and then contracting edges in the matching. An example of the implemented partitioning procedure, implementing three partitions, is presented in Figure 5.3(b).



(a) VOPD task graph



(b) Task graph partitioning



(c) Clustering

Figure 5.3: VOPD (a) application task graph , (b) partitioning using *Multilevel - KL* algorithm [41] and (c) clustering.

The proposed framework supports the automatic application partitioning with the integration of the Chaco graph partition tool [1]. With the automatic procedure, the proposed framework can generate multiple size partitions depending on ATG size and it can be used for a fast design space exploration of irregular NoC topologies.

5.2.2.2 Clustering

In the clustering procedure, the produced application's partitions are used as an input to our SystemC library. The goals of the clustering procedure are (i) to attach an appropriate multiple-port router to each partition and assign PEs accordingly and (ii) connect routers specifying particular characteristics and forming the irregular NoC topology (Figure 5.3(c)). According to the specified $K[R_i]$ and $Q[R_i R_j]$ for each R_i , a suitable router model is selected in order to be able to serve partition's bandwidth needs. In the developed SystemC library each router can be customized according to the (i) number of input / output ports, (ii) link control, (iii) switch, (iv) routing and (iv) arbitration units. Each router port has its own request and acknowledge flit ports and connections. These connections can be either links with other routers on the NoC or connections with PEs. The exploration of irregular NoC topologies demands different number of connections for each router in order to maximize network's performance. Also, in the clustering step, the designer can additionally specify other network's characteristics such as channel width, flits per packet, input channel buffer size, output channel buffer size etc.

The proposed framework supports a variety of SystemC router models with different characteristics which, based on the aforementioned partitioning procedure, can be used for a fast and automatic design space exploration. The definition of the irregular topology in the simulator is done with the usage of a configuration file like the one presented in Figure 5.4. The topology generated by this configuration file is shown in Figure 5.5.

In the example presented in Figure 5.4:

1. Router 0 (R0) has connections with router 1(R1), router 2 and processing element 7.

```

%Topology table file
%4 Routers, 8 PE's
%port in, port out, buffer in, buffer out, Router, layer
R0: 1,2,4,2,R1,0 2,6,3,4,R2,0 ; 3,9,4,P7,0 ;
R1: 1,3,4,8,R3,0 ; 3,2,8,P5,0 4,2,8,P4,0 5,2,8,P3,0 ;
R2: 9,5,4,5,R3,0 ; 2,4,4,P6,0 ;
R3: ; 6,4,4,P0,0 4,4,4,P1,0 9,5,5,P2,0 ;

```

Figure 5.4: Configuration file structure

2. The connection R0-R1 is established between port 1 of R0 and port 2 of R1.
3. The buffer dedicated to traffic from R1 to R0 has 4 slots and for flits arriving from R0 the R1 there is a 2 slots buffer in R1.
4. For the connection R0-R2, there is a 3 slots buffer on port 2 on the R0 side and a 4 slots buffer on port 6 on the R2 side.
5. R0 connects with P7 using its 3rd port and a buffer with 9 slots. Processing elements have only one port and no buffers, that's why number 4 previous to P7 doesn't declare anything.
6. Similarly, R1 connects to R3 and three processing elements (P3, P4, P5), besides the connection with R0 declared in the previous line.

5.2.2.3 Routing Table Generation

On irregular NoC architectures, conventional routing algorithms are hard to implement due to different architecture features and due to the fact that they should be specifically modified in order to serve the specific irregular topology. However, a small change in the NoC topology leads to a total routing algorithm redesign. This, has a result of a time consuming design-phase and it is quite likely to create routing problems such as deadlocks and bottlenecks.

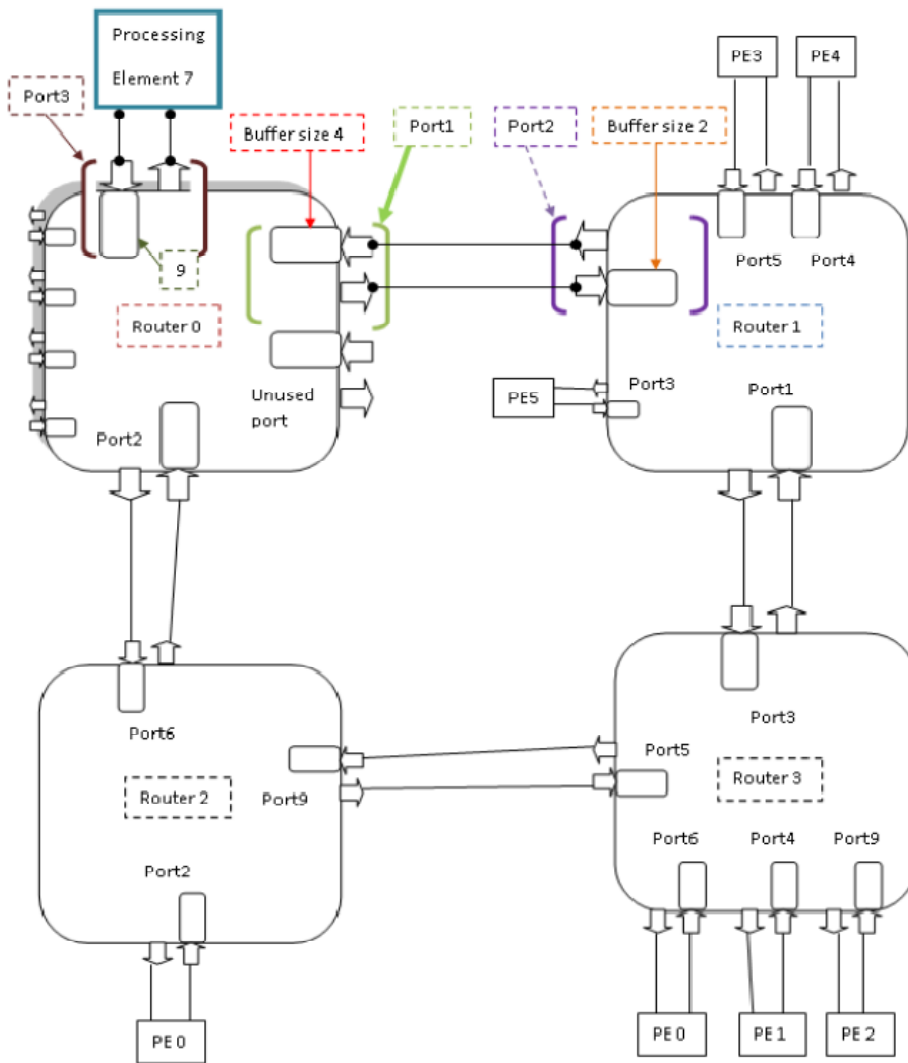


Figure 5.5: Example of the generated topology using the configuration file presented in Figure 5.4

The proposed framework supports two types of routing tables: (i) designer defined and (ii) Dijkstra based routing tables. In the first case, the designer creates and imports, through the corresponding interfaces, the routing tables to the generated, from the clustering step, routers. However, this selection cannot guarantee the lack of any mistakes in the routing procedure. By selecting the second choice, the proposed framework automatically generates routing tables for irregular NoC topologies based on Dijkstra shortest-path algorithm. The benefits of supporting the automatic building of routing tables are: (i) faster design deployment, (ii) livelock and deadlock avoidance, (iii) elimination of useless entries, (iv) no data overhead on packets, (v) shortest path guaranteed and (vi) easily understandable structure.

