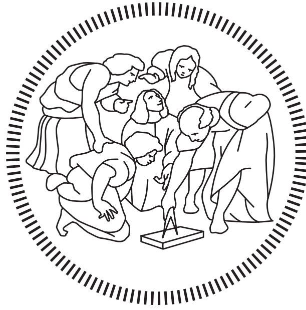


Politecnico di Milano
Department of Electronics, Information and Bioengineering
Doctoral Programme in Computer Science and Engineering

MULTICORE RESOURCE MANAGEMENT

A HORIZONTAL PERSPECTIVE

Doctoral Dissertation of
SIMONE LIBUTTI

Supervisor
PROF. WILLIAM FORNACIARI

Tutor
PROF. ANDREA BONARINI

The Chair of the Doctoral Program
PROF. ANDREA BONARINI

2017 – CYCLE XXX NOVEMBER

Simone Libutti
Multicore Resource Management: a Horizontal Perspective
© July 2017

"The very paradigm of explication, dear friends. Cogent, clear, if somewhat quaintly couched. Precision is a precise art. Poignancy is pre-eminent and precludes prevarication. Truths are no trivial thing, after all—"

— Steven Erikson, Memories of Ice

ABSTRACT

Modern computing systems strive to provide ever increasing performance levels despite increasingly strict system-wide optimization objectives. This is a vast problem that spans over a wide variety of architectures: due to the wild technological development caused by the spread of devices such as Smartphones, high-end embedded systems are quickly closing the gap with desktop computers; similarly, high performance and cloud-based systems are scaling up towards exascale to serve increasingly demanding workloads.

Indeed, this technological trend poses several, nontrivial problems: embedded systems are usually subject to thermal and energy constraints in order to maximize battery life, to minimize faults and, at least in the case of hand-held devices, to provide a comfortable user experience (users want to hold a long lasting, ever charged and cold device), while bigger systems are typically subject to thermal and power constraints in order to minimize supplying/cooling costs and, again, to prevent faults. On the other hand, users do not care about system optimization objectives: they just want their applications to comply with some Quality of Service requirement.

This problem does not have a simple solution, because system and user goals are orthogonal and increasingly demanding; however, it can be addressed by employing resource managers, which are software layers that act as brokers between computing systems and applications. Resource managers decide which and how many resources will be allocated to each application so that, whenever it is possible, both system-wide and user goals are complied with.

This dissertation explores the problem of resource management from a horizontal perspective. That is, we analyze the problem of CPU resource management spanning from high-end embedded to High Performance Computing systems. For each of those architectures, we try to understand what is yet missing to obtain an optimal resource management and how we can fill some of those gaps.

SOMMARIO

Gli odierni sistemi di calcolo mirano ad offire prestazioni sempre più aggressive nonostante sempre più stringenti vincoli di ottimizzazione di sistema.

Tale problema è complesso e riguarda una vasta gamma di architetture: a causa del selvaggio sviluppo tecnologico causato dalla diffusione di dispositivi come gli Smartphone, ad esempio, i circuiti integrati di fascia alta stanno rapidamente colmando il divario con i dispositivi desktop, in termini sia di prestazioni sia di complessità. Allo stesso modo, i sistemi ad alte prestazioni (supercomputer, cloud ...) stanno cercando di raggiungere potenze computazionali nell'ordine dell'exa-FLOP (10^{18} operazioni in virgola mobile al secondo), in modo da poter servire carichi di lavoro sempre più esigenti. Ciò comporta problemi non banali: i circuiti integrati sono solitamente soggetti a vincoli termici ed energetici per massimizzare la durata della batteria (ove presente), minimizzare i guasti e, per lo meno nel caso di dispositivi palmari, garantirne un comodo uso (l'utente vuole un dispositivo che duri anni, che abbia una batteria costantemente carica e che non sia bollente al tocco), mentre i grandi sistemi di calcolo sono soggetti a vincoli termici e di potenza onde minimizzare i costi di energia elettrica e raffreddamento e, anche in questo caso, per minimizzare la probabilità di guasto. Purtroppo, gli utenti non sono assolutamente interessati all'ottimizzazione del sistema; al contrario, vogliono soltanto ottenere le migliori prestazioni possibili dalle proprie applicazioni.

L'evidente ortogonalità tra gli obiettivi di utenti e amministratori di sistema fa sì che questo problema non sia facilmente risolvibile; tuttavia, esso può essere affrontato mediante l'uso di gestori di risorse, ovvero programmi che agiscono da mediatori tra sistema operativo e applicazioni. Più nel dettaglio, i gestori di risorse decidono quante e quali risorse di calcolo allocare a ogni applicazione in modo tale da soddisfare, ove possibile, sia i requisiti dell'utente sia quelli di sistema.

Questa tesi espora il problema della gestione di risorse seguendo un approccio orizzontale. Nel dettaglio, analizziamo il problema dell'allocazione di elementi processanti—cores, in questo caso—dai circuiti integrati fino ai sistemi ad alte prestazioni. Per ognuna di queste categorie, cercheremo di capire i requisiti fondamentali per una allocazione ottima delle risorse, i maggiori problemi ancora da affrontare e cosa possiamo fare per risolverli.

PUBLICATIONS

CONFERENCE PROCEEDINGS

2017 A. Portero, M. Podhoranyi, *Simone Libutti*, G. Massari and W. Fornaciari JUST-IN-TIME EXECUTION TO ADAPT ON DEMAND RESOURCE ALLOCATION IN HPC SYSTEMS 2017 International Conference on Algorithms, Computing and Systems (ICACS), ACM 2017

2017 N. Zompakis, M. Noltsis, L. Ndreu, Z. Hadjilambrou, P. Englezakis, P. Nikolaou, A. Portero, *Simone Libutti*, G. Massari and F. Sassi
HARPA: TACKLING PHYSICALLY INDUCED PERFORMANCE VARIABILITY
2017 Design, Automation & Test in Europe Conference & Exhibition (DATE) 97-102 2017 IEEE

2016 A. Portero, J. Sevcik, M. Golasowski, R. Vavrik, *Simone Libutti*, G. Massari, F. Catthoor, W. Fornaciari and V. Vondrák
USING AN ADAPTIVE AND TIME PREDICTABLE RUNTIME SYSTEM FOR POWER-AWARE HPC-ORIENTED APPLICATIONS
Green and Sustainable Computing Conference (IGSC0) 2016 Seventh International 1-6 2016 IEEE

2016 F. Reghenzani, G. Pozzi, G. Massari, *Simone Libutti* and W. Fornaciari
THE MIG FRAMEWORK: ENABLING TRANSPARENT PROCESS MIGRATION IN OPEN MPI
Proceedings of the 23rd European MPI Users' Group Meeting 64-73 2016 ACM

2016 C. Bolchini, S. Cherubin, G.C. Durelli, *Simone Libutti*, A. Miele and M.D. Santambrogio
A RUNTIME CONTROLLER FOR OPENCL APPLICATIONS ON HETEROGENEOUS SYSTEM ARCHITECTURES.
EWiLi 2016

2016 G. Massari, *Simone Libutti*, W. Fornaciari, F. Reghenzani, and G. Pozzi
RESOURCE-AWARE APPLICATION EXECUTION EXPLOITING THE BARBEQUERTRM
Proceedings of 1st Workshop on Resource Awareness and Application Autotuning in Adaptive and Heterogeneous Computing (RES4ANT), Dresden, Germany

2015 A. Portero, R. Vavrik, S. Kuchar, M. Golasowski, *Simone Libutti*, G. Massari, W. Fornaciari and V. Vondrak
Simulation of a runoff model running with multi-criteria in a cluster system
Proceedings of the Conference on Summer Computer Simulation 1-8 2015 Society for Computer Simulation International

2015 G. Massari, *Simone Libutti*, A. Portero, R. Vavrik, S. Kuchar, V. Vondrak, L. Borghese and W. Fornaciari
Harnessing Performance Variability: A HPC-Oriented Application Scenario
Digital System Design (DSD), 2015 Euromicro Conference on 111-116 2015 IEEE

2015 A. Portero, R. Vavrik, S. Kuchar, M. Golasowski, V. Vondrak, *Simone Libutti*, G. Massari and W. Fornaciari
Flood Prediction Model Simulation With Heterogeneous Trade-Offs In High Performance Computing Framework.
ECMS 115-121 2015