5.2.3 Buffer Sizing

Power consumption is a very crucial metric while evaluating NoC architectures since input buffers, crossbar switch and link circuits dominate network power consumption [76]. In order to employ efficient resource management in NoC design, power consumption should be low. A technique to efficiently reduce network's power consumption is the reduction of routers' input buffer slots while meeting application's constraints and requirements. As the final design step, the developed framework, uses a buffer sizing algorithm that is applicable to both regular and irregular topologies.

The buffer state is controlled by a two-state crossbar variable $c \in \{1, 2\}$, where $c(1)$ represents the idle state and $c(2)$ the busy state. At time t the total number of flits in the buffer is x_t , the number of flits entering is ξ_t , and the number of flits exiting is ν_t ; the buffer has a maximum capacity of $\nu|x^+$. The dynamics of the flits flow at one buffer is described by Equation 5.6 for $c_t = c(1)$ (idle state) and by Equation 5.7 for $c_t = c(2)$ (busy state).

$$x_{t+1} = \begin{cases} x^+ & \xi_t + x_t > x^+ \\ \xi_t + x_t & \xi_t + x_t \leq x^+ \end{cases} \quad (5.6)$$

$$x_{t+1} = \begin{cases} x^+ & \xi_t + x_t - \nu_t > x^+ \\ \xi_t + x_t - \nu_t & 0 \leq \xi_t + x_t - \nu_t \leq x^+ \end{cases} \quad (5.7)$$

where the function x^+ is defined in Equation 5.8.

$$x^+ = \begin{cases} x & : \quad x \geq 0 \\ 0 & : \quad x < 0 \end{cases} \quad (5.8)$$

Every t cycles the input buffer B_i of the i router has the utilization percentage u^t described by Equation 5.9

$$u_i^t = \frac{N_i - \mu_i \times E_i}{[B_i]} \quad (5.9)$$

where N_i is the packet arrival rate of the i router, μ_i is the packet service rate of the buffer, $[B_i]$ is the total size of the buffer in flits and

$$E_i = \begin{cases} 0, & \text{if buffer is empty and} \\ 1, & \text{if buffer is not empty} \end{cases} \quad (5.10)$$

The minimum buffer size, required by the application, is produced by Equation 5.11 after a simulation window in which, for every cycle t , u_i^t is calculated. The returned values are used as the lowest threshold for the input buffer slots.

$$B_i^t = \begin{cases} 0, & t = 0 \\ u_{k,l}^t, & t > 0, u_i^t > u_i^{t-1} \\ u_{k,l}^{t-1}, & t > 0, u_i^t < u_i^{t-1} \end{cases} \quad (5.11)$$

5.3 Evaluation

In order to validate our framework for resource management, we have performed extensive simulations using five DSP applications (a) Video Object Plane Decoder (VOPD) [61], (b) MPEG-4, (c) Multi-Window Display (MWD) [17], (d) Picture-In-Picture (PIP) [17] and (e) MultiMedia System (MMS) [46].

In modern platforms and applications, designers should guarantee that certain classes of traffic will meet application's requirements. In order to provide different classes of traffic management, further optimization

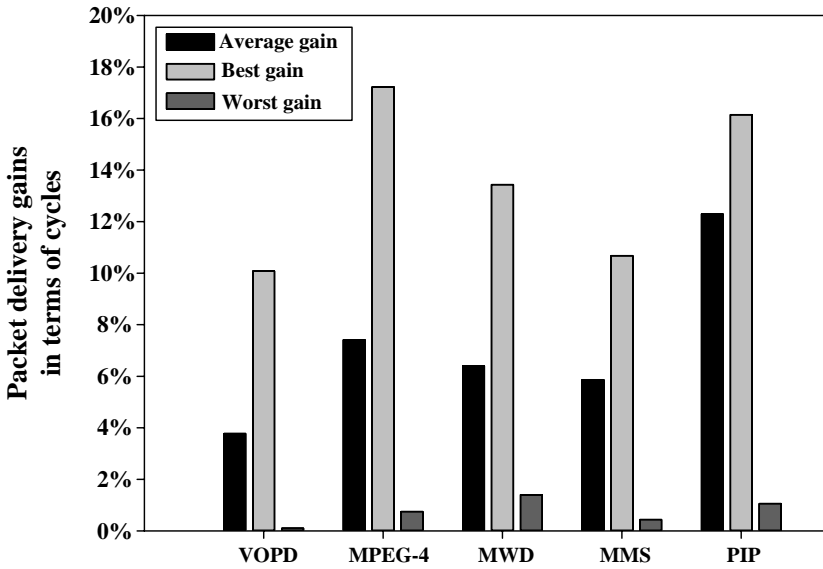


Figure 5.6: Average, best and worst gains ,in terms of cycles, for packet delivery time while employing priorities.

is needed with the employment of priorities assignment. For example, multimedia-based applications have strict time deadlines and specific tasks require in-time packet delivery. In order to guarantee the time deadlines, priorities are used in order to forward packets of the selected tasks faster inside the NoC and reduce the packet delivery time. For the previously presented applications we have tested the priorities assignment scheme to further reduce packet on-chip delivery time for certain traffic classes while keeping overall network's average delay constant in terms of execution cycles. We separated traffic to different flows and in each flow a different priority type was assigned. Specifically, for our tested applications we separated three types of traffic flow: a) Packets going to memory node, b) Packets coming from memory node and c) Packets from nodes with high average delay (more than 50% of average network delay). The reason for choosing memory node as the dominant node on which most priority types are applied is because on-chip communication with memory is costly due to memory's slow response time. Priorities were implemented only on regular NoC topologies since partitioning imposes the minimization of hop number among the communicating cores on irregular topologies. Figure 5.6 shows the average

(7.15% on average), the best (13.52% on average) and the worst gains (0.75% on average), in terms of cycles, for packet delivery time while employing priorities to specific traffic classes.

In the irregular NoC design path the experimental results focus (i) on NoC's throughput and (ii) NoC's average delay in terms of cycles. For the irregular NoC design framework, our goal is to perform comparisons between a variety of irregular NoC topologies and the regular mesh NoC topology. So, the experimental set-up for irregular NoCs is:

- We partitioned the VOPD, MPEG4 and MWD applications from 2 to 12 partitions and the MMS from 2 to 25 partitions according to the presented methodology.
- Each partition was clustered to the appropriate NoC topology automatically using components from the SystemC library
- Each packet consists of 4 flits
- The routing tables are automatically created using the Dijkstra shortest path algorithm.

The reason for simulating so many topologies is that it cannot be predicted beforehand whether an irregular design with few big routers would be more efficient than a design with more small ones. This point actually validates our framework to be used for *fast and automatic design space exploration*. The baseline for all our irregular NoC metrics, is the regular mesh NoC topology and Figures 5.7-5.8 are normalized according to that.

5.3.0.1 NoC's throughput

Throughput is a measure of the comparative effectiveness of large networks and it is considered as one of the most important characteristics of a network. Especially for networks that serve multimedia traffic there is great demand for high throughput. We define NoC throughput as the total amount of flits that are delivered per cycle to all destinations inside the NoC.

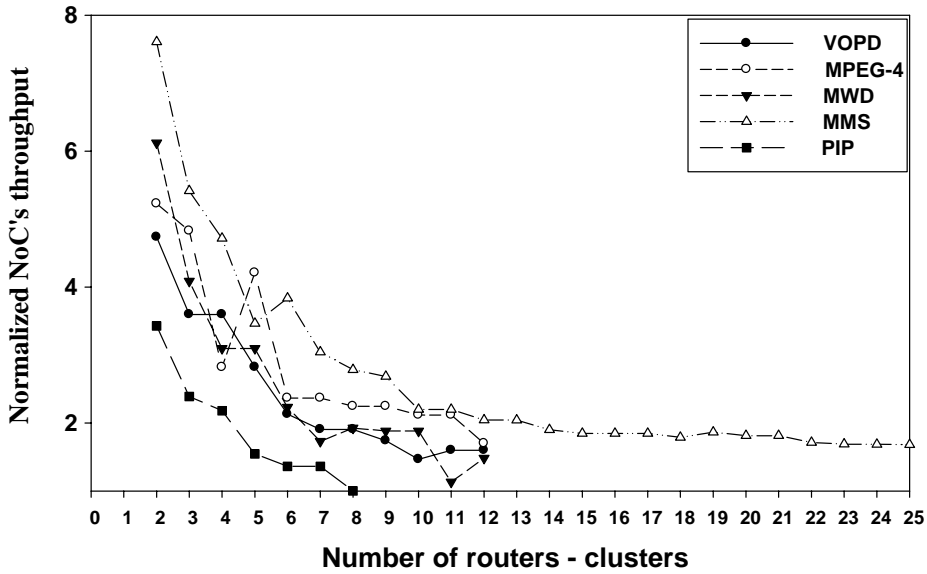


Figure 5.7: NoC's throughput for the selected applications

Figure 5.7, presents NoC throughput for the selected applications and as can be seen, the throughput of each application for each irregular topology is better than the mesh one. Specifically, for the VOPD and MMS applications, irregular topologies increase the throughput of the NoC, in best case, by $\times 4.7$ while for the MPEG4, PIP and the MWD applications, by using irregular NoC topologies we can achieve an increase of the throughput by $\times 8$, $\times 3.4$ and $\times 6.1$ respectively in best cases. Additionally, the average gain for VOPD, MMS, MWD and MPEG is approximately $\times 2.5$ while for the PIP is $\times 1.9$ compared to the mesh topology. To sum up, all irregular NoC topologies achieve better throughput than the mesh one due to the clustering that was performed which (a) offers better communication characteristics (b) takes care of topology's features and (c) keeps the number of hops low.

5.3.0.2 NoC's average delay

The simulation results for irregular NoC's average delay in comparison with regular topologies is presented in Figure 5.8. The average delay, in terms of cycles, of the selected applications for each irregular topology

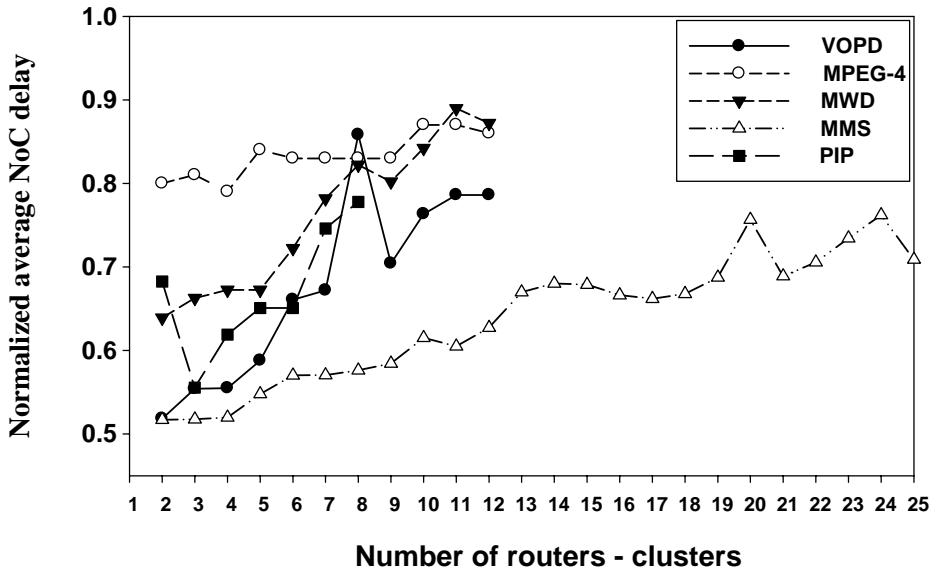


Figure 5.8: NoC's average delay for the selected applications

is better than the regular mesh one. Specifically, there is, in best case, a 50% delay reduction for the VOPD application and a 20% for the MPEG4. Also, MWD, PIP and MMS reduced the delay by 40%, 45% and 50% respectively in best case. The average gain for VOPD, MPEG4 and PIP is 30%, while for MWD and MMS is 17% and 25% respectively compared to the mesh topology. Irregular topologies can serve flits faster due to their application-specific partitioning and clustering connecting in the same router components that communicate frequently greatly reducing NoC's average delay.

5.3.1 Buffer's power consumption

As aforementioned input buffers dominate network power consuming up to 60% of the total network power consumption. Figure 5.9 shows the input buffer power consumption both for regular and irregular design with the addition of the buffer sizing algorithm presented in Chapter 5.2.3. The goal is to minimize buffer slots while keeping network's average delay untouched. The abolition of unnecessary buffer slots leads to great power reduction.