2015 D. Rodopoulos, S. Corbetta, G. Massari, *Simone Libutti*, F. Catthoor, Y. Sazeides, C. Nicopoulos, A. Portero, E. Cappe and R. Vavrik
HARPA: Solutions for dependable performance under physically induced performance variability
Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on 270-277 2015 IEEE

2014 D. Gadioli, *Simone Libutti*, G. Massari, E. Paone, M. Scandale, P. Bellasi, G. Palermo, V. Zaccaria, G. Agosta and W. Fornaciari
Opencl application auto-tuning and run-time resource management for multi-core platforms
Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on 127-133 2014 IEEE

2014 *Simone Libutti*, G. Massari, P. Bellasi and W. Fornaciari
Exploiting performance counters for energy efficient co-scheduling of mixed workloads on multi-core platforms
Proceedings of Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms 27 2014 ACM

international journals/transactions

2017 [Under review]
Simone Libutti, G. Massari and W. Fornaciari
On the Accuracy of CGroups-based CPU bandwidth Control

2017 [Under review]
Simone Libutti, G. Massari, A. Pupykina, G. Agosta and W. Fornaciari
Challenges in Runtime Support for Programming Models and Resource Management in Heterogeneous High Performance Computing

2015 *Simone Libutti*, G. Massari and W. Fornaciari
Co-scheduling tasks on multi-core heterogeneous systems: An energy-aware perspective
IET Computers and Digital Techniques Journal

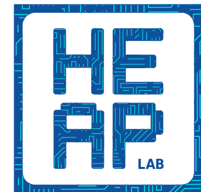
2015 R. Vavrik, A. Portero, S. Kuchar, M. Golasowski, *Simone Libutti*, G. Massari, W. Fornaciari and V. Vondrak
Precision-Aware application execution for Energy-optimization in HPC node system
arXiv preprint arXiv:1501.04557

ACKNOWLEDGMENTS

I'd like to sincerely thank all the members of my lab. Our group covers a lot of research areas; in fact, I never had the opportunity to collaborate with most of my lab fellows. Nonetheless, their constant presence was duly noted. In the life of a PhD student, good friends and a serene working environment are what really matters.

If you are willing to indulge me for a moment, I'd like to spend a couple of words about our research group (I'm shamelessly copy-pasting from the website).

Visit us at `http://www.heaplab.deib.polimi.it`



HEAP Lab (Politecnico di Milano, Italy) is a cross-disciplinary research team of around 20 people with skills covering Embedded and Cyber Physical Systems, Design Methodologies, Low-power Design of Software and Hardware, Compiler Construction, and Embedded Systems Security and Data Privacy.

The group has been active for over two decades in designing architectures and in developing methodologies and prototype tools to support the automation of different design phases of advanced embedded and computing systems.

I also wish to thank Prof. André Miede, who created the classicthesis Latex template based on Robert Bringhurst's seminal book on typography "The Elements of Typographic Style". Mr Miede, I'll definitely send you a postcard.

What to use the template? Check here: `http://www.miede.de`

CONTENTS

1 INTRODUCTION 1
1.1 The Resource Management Problem 1
1.1.1 From Single to Multi-Core 2
1.1.2 From Multi to Many-Core 3
1.1.3 Managing computing resources 4
1.2 The dissertation contribution in a nutshell 5
1.2.1 Addressing a real, complex problem 5
1.2.2 Dissertation Organization 7

I SINGLE-COMPUTING-NODE SYSTEMS 11

2 THE LINUX CONTROL GROUPS FRAMEWORK 13
2.1 Background 13
2.2 Control Groups in a nutshell 14
2.3 On the Accuracy of CGroups-based CPU bandwidth Control 17
2.3.1 Motivation and Background 17
2.3.2 Towards a fine-grained CPU bandwidth enforcement 18
2.3.3 Bandwdith allocation using the cgroups cpu controller 19
2.3.4 Simulating CPU bandwidth enforcement 21
2.3.5 Validation 28
2.3.6 Conclusions 34

3 RESOURCE ALLOCATION: SYSTEM-WIDE VS APPLICATION-SPECIFIC 37
3.1 Tailoring allocation to applications 37
3.1.1 From brute force prediction to application characterization 37
3.1.2 Selecting metrics to characterize resource contention 39
3.2 Energy-efficient co-scheduling using Performance Counters 40
3.2.1 A performance-counters-aware BarbequeRTRM 40
3.2.2 Experimental Results 44
3.2.3 Conclusions 48
3.3 Making applications adapt to allocations 48
3.3.1 Motivation 50
3.3.2 Methodology 52
3.3.3 Experimental Results 52
3.3.4 Conclusions 56

4 SINGLE-ISA HETEROGENEOUS PROCESSING: BIG.LITTLE ARCHITECTURES 59

4.1 Resource Contention in big.LITTLE Architectures . . . 59
4.1.1 Related Works 60
4.1.2 Methodology 61
4.1.3 Experimental Results 69
4.1.4 Conclusions 74
4.2 A heterogeneity-aware OpenCL support 74

II MULTIPLE-COMPUTING-NODES SYSTEMS 81

5 ENABLING A TRANSPARENT PROCESS MIGRATION IN OPEN MPI 83
5.1 Motivation 83
5.2 Related Works 84
5.3 Design and Implementation 87
5.3.1 Open MPI architecture 87
5.3.2 Open MPI extension 88
5.3.3 CRIU 90
5.3.4 Migration phases 91
5.4 Evaluation 92
5.4.1 Overheads due to multiple ORTE daemons per node 94
5.4.2 Overheads due to migration 98
5.5 Conclusions 102

6 CPU RESOURCE MANAGEMENT IN HPC SYSTEMS 105
6.1 Harnessing Performance Variability in HPC 106
6.1.1 Background 107
6.1.2 The HARPA Operating System 108
6.1.3 A feedback-based, performance-aware allocation policy 111
6.1.4 Experimental Setup 112
6.1.5 Conclusions 116
6.2 A Workload-Agnostic Resource Usage Optimization . 117
6.2.1 Background 118
6.2.2 Summary of the Work 119
6.2.3 A partially de-centralized resource management 120
6.2.4 Defining a throughput goal 121
6.2.5 Computing the ideal resource budget 123
6.2.6 The PerDeTemp Scheduling Policy 126
6.2.7 Resource allocation policy validation 128
6.2.8 Resource mapping policy validation 135
6.2.9 Conclusion 140

7 RESOURCE MANAGEMENT SUPPORT FOR DEEPLY-HETEROGENEOUS HPC 143
7.1 Overview 143
7.1.1 Resource Management 144
7.1.2 Memory Management 145
7.1.3 Programming Model Support 145

7.2 Programming Model Support 148
7.2.1 Message Passing Model 149
7.2.2 Shared Memory Model 150
7.2.3 Data Flow Model 151
7.2.4 Hybrid Models 152
7.2.5 Heterogeneous Platforms 152
7.2.6 Host-side low-level runtime 154
7.2.7 Device-side low-level runtime 155
7.2.8 Discussion 156
7.3 Runtime Management 157
7.3.1 Run-time resource manager 158
7.3.2 Distributed Management 160
7.3.3 Developing runtime-manageable applications . 161
7.3.4 Profiling runtime-manageable applications . . . 163
7.4 Memory Management 165
7.4.1 The MANGO approach 166
7.4.2 Choosing the most suitable memory. 167
7.4.3 Concurrent, thread-safe memory allocation and deallocation. 168
7.4.4 Runtime optimization. 169
7.5 Conclusions and Future Developments 169
8 CONCLUSIONS 171
8.1 Subjects covered by the dissertation 171
8.2 Contributions of the dissertation 172
8.2.1 Single-Computing-Node Systems 172
8.2.2 Multiple-Computing-Nodes Systems 173

III APPENDIX 177
A THE BARBEQUE RUN-TIME RESOURCE MANAGER 179
A.1 User-Space Resource Management 179
A.2 The BarbequeRTRM approach 180
A.2.1 Managed Applications Execution Flow 181
A.2.2 Integrating applications 182
A.3 Defining Resource Allocations 184