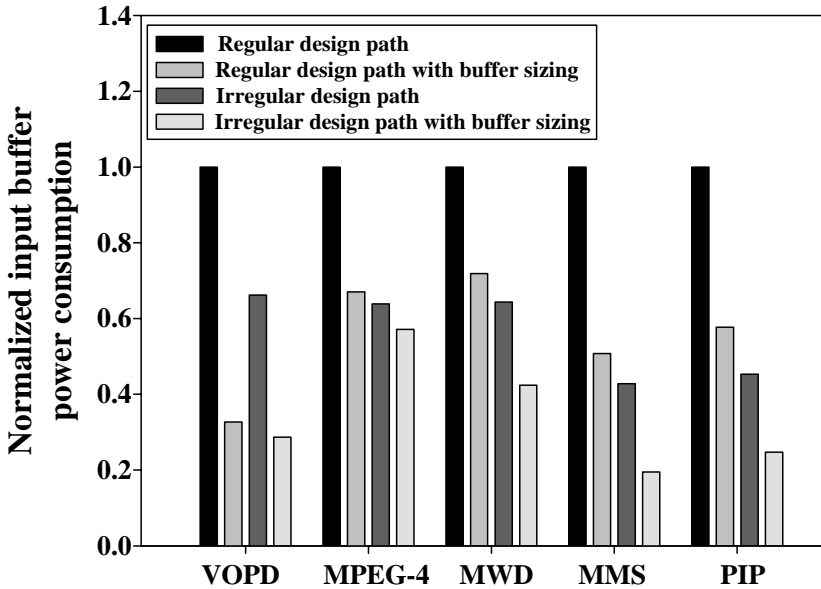


Figure 5.9: Normalized power consumption for both regular and irregular design path with the addition of the proposed buffer sizing algorithm

As depicted in Figure 5.9, the proposed buffer sizing algorithm can achieve at regular NoC design path, an average of 44% buffer power reduction compared to the case where no buffer sizing algorithm was performed. Furthermore, the irregular NoC design path can, by itself, achieve a reduction of 45% on average compared to the regular NoC design path. This can be explained by the fact that flits travel almost the minimum number of hops inside the NoC thus leading to no additional buffer storage in any intermediate router. Last, with the addition of the proposed buffer sizing algorithm we can further reduce the buffer power consumption by a factor of 20%.

5.3.2 Conclusions

NoCs have attracted significant research attention since they are recognized as a scalable paradigm to interconnect and organize a high number of cores. In this chapter we presented a high-level NoC customization

framework for supporting and evaluating resource management on NoC topologies. The presented framework gives the designer a variety of choices and application customizations in order for him to test and evaluate a variety of different configurations and resource management choices.

Chapter 6

Conclusions

This chapter presents the conclusions drawn from the Ph.D. Thesis. It also summarizes the novelties and presents the future extensions of the dissertation. The novelties achieved in the presented Thesis makes possible the development of new design methodologies that can build up and further improve the memory and run-time resource management in embedded systems.

6.1 Summary of Ph.D. Thesis

In this Ph.D. Thesis, we have presented memory management acceleration and customization frameworks. Our main goal is to efficiently address the problems of:

- Customized dynamic memory managers on many-core embedded platforms.
- Distributed functionality over a DSM environment.
- Low power consumption in dynamic memory management.

In order to address these critical issues that affect modern embedded systems, new methodologies were developed and presented:

- A framework for generating microcoded DMM services on top of a hardware dual-microcoded controller was presented. Microcoded approach was selected for supporting custom DMMs on many-core

platforms because it takes advantage of hardware performance while keeping software flexibility.

- A microcode-accelerated flexible, distributed and scalable allocator, called MAD-DMM, is proposed. MAD-DMM provides distributed functionality over a DSM environment, under microcode implementation, while keeping the standard C-API (`malloc()`/`free()`) thus being transparent to the application. Unlike high-level allocators, information about the allocator metadata is not stored at high-level but at microcode-level as part of a local heap table.
- A new design strategy is proposed for the implementation of an efficient high-level methodology of applying transparent monitor and DVFS decision mechanisms into any any C-allocator targeting low power consumption.

All of the presented methodologies were quantitatively evaluated and compared on a many-core platform composed of Processor-Memory (PM) nodes interconnected via a packet-switched mesh network. A PM node is composed of a LEON3 processor with its own I-Cache and D-Cache, a Dual Microcoded Controller (DMC) and memory which can be shared among the nodes [29]. Experimental results on several benchmarks have shown significant performance and power consumption gains regarding the preferred customization.

As aforementioned, resource management is a key technology for the successful use of modern many-core systems and run-time resource management paradigm has become prominent due to the fact that it can deal with the run-time dynamicity of applications and platforms. However, modern complex MPSoCs face various problems on the field of run-time resource management:

- Existing approaches to run-time resource management problem on many-core platforms, even if they expose some autonomic properties, are typically centralized thus creating a central point of failure.
- The central core that analyzes the data limits scalability and it becomes a bottleneck for processing and communication functions.
- Most run-time approaches lack of a self-adaptation process thus

creating starvation problems due to the high amount of incoming applications that are unable to be served by the platform.

In order to face these problems, a run-time resource management framework, targeting many-core platforms, was developed and addressed the issues in the following ways:

- A distributed (Divide & Conquer based) framework for run-time mapping on both homogeneous and heterogeneous many-core platforms has been developed. The framework achieves different levels of platform's resources utilization depending on application's needs.
- A distributed workload-aware distributed run-time framework for malleable applications, running on many-core platforms, is presented.
- A high-level customization framework and methodology for resource management on NoC architectures, both regular and irregular, has been developed.

The proposed frameworks are based on the idea of using multiple cores in different roles while, in all case, an on-chip intercommunication scheme ensures decision distribution. The proposed frameworks were evaluated on the experimental many-core platform presented in [29] and on the Intel Single Cloud Chip (SCC) many-core one [45].

6.2 Perspectives and Future Extensions

The work described in this Thesis can be the basis for addressing the new problems that will arise in the embedded world in a short or medium term. These new problems arise due to the increased complexity of hardware platforms (Figure 6.1) and software applications.

According to HiPEAC roadmap [4], the newest problems, relative to the thesis, in embedded systems is the integration of appropriate software in many-core platforms targeting run-time management and the acceleration in processing the huge amount of data produced by modern

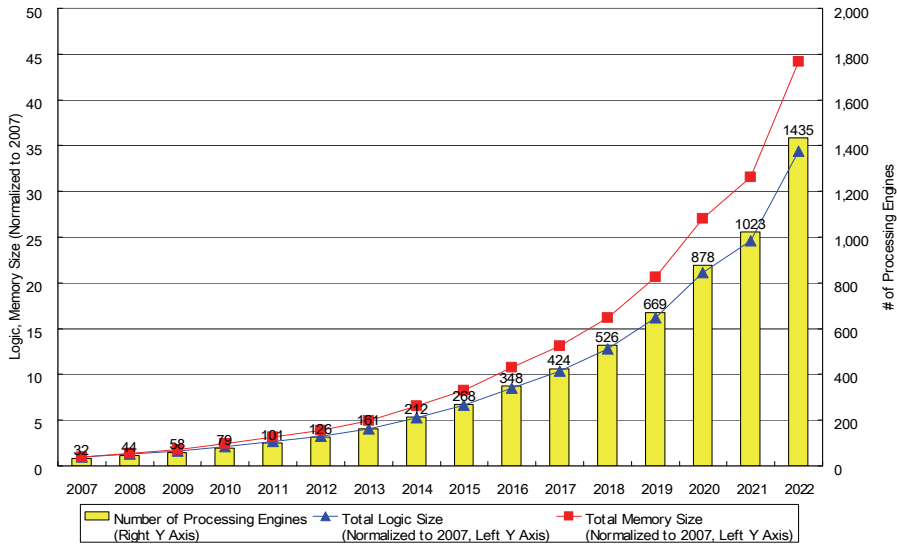


Figure 6.1: Trends of increased complexity in Systems-on-Chip targeting the market of consumer mobile devices [74]

applications. A first approach to a solution, is to expand the framework for generating customized microcoded DMM services in order to change the allocator policies at run-time. Until now, the allocator works on predefined policies for handling application requests. These policies have to do with the usage of coalesce and split procedures, the usage of fixed lists etc. A memory allocator that is able to adjust these parameters at run-time and according to application needs, will be much more efficient. As aforementioned, in a many-core platform a big number of application arrives every moment. Each of these applications is different in terms of memory usage for their dynamic data. Thus, a distributed and microcoded allocator that also has the ability to adjust its settings at run-time will be much more efficient in terms of memory footprint and performance. In order to achieve this, the allocator must work on a scenario-based configuration. That means, that the DMM will have some pre-defined working modes and it can change between them at run-time. The criteria for changing working mode, can be application performance, memory utilization, power consumption etc.

Another extension, in the run-time resource management field, it would

be an integrated task migration mechanism. So, instead of just taking the decision on which processor a task will be mapped on, it would be very useful to change, at run-time, the type of processor that this task is being executed. For example, until now, when a new application enters the system, it waits for an available core to serve it. However, this core might not be the optimal. So, during the self-optimization process, it would really help application's performance if a task "migrated" to another type of core. This means, that all stored data and information must follow the task to the new core, including cache data. In that case, a task migration mechanism would take care of it so as not to lose time and valuable resources.

Following HiPEAC roadmap [4], today's embedded systems are almost universally connected, and in comparison with the past, many attack-hacking incidents have demonstrated that security needs to be improved. This need is especially focused on modern devices as they are continuously handling users' private and personal data. Heap based attacks are an ongoing threat. Bad memory management from the programmer may lead to various errors that could interfere with other processes and enable heap based attacks. Instead of relying on the programmers' memory managing skills, an approach to prevent these heap based attacks is to make the memory allocator safe. This basically means that the allocator tries to tolerate the inevitable memory errors and let them not interfere with anything but the processes that caused them. Most memory allocators ignore security issues. Instead, they focus on maximizing performance and limiting fragmentation and waste. While these are very important issues for a memory allocator, in the days of worms that use code injection attacks to cause significant economical damage, security cannot be ignored. Thus, another extension to the presented thesis would be the integration of security mechanisms in memory allocators targeting distributed and DSM systems. Also, the usage of heterogeneity and hardware accelerators would definitely improve allocator performance since most of the state-of-art techniques in securing heap, require a lot of processing cycles in order to to guarantee the security of metadata. Microcoded implemented security features would accelerate functions regarding metadata validation and prevent heap exploitations faster.

Last, today's large data centers process the massive amounts of data generated by embedded and mobile computing systems, online trans-

actions, and scientific simulations. Since the input and the process time for these systems is unknown, memory allocators have an essential role to the overall performance. So, an extension to the presented frameworks, it would be the usage of hardware accelerators and customized per application memory allocators for specific application domains in high-performance computing. Even though these systems are not limited by the space of memory, the size of memory transactions is so big that turns out to be one of the most important performance factors.

Publications

Book chapters

1. I. Anagnostopoulos, S. Xydis, A. Bartzas, Z. Lu, D. Soudris and A. Jantsch, “*Chapter 8 Middleware memory management in NoC*”, in “*Designing 2D and 3D Network-on-Chip Architectures*,” to appear, Springer
2. B. Candaele, S. Aguirre, M. Sarlotte, I. Anagnostopoulos, S. Xydis, A. Bartzas, D. Bekiaris, D. Soudris, Z. Lu, X. Chen, J.-M. Chabloz, A. Hemani, A. Jantsch, G. Vanmeerbeeck, J. Kreku, K. Tiensyrja, F. Ieromnimon, D. Kritharidis, A. Wiefrink, B. Vanthournout, P. Martin, “*Chapter 11: Mapping Optimisation for Scalable multi-core ARchiTecture: The MOSART approach*,” in “VLSI 2010 Annual Symposium,” Springer
3. S. Xydis, A. Bartzas, I. Anagnostopoulos, D. Soudris, “*Chapter 3: Application-Specific Multi-Threaded Dynamic Memory Management*,” in “Scalable Multi-core Architectures: Design Methodologies and Tools,” 2011, Springer

Journals

1. I. Anagnostopoulos, A. Bartzas, I. Filippopoulos, D. Soudris, “*High-level customization framework for application-specific NoC architectures*,” in Springer Design Automation for Embedded Systems, vol.16, no.4, pp.339-361, 2013, doi: 10.1007/s10617-013-9114-5
2. I. Anagnostopoulos, J.M. Chabloz, I. Koutras, A. Bartzas, A. Hemani, D. Soudris, “*Power-aware Dynamic Memory Management on*

Many-core Platforms utilizing DVFS,” ACM Transactions on Embedded Computing Systems, vol.13, no.1, pp.40:1–40:25, November 2013

3. I. Anagnostopoulos, S. Xydis, A. Bartzas, Z. Lu, D. Soudris, A. Jantsch, “*Custom Microcoded Dynamic Memory Management for Distributed On-Chip Memory Organizations,*” in IEEE Embedded System Letters, vol.3, no.2, pp.66,69, June 2011