BIBLIOGRAPHY 187

LIST OF FIGURES

Figure 1 Evolution of microprocessors over time. 2
Figure 2 The end of Dennard scaling: active power vs Vdd. 3
Figure 3 Summary of HiPEAC Visions 2015 and 2017. . 6
Figure 4 Subjects covered by the dieesrtation. 7
Figure 5 The Linux Control Groups: cpuset inclusion. . 15
Figure 6 Dividing an application into monitorable Execution Cycles. 19
Figure 7 The Linux Control Groups: the cpu controller. 21
Figure 8 Enforcing CPU bandwidth on applications whose CPU usage is irregular over time. 22
Figure 9 Enforcing CPU bandwidth on applications whose CPU usage is regular and composed by bursts. 23
Figure 10 Execution Cycle usage profiles used during the simulation. 24
Figure 11 Bandwidth enforcement simulation: homogeneous scenario. 26
Figure 12 Bandwidth enforcement simulation: heterogeneous scenario. 27
Figure 13 Hardware setup. 30
Figure 14 Experimental results: equivalent core usage. . 31
Figure 15 Experimental results: equivalent core usage (average). 33
Figure 16 Spearman’s rank correlation coefficient for each performance counter, along with correlation ρ for each application. 42
Figure 17 Off-line applications characterization: correlating performance counters with energy consumption. 42
Figure 18 Off-line applications characterization: allocated CPU bandwidth vs performance. 43
Figure 19 Performance and energy speedups induced by CoWs on YaMS. 45
Figure 20 Linux scheduler compared to CoWs: workload execution time, system wide energy consumption and EDP. 47
Figure 21 Application and platform domain of the proposed methodology. 51
Figure 22 Design Space Exploration phase. 53

Figure 23 Throughput and percent disparity error for a plain Linux implementation and for the proposed run-time management strategy. 54
Figure 24 Dynamic workload analysis under a variable number of Stereo-Matching instances. 56
Figure 25 Schema representing the proposed approach. . 63
Figure 26 Flowchart describing the co-scheduling policy. 68
Figure 27 Stakes function examples on the big cores, with increasing number of co-running threads. . . . 72
Figure 28 Enforcing a custom system view on the OpenCL runtime. 76
Figure 29 Architecture of Open MPI modules. 88
Figure 30 Open MPI modules architecture used in the `mig` approach. 90
Figure 31 The mig framework: migration phases. 93
Figure 32 Execution time of each benchmark (input class = B) when running 16 processes using 1 to 16 ORTE daemons. 95
Figure 33 Execution time of each benchmark (input class = C) when running 16 processes using 1 to 16 ORTE daemons. 96
Figure 34 Execution time of each benchmark (input class = D) when running 16 processes using 1 to 16 ORTE daemons. 96
Figure 35 Experimental scenario used to asses the overheads induced by migration. 99
Figure 36 Size of the compressed process image normalized to the uncompressed size. 100
Figure 37 Time required to migrate a group of four processes. For each application, we list migration times for datasets type B and C, with and without compression. 101
Figure 38 Migration time composition (percentage). . . . 102
Figure 39 Overall view of HARPA-OS (applications side). 110
Figure 40 Main catchments and outlet hydrographs. . . . 113
Figure 41 Four instances of the Uncertainty application running on a system that features 48 cores. . . 115
Figure 42 Power consumption of a single instance of Uncertainty, in HARPA-OS-managed and unmanaged mode, on a 16-cores NUMA machine. . . 116
Figure 43 Example of interaction between BarbequeRTRM and the runtime library. 122
Figure 44 Runtime-library-side resource allocation. . . . 123
Figure 45 The PerDeTemp allocation policy. 127
Figure 46 Maximum throughput of the applications over 100 execution cycles. 129

Figure 47 Runtime Library overhead over 100 execution cycles. . . . 130
Figure 48 Managed application forwarding runtime profiles over 200 cycles. . . . 131
Figure 49 Error with respect to performance goal and CPU allocation over 100 execution cycles. . . . 132
Figure 50 Error with respect to performance goal and CPU allocation over 100 execution cycles, under runtime-variable performance requirements. 134
Figure 51 CPS over time for the three concurrently running applications 134
Figure 52 CPS and allocated CPU over time for the three co-running applications. . . . 135
Figure 53 Schema of the blade. Since the fan is on the right of the blade, there is an air cooling gradient between socket 1 and socket 0. . . . 136
Figure 54 Results for α scenarios, listing number of allocated cores and level of satisfaction. . . . 138
Figure 55 Results for β scenarios, listing number of allocated cores and level of satisfaction (the closer to 100%, the better). . . . 139
Figure 56 Heatmap of the computing node in managed and unmanaged configuration. In both experiments, we used the cpufreq performance governor. . . . 141
Figure 57 The MANGO architecture. . . . 146
Figure 58 The MANGO Software Stack. . . . 147
Figure 59 Hierarchical and distributed run-time resource management strategy. . . . 160
Figure 60 The synchronization mechanisms between the application execution and the resource manager (BarbequeRTRM) control actions. . . . 163
Figure 61 Example of user-space-managed environment. 179
Figure 62 Example of BarbequeRTRM-managed environment. . . . 180
Figure 63 The BarbequeRTRM: execution flow of a managed application. . . . 182

LIST OF TABLES

Table 1 Applications used during the tests. 29
Table 2 Average Power and energy consumption of the applications on the ODROID-XU3 board (big cluster). 66
Table 3 Results of the application characterization: CPU demand and minimum/maximum memory sensitivity. 70
Table 4 Summary of the big and LITTLE test scenarios. 73
Table 5 Summary of the big.LITTLE test scenarios . . . 78
Table 6 Experimental results. 79
Table 7 Problem data sizes (MB) for the A, B, C, and D classes of each benchmark. 94
Table 8 Static overhead of `IS` and `MG` with increasing migration granularity. 97
Table 9 Static overhead of `BT`, `SP`, and `LU` with increasing migration granularity. 98
Table 10 For each experiment, number of Monte Carlo samples to be performed by the instance that models each catchment. 137
Table 11 Allocation and access capabilities of both host and devices on the four OpenCL memory address spaces 153

LISTINGS

Listing 1 Pseudo-code of the performance-aware resource allocation policy. 112
Listing 2 Example of HLR use: a FIFO communication is set up between the host and the single kernel, which is loaded from an external file. 156
Listing 3 Example of DLR use: the DLR API is used to access the shared memory region, which is then read by the parallel tasks that are spawned from the main executor. 157
Listing 4 Example of MANGO application Recipe. . . . 164
Listing 5 Simplified example of a frame processing application in its non-integrated version. The application parses the arguments, initializes some data structure and selects a parallelism level. Then, it processes the frames in "threads_number"-sized bursts. Finally, it checks the results, joins the threads and terminates. 183
Listing 6 Main file of the integrated version of the application from Listing 5. The application initializes the RTLib and uses it to instantiate the class whose methods encapsulate the processing code. Then, it launches the processing and waits for it to terminate. 183
Listing 7 Simplified code of the integrated version of the application from Listing 5 (header). 184
Listing 8 Simplified code of the integrated version of the application from Listing 5 (implementation). Underlined functions are calls to the RTLib APIs. 185
Listing 9 Example of application recipe. A recipe is an XML file that contains a static list of Application Working Modes (AWMs). When computing the resource allocation for an application, the scheduling policy chooses an AWM between those that are listed in its recipe. . . . 186

ACRONYMS

AEM (BarbequeRTRM) Abstract Execution Model

API Application Program Interface

AWM (BarbequeRTRM) Application Working Mode

BarbequeRTRM The Barbeque Run-Time Resource Manager framework

BLCR Berkley Lab's Checkpoint/Restart kernel-space tool

cgroups The Linux Control Groups framework

CPS (BarbequeRTRM) Execution Cycles Per Second

CPU Central Processing Unit

C/R Checkpoint/Restart

CRIU The Checkpoint/Restore In Userspace tool

DSE Design Space Exploration

DVFS Dynamic Voltage and Frequency Scaling

EC (BarbequeRTRM) Execution Cycle

EDP Energy-Delay Product

FPGA Field Programmable Gate Array

FLOP Floating Point OPeration

GN (MANGO) General Purpose Computing Node

GPGPU General-purpose computing on GPU

GPU Graphic Processing Unit

HARPA (EU Project) HARnessing Performance vAriability

HARPA-OS (HARPA) HARPA Operating System

HARPA-RTE (HARPA) HARPA Run-Time Engine

HiPEAC European Network on High Performance and Embedded Architecture and Compilation