Conferences

1. I. Anagnostopoulos, V. Tsoutsouras, A. Bartzas, D. Soudris, “*Distributed run-time resource management for malleable applications on many-core platforms,*” in Proceedings of DAC conference 2013
2. I. Anagnostopoulos, A. Bartzas, G. Kathareios, D. Soudris, “*A Divide and Conquer based Distributed Run-time Mapping Methodology for Many-Core platforms,*” in Proceedings of DATE conference 2012
3. K. Siozios, D. Diamantopoulos, I. Kostavelis, E. Boukas, L. Nalpantidis, D. Soudris, A. Gasteratos, M. Aviles, I. Anagnostopoulos, “*SPARTAN project: Efficient implementation of computer vision algorithms onto reconfigurable platform targeting to space applications,*” in Proceedings of the 6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011
4. C. Silvano, W. Fornaciari, S. Crespi Reghizzi, G. Agosta, G. Palermo, V. Zaccaria, P. Bellasi, F. Castro, S. Corbetta, E. Speziale, D. Melpignano, JM. Zins, H. Hubert, B. Stabernack, J. Brandenburg, M. Palkovic, P. Raghavan, C. Ykman-Couvreur, I. Anagnostopoulos, A. Bartzas, D. Soudris, T. Kempf, G. Ascheid, J. Ansari, P. Mahonen, B. Vanthournout, “*Parallel programming and run-time resource management framework for many-core platforms: The 2PARMA approach,*” in Proceedings of the 6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011
5. A. Bartzas, P. Bellasi, I. Anagnostopoulos, C. Silvano, W. Fornaciari, D. Soudris, D. Melpignano, C. Ykman-Couvreur, “*Runtime*

Resource Management Techniques for Many-core Architectures: The 2PARMA Approach,” in Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), 2011

6. K. Siozios, I. Anagnostopoulos, D. Soudris, “*Multiple Vdd on 3D NoC Architectures,*” in Proceedings of 17th IEEE International Conference on Electronics, Circuits, and Systems (ICECS) 2010
7. S. Xydis, A. Bartzas, I. Anagnostopoulos, D. Soudris, K. Pekmestzi, “*Custom Multi-Threaded Dynamic Memory Management for Multiprocessor System-on-Chip Platforms,*” in Proceedings of International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS) 2010
8. B. Candaele, S. Aguirre, M. Sarlotte, I. Anagnostopoulos, S. Xydis, A. Bartzas, D. Bekiaris, D. Soudris, Z. Lu, X. Chen, J.-M. Chabloz, A. Hemani, A. Jantsch, G. Vanmeerbeeck, J. Kreku, K. Tiensyrja, F. Ieromnimon, D. Kritharidis, A. Wiefrink, B. Vanthournout, P. Martin, “*Mapping Optimisation for Scalable multi-core ARchiTecture: The MOSART approach,*” in Proceedings of IEEE Computer Society Annual Symposium on VLSI (IS-VLSI) 2010
9. I. Filippopoulos, I. Anagnostopoulos, A. Bartzas, D. Soudris, G. Economakos, “*Systematic Exploration of Energy-Efficient Application-Specific Network-on-Chip Architectures,*” in Proceedings of IEEE Computer Society Annual Symposium on VLSI (IS-VLSI) 2010
10. K. Siozios, I. Anagnostopoulos, D. Soudris, “*A High-Level Mapping Algorithm Targeting 3D NoC Architectures with Multiple Vdd,*” in Proceedings of IEEE Computer Society Annual Symposium on VLSI (IS-VLSI) 2010
11. I. Anagnostopoulos, A. Bartzas, D. Soudris, “*Application-Specific Temperature Reduction Systematic Methodology for 2D and 3D Networks-on-Chip,*” in Proceedings of International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), 2009
12. I. Anagnostopoulos, A. Bartzas, I. Vourkas, D. Soudris, “*Node Resource Management for DSP Applications on 3D Network-on-Chip architectures,*” in Proceedings of 16th International Conference on

Digital Signal Processing (DSP), 2009

Bibliography

- [1] Chaco: Software for Partitioning Graphs. URL <http://www.cs.sandia.gov/~bahendr/chaco.html>.
- [2] Noxim: network-on-chip simulator. URL <http://sourceforge.net/projects/noxim/>.
- [3] ECN Magazine. URL <http://www.ecnmag.com/articles/2011/01/urgent-need-digital-media-subscriber-modeling>.
- [4] HiPEAC Roadmap - 2013, . URL http://www.hipeac.net/system/files/hipeac_roadmap1_0.pdf.
- [5] HSA: Heterogeneous System Architecture Foundation, . URL <http://hsafoundation.com/>.
- [6] HSA: Heterogeneous System Architecture Foundation, . URL <http://chipdesignmag.com/sld/shuler/tag/hsa-foundation/>.
- [7] A. Agarwal, , et al. The MIT Alewife machine: architecture and performance. In *Proc. of ISCA 1995*, pages 2–13, 1995.
- [8] S. Agarwala et al. A 65nm c64x+ multi-core dsp platform for communications infrastructure. In *Proc. of ISSCC*, pages 262 –601, feb. 2007. doi: 10.1109/ISSCC.2007.373394.
- [9] Mohammad Abdullah Al Faruque et al. Adam: run-time agent-based distributed application mapping for on-chip communication. In *Proc. of DAC*, pages 760–765. ACM, 2008. ISBN 978-1-60558-115-6.
- [10] I. Anagnostopoulos et al. Custom microcoded dynamic memory management for distributed on-chip memory organizations. *Embedded Systems Letters, IEEE*, 2011.

- [11] David Atienza et al. Systematic dynamic memory management design methodology for reduced memory footprint. *ACM TODAES*, 11(2):465–489, 2006. ISSN 1084-4309. doi: <http://doi.acm.org/10.1145/1142155.1142165>.
- [12] Ke Bai and Aviral Shrivastava. Heap data management for limited local memory (llm) multi-core processors. pages 317–326, 2010.
- [13] Alexandros Bartzas et al. Software metadata: Systematic characterization of the memory behaviour of dynamic applications. *Journal of Systems and Software*, In Press, Corrected Proof:–, 2010. ISSN 0164-1212. doi: DOI:10.1016/j.jss.2010.01.001.
- [14] Adam Beguelin et al. Application level fault tolerance in heterogeneous networks of workstations. *J. Parallel Distrib. Comput.*, 43(2): 147–155, June 1997. ISSN 0743-7315. doi: 10.1006/jpdc.1997.1338. URL <http://dx.doi.org/10.1006/jpdc.1997.1338>.
- [15] L. Benini and G. de Micheli. Networks on chips: a new SoC paradigm. *Computer*, 35(1):70–78, 2002.
- [16] Emery D. Berger et al. Hoard: a scalable memory allocator for multithreaded applications. In *Proc. of ASPLOS, Cambridge, MA, USA*, pages 117–128. ACM, 2000. doi: <http://doi.acm.org/10.1145/356989.357000>.
- [17] D. Bertozzi et al. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE TPDS*, 16(2):113–129, Feb 2005.
- [18] M.K. Bhatti et al. In *Proc of DASIP*, pages 136 –143, 2010. doi: 10.1109/DASIP.2010.5706257.
- [19] R. Bitirgen et al. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proc. of MICRO-41*, pages 318 –329, Nov. 2008.
- [20] S. Borkar. Thousand core chips: A technology perspective. In *Proc. of DAC*, pages 746–749, 2007.
- [21] Hajo Broersma et al. The computational complexity of the minimum weight processor assignment problem. In *Proc. of WG*, pages 189–200, 2004.

- [22] Junwei Cao et al. Arms: An agent-based resource management system for grid computing. *Sci. Program.*, 10:135–148, April 2002. ISSN 1058-9244.
- [23] Ewerson Carvalho et al. Heuristics for dynamic task mapping in noc-based heterogeneous mpsoes. In *Proc. of IWRSP*, pages 34–40. IEEE Computer Society, 2007. ISBN 0-7695-2834-1.
- [24] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 14:141–154, February 1988. ISSN 0098-5589.
- [25] F. Cattoor et al. *Data access and storage management for embedded programmable processors*. Kluwer Academic Publishers, 2002.
- [26] Jean-Michel Chabloz and Ahmed Hemani. Lowering the latency of interfaces for rationally-related frequencies. In *ICCD*, pages 23–30, 2010.
- [27] J. Morris Chang and Edward F. Gehringer. A high-performance memory allocator for object-oriented systems. *IEEE Trans. Comput.*, 45(3):357–366, 1996. ISSN 0018-9340. doi: <http://dx.doi.org/10.1109/12.485574>.
- [28] M. Chaudhuri and M. Heinrich. SMTp: an architecture for next-generation scalable multi-threading. In *Proc. of ISCA*, pages 124–135, 2004.
- [29] Xiaowen Chen et al. Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller. In *Proc. of DATE, Dresden, Germany*, pages 39–44, 2010.
- [30] Chen-Ling Chou and Radu Marculescu. Incremental run-time application mapping for homogeneous nocs with multiple voltage levels. In *Proc. of CODES+ISSS*, pages 161–166. ACM, 2007.
- [31] D. Feitelson. Parallel Workloads Archive, <http://www.cs.huji.ac.il/labs/parallel/workload>. URL <http://www.cs.huji.ac.il/labs/parallel/workload>.
- [32] G. De Micheli. An Outlook on Design Technologies for Future Integrated Systems. *IEEE TCAD*, 28(6):777, 2009.

- [33] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL <http://doi.acm.org/10.1145/1327452.1327492>.
- [34] Travis Desell et al. Malleable applications for scalable high performance computing. *Cluster Computing*, 10(3):323–337, 2007.
- [35] Robert P. Dick et al. Tgff: task graphs for free. In *CODES'98*, pages 97–101, 1998.
- [36] Allen B. Downey. A model for speedup of parallel programs. Technical report, 1997.
- [37] Dror G. Feitelson and Larry Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Proc. of JSSPP*, pages 1–26. Springer-Verlag, 1996.
- [38] Kees Goossens et al. Aherreal network on chip: Concepts, architectures, and implementations. *IEEE Des. Test*, 22(5):414–421, 2005. ISSN 0740-7475.
- [39] Andreas Hansson et al. A unified approach to constrained mapping and routing on network-on-chip architectures. In *Proc. of CODES+ISSS*, pages 75–80. ACM, 2005. ISBN 1-59593-161-9.
- [40] T. Henderson, D. Kotz, and I. Abyzov. The Changing Usage of a Mature Campus-wide Wireless Network. In *Proc. of MobiCom*, 2004.
- [41] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing*, volume 95, page 285, 1995.
- [42] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2007.
- [43] K. Hirata and J. Goodacre. ARM MPCore; The streamlined and scalable ARM11 processor core. In *Proc. of ASP-DAC*, pages 747–748. IEEE Computer Society, 2007.
- [44] M. Horowitz et al. Low-power digital design. In *Proc. of SLPD*, pages 8 –11, oct 1994. doi: 10.1109/LPE.1994.573184.

- [45] J. Howard et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Proc. of ISSCC*, pages 108–109, feb. 2010. doi: 10.1109/ISSCC.2010.5434077.
- [46] Jingcao Hu and R. Marculescu. Energy- and performance-aware mapping for regular noc architectures. *IEEE TCAD*, 24(4):551–562, 2005. ISSN 0278-0070.
- [47] Information Sciences Institute. RFC 793: Transmission Control Protocol. <http://tools.ietf.org/html/rfc793>, 1981.
- [48] A. Iyengar. Parallel dynamic storage allocation algorithms. In *Parallel and Distributed Processing, 1993. Proceedings of the Fifth IEEE Symposium on*, pages 82–91, 1993.
- [49] Axel Jantsch and Hannu Tenhunen, editors. *Networks on chip*. Kluwer Academic Publishers, 2003. ISBN 1-4020-7392-5.
- [50] Sebastian Kobbe et al. DistRM: distributed resource management for on-chip many-core systems. In *Proc. of CODES+ISSS*, pages 119–128. ACM, 2011. ISBN 978-1-4503-0715-4. doi: 10.1145/2039370.2039392. URL <http://doi.acm.org/10.1145/2039370.2039392>.
- [51] Samuel Kounev et al. Towards self-aware performance and resource management in modern service-oriented systems. In *Proc. of SCC*, pages 621–624. IEEE CS, 2010. ISBN 978-0-7695-4126-6. doi: 10.1109/SCC.2010.94. URL <http://dx.doi.org/10.1109/SCC.2010.94>.
- [52] J. Kuskin, , et al. The Stanford FLASH multiprocessor. In *Proc. of ISCA*, pages 302–313, 1994.
- [53] Kevin Lai et al. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, 1: 169–182, August 2005. ISSN 1574-1702.
- [54] P. Larson and M. Krishnan. Memory allocation for long-running server applications. In *Proceedings of the 1st international symposium on Memory management*, pages 176–185, 1998.
- [55] D. Lea. A memory allocator, 1996. URL <http://g.oswego.edu/dl/html/malloc.html>.