HMP Heterogeneous Multi-Processing

HN (MANGO) Heterogeneous Computing Node

HNP (OpenMPI) Head Node Process

HPC High Performance Computing

HSA Heterogeneous System Architecture

I/O Input/Output

ISA Instruction Set Architecture

JPS (BarbequeRTRM) Executed Jobs Per Second

LLC Last Level Cache

MANGO (EU Project) Exploring Manycore Architectures for Next GeneratiOn HPC systems

MCA Modular Component Architecture

MC MonteCarlo

MPI Message Passing Interface

NP Nondeterministic Polynomial time

NUMA Non-Uniform Memory Access

OMPI (OpenMPI) OpenMPI application-side API

OPAL (OpenMPI) Open Portable Access Layer

ORTE (OpenMPI) Open Run-Time Environment

PE Processing Element

RAM Random Access Memory

RTRM Run-Time Resource Manager

RTlib (BarbequeRTRM) The application Run-Time Library

R/R Rainfall-Runoff

SoC System on Chip

SMT Simultaneous Multi-Threading

TBB Intel Threading Building Blocks

QoS Quality of Service

VM Virtual Machine

VPS Virtual Private Server

1 INTRODUCTION

This chapter provides an introduction for the dissertation. In Section 1.1, we describe the factors that drove the evolution of processors towards multi and many-core and we explain the role of resource management in modern multi/many-core systems. In Section 1.2, we further detail the motivations of this dissertation and we provide a high-level overview of our original contributions.

THE RESOURCE MANAGEMENT PROBLEM

In 1965, the Intel co-founder Gordon Moore observed that, during the following decade, the number of transistors that could cost-effectively fit in a single chip would double once per year [1]. In 1975, he corrected the pace to a doubling every two years [2]. This market and technological trend, which proved to be true and drove the evolution of processors design and manufacturing for decades, became widely known as the *Moore's law*.

In 1974, the IBM researcher Robert H. Dennard co-authored a paper according to whom transistors power is proportional to their area because the smaller the transistor area, the lower current and operating voltage a transistor needs to properly work [3]. This law, which allowed manufacturers to drastically raise clock frequencies from one generation to the next without significantly affecting the overall circuit power consumption, became widely known as *Dennard scaling*.

Moore's law and Dennard scaling had a tremendous effect on the evolution of processors, inasmuch as they led to chips that featured ever higher frequencies and transistors numbers. However, starting from the first decade of this millennium, keeping up with this trend became increasingly difficult. On one hand, Moore's law is beginning to slow down. As shown by Figure 1, the number of transistors per chip (red circles) continued sticking to the doubling trend; however, as recently stated by Intel, shrinking the transistors in a cost-effective way is becoming more difficult, and, starting from 2017, the pace between successive generations of processors will be officially slowed down to a doubling every three years [4]. It is worth remarking that, regardless of the production costs, a Skylake transistor is already around 100 atoms across; although emerging technologies are pushing towards even lower transistor sizes [5], we are clearly reaching the limits imposed by physics. On the other hand, Dennard scaling already ended, and this is the reason that initially drove processors design towards multi-cores. The transistor power

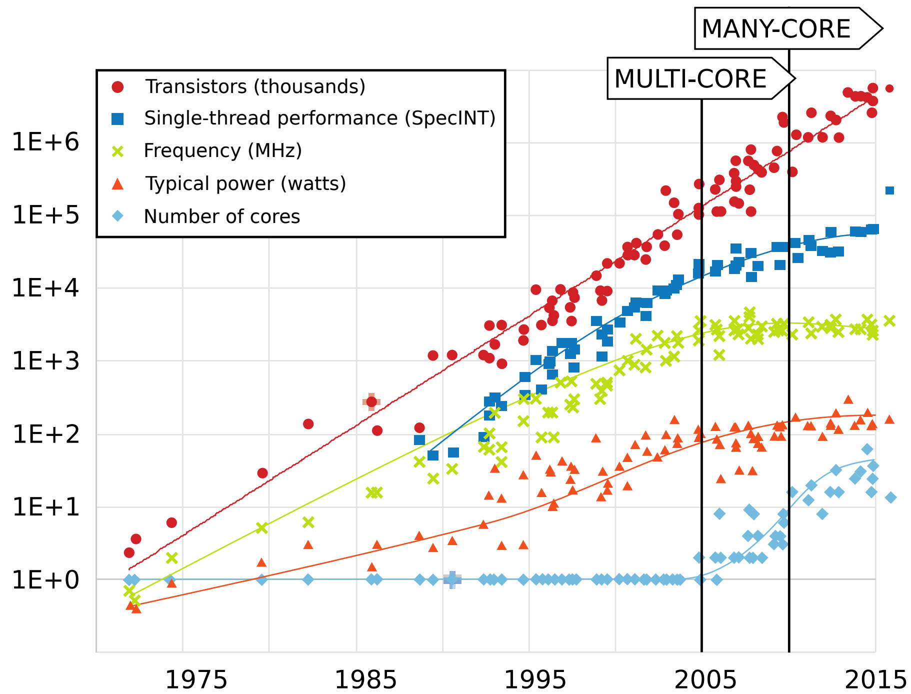


Figure 1: Evolution of microprocessors over time [7]. From the first years of this millennium, the end of Dennard scaling caused frequency to stagnate and led to the rise of multi and many-cores.

consumption as computed by Dennard takes into account only the dynamic power, i.e., the power consumed by the transistor to switch state. However, due to miniaturization, the leakage power, which is the power consumed because of leakage current, is no longer negligible (see Figure 2). Consequently, scaling threshold voltage—hence operating voltage—became difficult if not unfeasible [6], and frequencies could not be further raised without affecting power consumption.

From Single to Multi-Core

When the Dennard scaling started to falter, the potential benefits of parallelism (at least when employing multiple independent processors) were already a well established concept. This explains why, in the early 2000s, processors design went multi-core (light blue diamonds in Figure 1). The main idea was to provide each chip with multiple energy-efficient—hence slower—processing cores instead of an increasingly powerful one [9]. In a typical multi-core processor, the cores share the last level cache and the memory hierarchy, so that threads from different cores can locally cooperate in executing applications. This way, even though each core is less performing than that of a typical single-core processor, the overall performance is higher due to the concurrent execution of parallel threads.

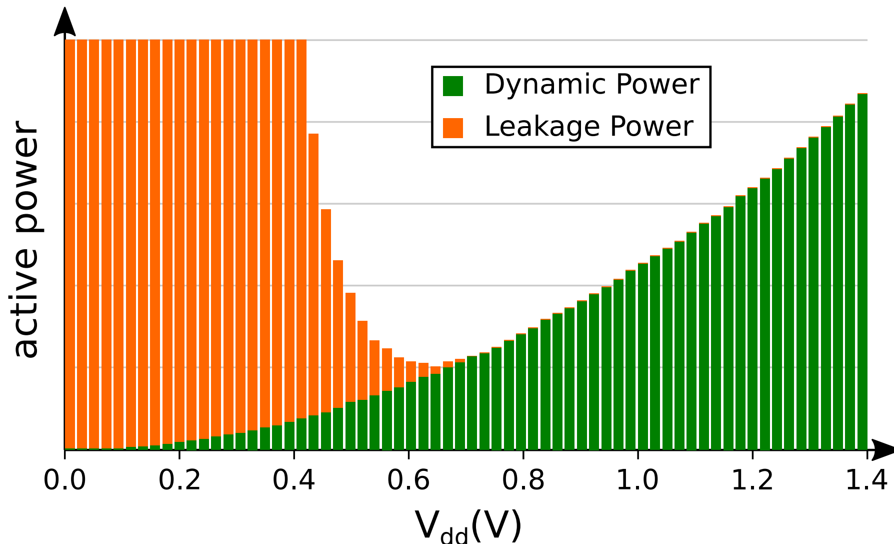


Figure 2: Active power vs Vdd at constant frequency [8]. As the Vdd gets smaller, active power gets higher due to the rise of leakage power.