- [56] Spyros Lyberis et al. The myrmics memory allocator: hierarchical, message-passing allocation for global address spaces. In *Proc. of Symp. on Principles and practice of parallel programming*. ACM, 2012.
- [57] Stylianos Mamagkakis et al. Energy-efficient dynamic memory allocators at the middleware level of embedded systems. In *Proc. of EMSOFT*. ACM, 2006. ISBN 1-59593-542-8.
- [58] T.G. Mattson, R.F. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core scc processor: the programmer’s view. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, 2010. doi: 10.1109/SC.2010.53.
- [59] Matteo Monchiero et al. Exploration of distributed shared memory architectures for NoC-based multiprocessors. *JSA*, 53(10):719–732, 2007. ISSN 1383-7621. doi: <http://dx.doi.org/10.1016/j.sysarc.2007.01.008>.
- [60] Srinivasan Murali and Giovanni De Micheli. Bandwidth-constrained mapping of cores onto noc architectures. *Design, Automation and Test in Europe Conference and Exhibition*, 2:20896, 2004. ISSN 1530-1591. doi: <http://doi.ieeecomputersociety.org/10.1109/DATE.2004.1269002>.
- [61] Srinivasan Murali and Giovanni De Micheli. Bandwidth-constrained mapping of cores onto NoC architectures. In *Proc. of DATE*, page 20896. IEEE Computer Society, 2004. ISBN 0-7695-2085-5-2.
- [62] K.J. Nesbit et al. Multicore resource management. *Micro, IEEE*, 28(3):6 –16, May-June 2008.
- [63] V. Nollet et al. Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In *Proc. of DATE*, pages 234–239. IEEE Computer Society, 2005. ISBN 0-7695-2288-2.
- [64] U.Y. Ogras and R. Marculescu. Application-specific network-on-chip architecture customization via long-range link insertion. In *Proc. of ICCAD*, pages 246–253, 6-10 Nov. 2005.

- [65] Paul Pop et al. An approach to incremental design of distributed embedded systems. In *Proc. of DAC*, pages 450–455. ACM, 2001. ISBN 1-58113-297-2.
- [66] R. Rajkumar et al. A resource allocation model for qos management. In *Proc. of RTSS*, pages 298 –307, 1997.
- [67] S. K. Reinhardt et al. Tempest and Typhoon: user-level shared memory. In *Proc. of ISCA*, pages 325–336, 1994.
- [68] Gerald Sabin et al. Moldable parallel job scheduling using job efficiency: an iterative approach. In *Proc. of JSSPP*, pages 94–114. Springer-Verlag, 2007. ISBN 978-3-540-71034-9. URL <http://dl.acm.org/citation.cfm?id=1757044.1757049>.
- [69] Bratin Saha et al. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proc. of Symp. on Principles and practice of parallel programming*. ACM, 2006.
- [70] T. Sakurai and A.R. Newton. Alpha-power law mosfet model and its applications to cmos inverter delay and other formulas. *Solid-State Circuits, IEEE Journal of*, 25(2):584 –594, apr 1990. ISSN 0018-9200. doi: 10.1109/4.52187.
- [71] Scott Schneider. Scalable locality-conscious multithreaded memor allocation. In *In Proc. of the 2006 ACM SIGPLAN International Symposium on Memory Management*, 2006.
- [72] Larry Seiler et al. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008. ISSN 0730-0301.
- [73] Semiconductor Industry Association. International technology roadmap for semiconductors, 2006. URL <http://www.itrs.net/Links/2006Update/2006UpdateFinal.htm>.
- [74] Semiconductor Industry Association. International technology roadmap for semiconductors, 2008. URL <http://www.itrs.net/Links/2008Update/2008UpdateFinal.htm>.
- [75] Mohamed Shalan and Vincent J. Mooney. Hardware support for real-time embedded multiprocessor system-on-a-chip memory management. In *Proc. of CODES, Estes Park, Colorado*,

- USA, pages 79–84. ACM, 2002. ISBN 1-58113-542-4. doi: <http://doi.acm.org/10.1145/774789.774806>.
- [76] Li Shang, Li-Shiuan Peh, and Niraj K. Jha. Powerherd: dynamic satisfaction of peak power constraints in interconnection networks. In *Proc. of ICS*, pages 98–108. ACM, 2003. ISBN 1-58113-733-8.
- [77] Youngsoo Shin et al. Power optimization of real-time embedded systems on variable speed processors. In *Proc. of ICCAD*, pages 365–368. IEEE Press, 2000. ISBN 0-7803-6448-1. URL <http://dl.acm.org/citation.cfm?id=602902.602984>.
- [78] SIA. Semiconductor industry association, international technology roadmap for semiconductors, 2011. URL <http://www.itrs.net/Links/2009ITRS/Home2011.htm>.
- [79] Lodewijk T. Smit et al. Run-time mapping of applications to a heterogeneous soc. In *Proc. of SoC*, 2005.
- [80] STMicroelectronics. STNoC: Building a new system-on-chip paradigm. White Paper, 2005.
- [81] T. Mattson, Rob van der Wijngaart. RCCE: A small library for many-core communication, <http://www.intel.com/content/www/us/en/research/intel-labs-rcce-single-chip-cloud-brief.html>. URL <http://www.intel.com/content/www/us/en/research/intel-labs-rcce-single-chip-cloud-brief.html>.
- [82] Justin Talbot et al. Phoenix++: modular mapreduce for shared-memory systems. In *Proc. of MapReduce*, pages 9–16. ACM, 2011. ISBN 978-1-4503-0700-0. doi: 10.1145/1996092.1996095. URL <http://doi.acm.org/10.1145/1996092.1996095>.
- [83] Shyamkumar Thoziyoor and Naveen Muralimanohar. Cacti 5.0, technical report hpl-2007-167, hp labs, 2007.
- [84] C. H. (Kees) van Berkel. Multi-core for mobile phones. In *Proc. of DATE*, pages 1260–1265. EDAA, 2009. ISBN 978-3-9810801-5-5.
- [85] S. Vangal et al. An 80-Tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS. In *Proc. of ISSCC*, pages 98–589. IEEE, 2007.

- [86] S. Vassiliadis et al. Microcode processing: Positioning and directions. *IEEE MICRO*, 23(4):21–30, 2003.
- [87] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator, 1996.
- [88] P. R. Wilson et al. Dynamic storage allocation: A survey and critical review. In *Proc. of IWMM, Kinross, Scotland, UK*, pages 1–116. Springer-Verlag, 1995. ISBN 3-540-60368-9.
- [89] Sotirios Xydis et al. Custom mutli-threaded dynamic memory management for multiprocessor system-on-chip platforms. In *Proc. of ICSAMOS, Samos Island, Greece*, pages 102–109, jul. 2010.
- [90] Terry Tao Ye, Luca Benini, and Giovanni De Micheli. Packetized on-chip interconnect communication analysis for mp soc. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, 2003.
- [91] Voon yee Vee and Wen jing Hsu. A scalable and efficient storage allocator on shared-memory multiprocessors. In *In International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN'99)*, pages 230–235, 1999.
- [92] Richard M. Yoo et al. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proc. of IISWC*, pages 198–207. IEEE Computer Society, 2009. ISBN 978-1-4244-5156-2. doi: 10.1109/IISWC.2009.5306783. URL <http://dx.doi.org/10.1109/IISWC.2009.5306783>.
- [93] Wangyuan Zhang and Tao Li. Managing multi-core soft-error reliability through utility-driven cross domain optimization. In *Proc. of ASAP 2008*, 2008.