Indeed, producing multi-core processors is only half of the solution: applications and operating systems must be specifically designed to leverage parallelism. Developers had to learn how to split applications into suitable pieces that could be locally executed by multiple cores, and this, obviously, had a tremendous effect on the complexity of the applications source code.

Even though it added complexity to both the hardware and the software side, multi-core processing proved very effective in boosting processors performance while avoiding harsh increments of power consumption. However, due to the presence of multiple processing elements and of shared resources such as the memory hierarchy, selecting a suitable amount of parallelism for each application and understanding how the execution of an application affects the performance of the rest of the workload became increasingly important.

From Multi to Many-Core

In 1967, well before the advent of Dennard scaling, the IBM researcher Gene M. Amdahl formulated what became known as the *Amdahl's law* [10]: the speedup obtainable by splitting a task among multiple devices affects only the parallel parts of the applications code, whereas the serial parts are not affected. That is, there are limitations to the maximum amount of parallelism an application can benefit from. Multi-core processors introduce yet another degree of complexity, as the Amdahl's law refers to applications that are split among several single-core processors. In the case of multi-core processing, the presence of shared resources—and hence of resource contention—further limits the potential parallelism-induced speedup [11].

Amdahl's law states that if P and $1-P$ are respectively the parallel and serial portions of a system or program, then the maximum speedup achievable by using N processors is $\frac{1}{(1-P)+(P/N)}$.

Given these premises, how far can we push the multi-core approach? According to the Amdahl's law, the limitations of parallelism-induced speedup depend on the applications code. In case of embarrassingly parallel applications, i.e., applications that can be effortlessly parallelized into a massive amount of independent tasks (e.g. graphical processing), the multi-core approach can be indeed pushed to its limits, and this led to the design of processors that featured tens to hundreds of cores (e.g. GPUs, which are manycore vector processors). Obviously, in order to cope with the subsequent increase of complexity, such cores are usually kept as simple as possible: most of the complexity is instead moved to the software level by introducing ad-hoc programming models that support *explicit parallelism* (e.g. OpenCL and CUDA).

Nowadays, the typical setup of high-end embedded to High Performance Computing systems consists of one or more multi-core processors and one or more many-core accelerators, the latter used to execute only specific kinds of tasks. For small systems such as Smartphones and Desktop computers, the common trend is to have a multicore processor and a GPU.

Managing computing resources

The problem of improving performance while guaranteeing a low power consumption is not yet solved. Indeed, multi and many-cores proved to be a very effective mean to mitigate this issue, but they also introduced a non-negligible degree of complexity both at hardware and at software level. Moreover, generation after generation of processors, the increasing performance caused the chips power consumption to constantly raise up to the point where it is not possible to power on all the components of a chip without causing a burnout [12]. This problem is referred to as *dark silicon problem*, meaning that some parts of the chip must be switched off (made dark) in order for the remaining ones to be able to work.

To summarize, modern systems are subject to multiple problems: first of all, the available computing resources consist in *several* and possibly *heterogeneous* cores. Each application may have a different *Quality of Service* requirement to comply with, a different *degree of parallelism* and, due to the presence of shared resources, a different *interference* on the performance of concurrently running applications. Second, the high power consumption of modern systems causes serious issues such as thermal hot-spots [13], performance variability induced by aging [14] and, in case of large systems, unsustainable supplying and cooling costs [15].

Addressing the aforementioned issues is paramount; however, doing so requires to add a huge degree of complexity to either (or all) hardware, operating system or software side. A valid alternative ap-

proach is instead to employ a software layer that acts as a broker between operating system and applications and that hides the complexity of orchestrating hardware configuration (e.g. setting cores frequencies according to system status and current workload) and resource allocation. These layers are called *Resource Managers*.

THE DISSERTATION CONTRIBUTION IN A NUTSHELL

This dissertation analyzes the problem of multi and many-core resource management from a horizontal perspective. We analyze the main goals and issues regarding the allocation of CPU resources (primarily cores), ranging from high-end embedded to HPC systems.

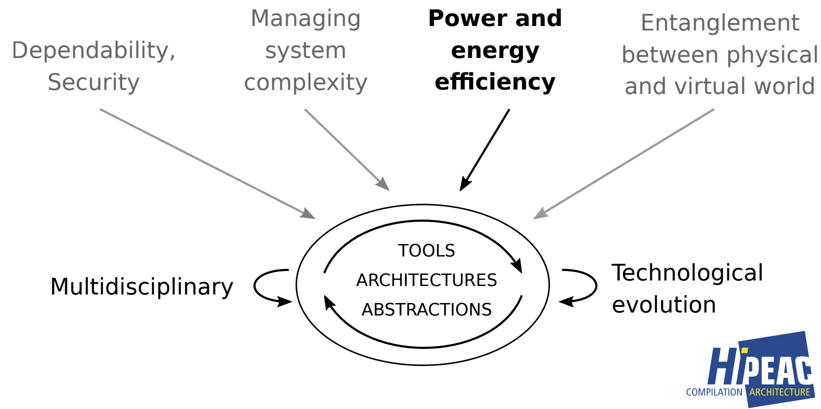
Addressing a real, complex problem

Figure 3 shows the most recent research road-maps suggested by the European Network on High Performance and Embedded Architecture and Compilation (HiPEAC). These road-maps, which are also called HiPEAC visions, deal with the most critical challenges that are faced by academia and industry and that are currently driving the European research activity. One of the main topics of the HiPEAC vision 2015 (Figure 3a) was achieving power and energy efficiency by developing formalisms, methodologies and tools to deal with the "desired Quality of Service" of applications: by assigning to applications only the computational resources that they need to comply with their Quality of Service goals, it is indeed possible to reduce over-specification. In the HiPEAC vision 2017 (Figure 3b), energy efficiency still remains a major challenge, and not only for the environment sake: in order to stay functional and economically viable, both Exaflops computers and battery-based embedded systems need significant improvements to energy efficiency. The HiPEAC vision 2017 also stresses the fact that, in order to address the aforementioned challenges, it is paramount to successfully master the parallelism and heterogeneity opportunities that are offered by modern systems.

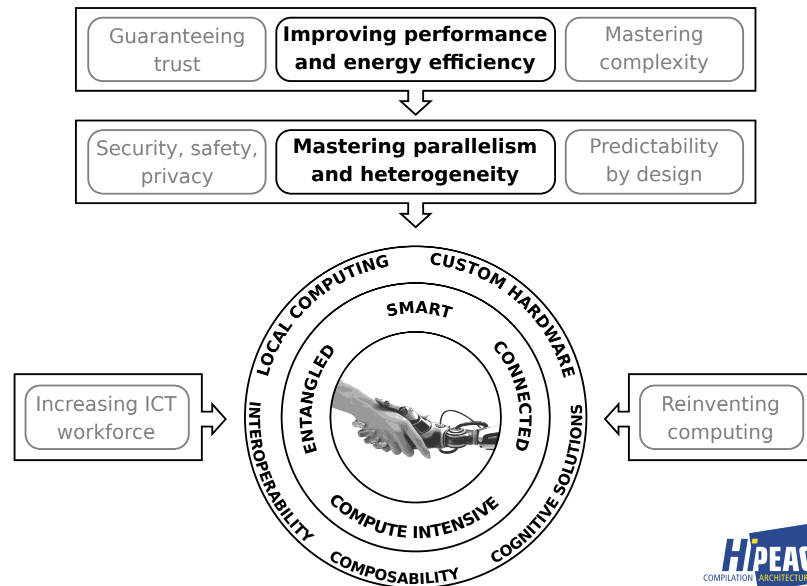
Most applications are not interested in executing "as fast as possible". Rather, they are content with being executed "at least that fast". The same thing happens with QoS, which, in any case, can be often related to performance.

This dissertation specifically tackles those challenges. Figure 4 provides an immediate representation of the different subjects and scenarios that we address in the dissertation. From the resource management perspective, we mostly focus on scheduling policies. That is, we try to understand, depending from the target architecture and the optimization goals, how the resource manager can compute the most suitable resource allocation for each application. We also analyze how the Linux operating system supports resource management across different architectures, and, when needed, we implement new mechanisms to support resource allocation.

We exploit resource management techniques to address the problems described in Section 1.1: maximizing performance and minimiz-



(a) The HiPEAC vision 2015 [16].



(b) The HiPEAC vision 2017 [17].

Figure 3: The latest High Performance and Embedded Architecture and Compilation (HiPEAC) visions, listing power/energy efficiency and the exploitation of parallelism and heterogeneity as major challenges for the European research community.

ing power (mostly HPC) and energy consumption (mostly battery-based embedded), but also minimizing thermal hot-spots and mitigating the effects of memory contention and performance variability. We do this by trying to understand which is the minimum amount of resources that applications need to comply with their Quality of Service goals.

Regarding architectures, we evenly focus on embedded, desktop and HPC systems. For each of those, we address both homogeneous and heterogeneous scenarios.

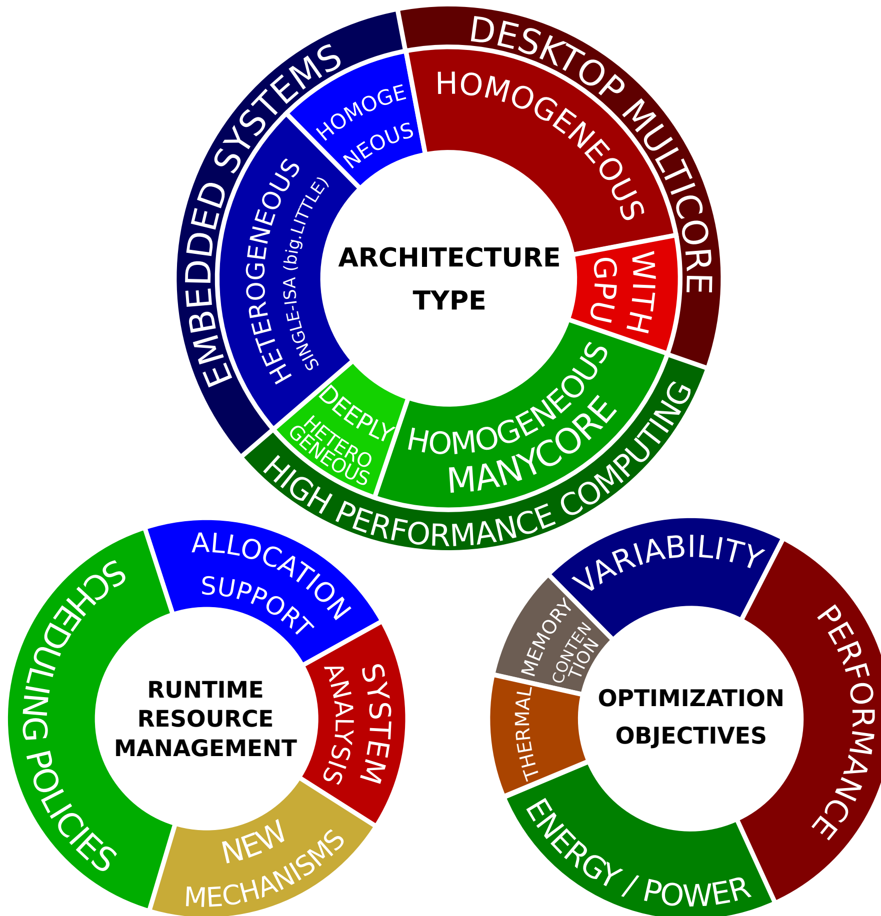


Figure 4: A horizontal perspective of multi and many-core resource management. The pie charts provide an immediate representation of the distribution of management-related subjects in the dissertation.

Dissertation Organization

This dissertation is split in two parts. The first one, which is composed by Chapters 2 to 4, bundles all the works that pertain to single computing node architectures. Part of those works are indeed applicable also to HPC systems, which usually consist of multiple interconnected computing nodes and can therefore benefit from a suitable node-level resource management support.

When dealing with distributed systems, resources must be suitably managed at both system and node-level, hence the division into "single vs multi-node approaches" instead of "embedded and desktop vs HPC".

First of all, we analyze the resource manager problem from the operating system standpoint. We do that in order to understand how the resource allocation choices of the resource manager are actuated on the system. In particular, we present the Linux Control Groups (also referred to as cgroups), which is a Linux framework that is used to allocate computing resources or to put constraints on their usage. After a brief introduction on the structure of cgroups, we present a study on the accuracy of cgroups-based CPU bandwidth control. We

show that, in order for CPU time enforcement to be accurate, the configuration of cgroups needs to be set for each application separately.

Second, we move to the resource management layer. In particular, we deal with application characterization, i.e., how to extract application features that can be effectively exploited by resource managers to perform clever allocation choices. We also try to understand how to suitably distribute the management logic among the resource manager and applications. In particular, we investigate the synergies between system-wide resource management and application-specific auto-tuning.

Finally, we focus on big.LITTLE architectures, which are multi-core heterogeneous single Instruction Set Architecture processors that, by featuring different types of cores in the same chip, allow operative systems to exploit the trade-off between performance and power consumption. We try to understand how such architectures are dealt with by modern systems. More in detail, we study how to support the Linux Heterogeneous Multi-Processing scheduler in performing energy-aware scheduling and how to provide the OpenCL runtime with heterogeneity-awareness when using big.LITTLE processors as heterogeneous OpenCL devices.

Once having achieved a suitable resource management support at node level, we go multi-node. The second part of the dissertation, which is composed by Chapters 5 to 7, specifically targets distributed systems such as those used in HPC. Such systems are usually composed by interconnected computing nodes, and this poses questions that are not present in multi-node scenarios, such as how to efficiently migrate applications and their data from one node to another; how to minimize power consumption in order to mitigate the high supplying and cooling costs of large systems; and how to minimize (or deal with) the presence of faults, whose probability gets higher as the size of the system increases.

First of all, we perform and interesting study on how the freeze/restore-based process migration of MPI applications, which is usually performed at node granularity to address faults, can be made fine-grained in order to migrate only parts of the application on a different computing node. This allows resource managers to perform optimizations such as load balancing, resource consolidation, or also to counteract the effects induced on the hardware by aging (i.e., migrating processes from faulty to healthy nodes).

Second, we present a resource management approach that exploits the trade-off between power consumption and performance when executing HPC applications that must comply with runtime-variable Quality of Service requirements. In this context, we will employ an approach composed by an application-agnostic, feedback-based resource allocation and a performance and resource-aware mapping strategy. The goal of such approach is to make applications comply

with their quality requirements while minimizing the effects of performance variability and evenly spreading temperature throughout the chip.

Finally, we perform the first steps towards an unified runtime management support for deeply heterogeneous HPC systems.

We close the dissertation with the conclusion (Chapter 8) and the Appendix. In the former, we summarize the problems we encountered during our study and how we choose to solve them; in the latter, we describe the BarbequeRTRM, the runtime resource manager that we extensively used, improved and developed during our work.

Part I

SINGLE-COMPUTING-NODE SYSTEMS

The first part of this dissertation deals with *single computing node systems*.

First of all, we will study how the Linux Control Groups actuate CPU time allocation in multi-core processors. We will show that, in order for the enforcement of CPU time allocation to be accurate, the configuration of cgroups must be separately set for each application.

All the works presented in this disseration actuate the allocation of CPU cores by using Control Groups

Second, we will address CPU resource management in homogeneous nodes, and we will do so with a dual approach. On one hand, we will study how the computation of resource allocation can take into account the characteristics of applications. On the other hand, we will try to understand how applications can change their behavior in order to optimally exploit a resource allocation. We will show that the synergistic collaboration between system-wide resource managers and application-specific application auto-tuners leads to quite promising results.

Finally, we address single-ISA heterogeneous processing. In particular, we will study how to support the Linux Heterogeneous Multi-Processing scheduler in performing energy-aware scheduling and how to provide the OpenCL runtime with heterogeneity-awareness when using single-ISA heterogeneous processors as heterogeneous OpenCL devices. Those works specifically target big.LITTLE processors, i.e., processors that feature a cluster of performing but power-hungry cores and a cluster of low-performance and power-efficient ones. Given that the two clusters share the same ISA, threads can be freely migrated between clusters at runtime in order to leverage the trade-off between performance and power efficiency.

Single-ISA heterogeneous processors feature different computing cores that share the same Instruction Set Architecture. This means that an application binary can be indifferently executed on (or migrated to) any available core.

2 THE LINUX CONTROL GROUPS FRAMEWORK

In this chapter, we will briefly introduce the Linux Control Groups (also referred to as cgroups), a Linux kernel feature that allows system administrators to limit the amount of resources that can be used by any group of processes.

BACKGROUND

In 1979, the `chroot` (from "change root") system call was introduced in Unix. The purpose of `chroot` is to create and host a virtualized copy of the software system and to use it for testing, recovery and security purposes. In 1992, in a quite interesting paper where `chroot` was used to isolate the activities of a cracker, Bill Cheswick created the term "chroot jail" [18]. This term—and some years later, also the term "jailbreak" [19]—quickly become quite popular, and that is why the virtualization mechanism introduced in 2000 in FreeBSD is called jail [20]. The `chroot` approach, however, is subject to two major limitations: first of all, the isolation provided by `chroot` can be bypassed by tasks that have root privileges; second, `chroot` offers limited resource allocation capabilities, e.g., it does not support memory quotas, I/O and CPU quota limiting and network isolation.

Jailbreaking is a well known term, especially for Apple users: it refers to the process by which an iPhone firmware is modified to allow unsigned code to gain privileged access to the system's files.

In order to overcome those limitations, several companies endowed each physical server with a hypervisor, which could be either native (type 1) or hosted (type 2) and allowed multiple Virtual Machines to run on the same physical system. Indeed, each VM had its own dedicated resources (e.g., CPU, memory, I/O). This approach is often referred to as Server Virtualization.

As the number of Virtual Machines increased, there was a strong push towards more lightweight virtualization techniques. This led to the concept of software container, i.e., a way to encapsulate applications into lightweight virtualized boxes that run on top of a single operating system. In 2001, Jacques Gélinas created the VServer project, which aimed at separating the user-space environment into distinct units called Virtual Private Servers (VPS), so that each VPS could be seen as a real server by the processes that it contained. Running multiple servers on the same kernel allows system administrators to avoid virtualization overheads and to dramatically enhance the efficiency of resource usage by allowing a single kernel to dynamically allocate resources to each VPS [21].

The Linux Control Groups were originally introduced in 2006 under the name "process containers" [22]. In 2008, the process contain-

ers mechanism was merged into the Linux kernel mainline (kernel version 2.6.24). The name was then changed to "Control Groups" in order for it not to be confused with the concept of software container. Control Groups are not containers themselves: they are a mechanism that can be used in Linux-based systems to perform resource isolation, and this in turn enables the creation of containers [23].

Control Groups enforce a specific view of the system on applications: access to resources such as processing elements, memory nodes or external accelerators can be totally denied to some applications, thus partially isolating a subset of resources from the rest of the system; access to inherently shared resources such as memory, network and CPU time, conversely, can be limited in order to minimize resource contention or to guarantee different Quality of Service levels to each class of applications.

At the time of writing, there are two orthogonal implementations of Control Groups: cgroup-v1 and cgroup-v2. In this dissertation we focus on the former, which is more widespread (cgroup-v2 has been declared non-experimental starting from Linux Kernel 4.5 [24]).

CONTROL GROUPS IN A NUTSHELL

A control group is a collection of processes that share the same view of the available resources, i.e., each of the processes that belong to the same control group is subject to the same parameters and resource usage constraints.

Control Groups are hierarchical: the children of a cgroup explicitly inherit the constraints of their parent. Figure 5 shows a typical example of parent-to-child inheritance. In this case, cgroups are used to allocate a subset of the available processing elements to different classes of tasks. The *root cgroup*, which is mounted by default and has got no parent, is able to access all the available processing elements; therefore, depending on the chosen configuration, all its children may be also able to use them. Control Group A, which is a child of the root cgroup, has been configured so that it can access all the processing elements but PE 5 and PE 8 (according to the cgroup notation, the *cpuset* of A is composed by PEs $0-4$, $6-7$). Its children will therefore inherit this constraint: regardless of the chosen configuration, each of the children of cgroup A will be unable to use PE 5 and PE 8. The allocation of cgroup B must be therefore equal or stricter than that of cgroup A: in this case, B is able to access PEs $0-4$, which are a subset of the ones that are accessible by A.

Usually, the terms "Processing Element" and "core" are interchangeable. In case of hyper-threading, however, a core features two PEs instead of one.

The set of constraints a cgroup is subject to is provided by one or more *subsystems*, which are modules that rely on the cgroups task grouping facilities to treat groups of tasks in a particular way. For instance, the control hierarchy that is shown in Figure 5 is constrained

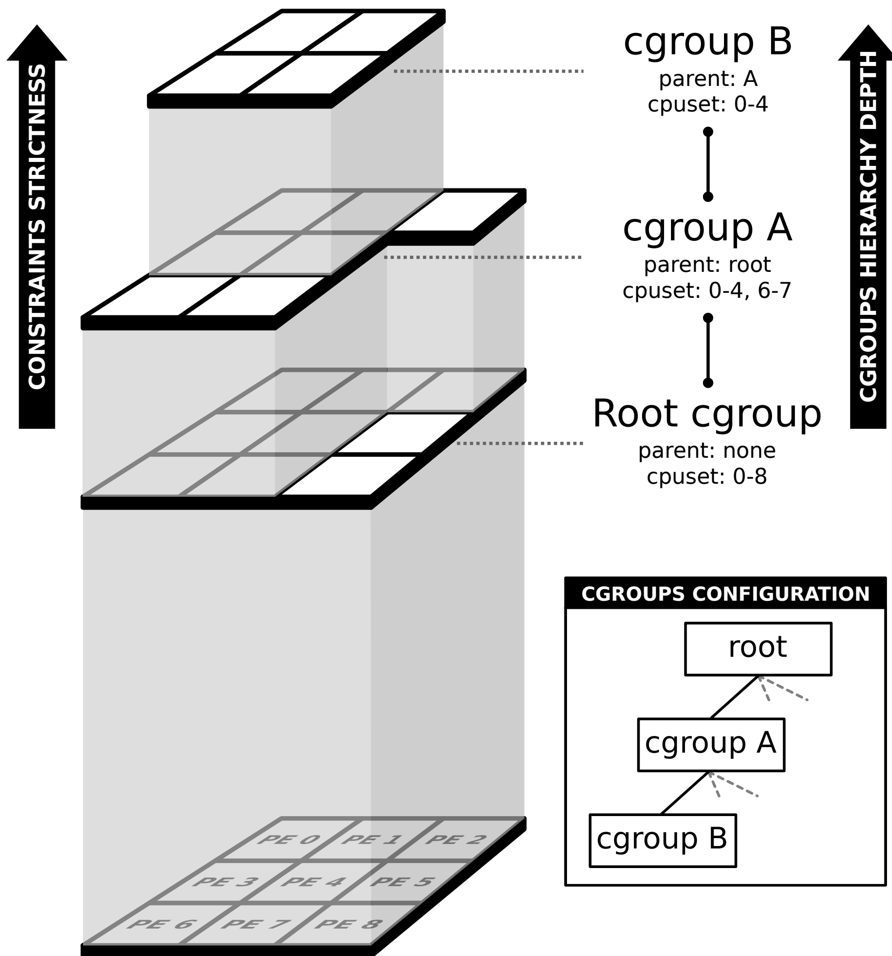


Figure 5: Control Groups hierarchy example: cpuset inclusion. The root cgroup is able to access all the nine processing elements that compose the system. Its child (cgroup A), conversely, has been configured so that it can access only some of them. The child of A (cgroup B) inherits the limits of its parent: all the processing elements that are not accessible by A are therefore explicitly denied to B. In general, the resource allocation of a child is always equal or stricter than that of its parent.

by a single subsystem, i.e., *cpuset*. Indeed, each of the resources that are handled by cgroups is managed by a specific subsystem.

The available subsystems are:

BLKIO

The Block Input/Output controller. At the time of writing, there are two I/O control policies. The first one uses the Linux Completely Fair Queuing (CFQ) I/O scheduler [25] to perform a weighted time-based division of disk access. The second policy, conversely, allows system administrators to specify upper I/O rate limits on devices, and it enforces the allocation by performing throttling.

CPU
The CPU bandwidth controller. It imposes constraints on the Linux Completely Fair Scheduler (CFS) [26], thus limiting the CPU time that is allocated to a group of tasks during a given time frame. This controller can also be used to assign a relative priority to a cgroup, so that its tasks are scheduler more often with respect to the tasks of other cgroups. The cpu controller also supports the Linux Real Time scheduler.

CPUACCT
The CPU accounting controller. By using this subsystem, system administrators are able to bundle multiple tasks together, so that their CPU usage can be jointly monitored.

CPUSETS
The cpuset controller. It allows system administrators to assign a set of CPUs and — usually in NUMA systems — memory nodes to groups of tasks, hence isolating the tasks in a subset of the available resources. It is worth noticing that system calls such as sched_setaffinity and mbind can be used in conjunction with this controller, but such requests are filtered through the cpuset of the invoking task.

For more information about sched_setaffinity and mbind, see `http://man7.org/linux/man-pages/man2/sched_setaffinity.2.html` *and* `http://man7.org/linux/man-pages/man2/mbind.2.html`*.*

DEVICES
Device whitelist Controller. It associates a device access whitelist to a group of tasks.

FREEZER
The checkpoint/restart subsystem. It is usually used in HPC clusters to define the set of tasks that must be started/stopped by the batch job management system, but it can also be used to stop tasks in order for them to be either restarted after a fault or migrated to another computing node.

HUGETLB
The HugeTLB controller, which limits the usage of huge pages. It is worth noticing that, as for the memory controller, tasks that surpass their limit will receive an error signal. Tasks must therefore be aware of their limit and behave accordingly. It follows that the usage of this controller is not transparent to tasks.

MEMORY
The memory controller. It limits the amount of memory that can be used by a group of tasks, and it is mostly used to prevent a group of memory-hungry tasks from monopolizing the entire system memory. As for the hugetlb controller, tasks must be aware of their limit; otherwise, depending on the controller configuration, they could be either killed or stopped until the system administrator chooses to grant them additional memory.

NET_CLS
Network packets classifier subsystem. Its only purpose is to tags packets with a class identifier. This subsystem must be used in conjunction with the Linux Traffic Controller (tc), which uses the identifiers to enforce the selected bandwidth limit on the packets.

For more information avout the Linux Traffic Controller, see https://linux.die.net/man/8/tc.

NET_PRIO
The Network priority subsystem. It dynamically sets the priority of network traffic generated by a set of tasks on a given interface.

PIDS
The process number controller. It is used to stop any new tasks from being forked or cloned after a certain limit is reached.

RDMA
The RDMA controller. It allows system administrators to limit RDMA/InfiniBand specific resources for a given set of tasks.

ON THE ACCURACY OF CGROUPS-BASED CPU BANDWIDTH CONTROL

In this section, we analyze how the Linux Control Groups perform CPU bandwidth control. First of all, we employ simulation to demonstrate that, depending on the CPU demand of the applications, Control Groups may dramatically fail in limiting CPU bandwidth. Then, we perform tests using real applications to show how Control Groups can be properly configured to achieve an accurate enforcement accuracy.

The contents of this section are partially submitted in a Transaction. They are currently under review.

Motivation and Background

Modern architectures feature a high number of shared resources that are used to concurrently run multiple tasks or to accelerate parallel ones. Performance increase, however, comes at a cost: the power consumption of those architectures is becoming unbearable in terms of supplying and cooling costs [27, 28] (mostly in HPC) and chip reliability [29]. Moreover, contention on shared resources has negative effects on both performance and energy efficiency [30].

Managing such systems is indeed challenging: on the applications side, performance and Quality of Service (QoS) are paramount. This is not surprising, given that users usually pay for computation by either renting computational power or directly buying the hardware. System-side goals, conversely, are completely orthogonal: minimizing power consumption, maximizing energy efficiency, and counteracting aging/thermal issues.

The Control Groups framework is usually employed to monitor resource usage [31] or to execute applications in lightweight virtual-

ization environments [32]; however, they can also be used in an adaptive fashion: the allocation can be adapted to the runtime-variable demand of applications [33, 34], thus minimizing resource usage—hence power consumption, heat and resource contention—while letting applications comply with their QoS requirements. For instance, the CPU time allocated to an application can be shrunk as far as the QoS goal allows, thus reducing the system load and allowing multiple applications to peacefully co-run on the same set of cores [35].

When allocating CPU cores, a resource manager can perform three kinds of allocation:

CPUSET-CENTRIC
: The resource manager allocates an exclusive set of cores (also called cpuset) to each application;

BANDWIDTH-CENTRIC
: The resource manager allocates a suitable share of CPU time to each application—hence the term "CPU bandwidth", i.e., "CPU time used over a given period";

MIXED
: The resource manager allocates a cpuset to each application. In case of overlapping cpusets, it also allocates a suitable share of CPU time to each application.

When the resource manager allocates CPU resources in a cpuset-centric fashion, it isolates each application on an exclusive set of cores; hence, inter-application interference is limited to inherently shared resources such as the last level cache and memory. Conversely, in the case of bandwidth-centric allocation, resource contention happens at all the levels from memory hierarchy to cores pipeline. An application that uses less CPU bandwidth than expected may be unable to comply with its QoS requirements, while an application that uses too much bandwidth will steal CPU time from its co-runners, thus inducing performance degradation on the entire workload. It follows, then, that it is paramount to choose a suitable bandwidth for each task and to make sure that it is accurately enforced. We specifically tackle the aforementioned problem by studying how the configuration of the cgroups *cpu* controller affects the accuracy of CPU bandwidth enforcement.

Towards a fine-grained CPU bandwidth enforcement

The term "CPU bandwidth" refers to the amount of CPU time that is used by an application over a given time frame. Therefore, in order to enforce an accurate bandwidth on applications, the CPU time they effectively use must be monitored at regular intervals during their

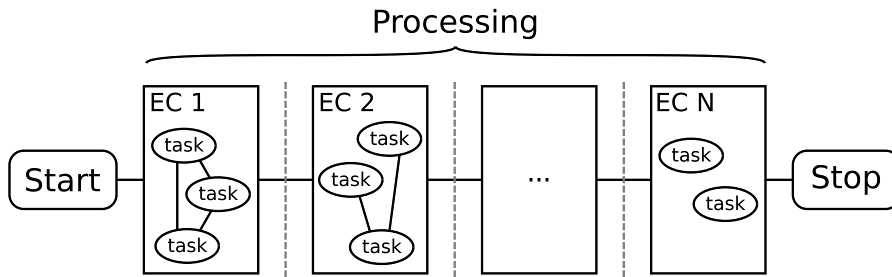


Figure 6: Dividing an application into *Execution Cycles* to enable a fine-grained CPU bandwidth monitoring. The division (dashed lines) can be performed time-wise, using barriers or with other synchronization mechanisms.

execution. To do that, we split applications into sequential processing windows that can be monitored individually. We call these windows *Execution Cycles* (*ECs*). Enforcing an accurate CPU bandwidth means making sure that each *EC* of an application uses no more and no less than the allocated CPU bandwidth, which is very useful both to minimize resource contention and to ensure the compliance with power and thermal budgets throughout the entire execution of a workload.

The division of applications into *ECs* follows the schema presented in Figure 6. It can be performed periodically, on an event basis or using barriers or other synchronization mechanisms (e.g. heartbeat). For instance, a video encoding application may divide its processing into 25-frame *ECs*, so that CPU bandwidth can be evaluated for each group of 25 frames. Please note that the schema presented above can be applied to a huge variety of applications, e.g. image processing (*EC* = "a group of frames"), web servers (*EC* = "a group of requests"), number-crunching (*EC* = "a given amount of calculations"), or streaming applications.

It is worth noticing that bandwidth-centric allocation is typical of embedded systems, which feature a limited amount of cores and therefore are often unable to reserve an isolated set of cores for each application. Nonetheless, in order to make the system comply with thermal and power constraints, bandwidth-centric allocation (in its mixed variant) can be also used in HPC scenarios [36]. Thus, our study also applies to big systems, where applications that use less bandwidth than expected will suffer quality losses, while applications that use too much bandwidth are likely to make the hardware violate some thermal or power budget.

In the second part of the disseration, we will prove that bandwidth allocation is indeed very effective in HPC.

Bandwdith allocation using the cgroups cpu controller

The *cpu* controller limits the CPU time that the Linux scheduler may allocate to tasks. This is not about isolating tasks on a subset of the available CPU cores (something that pertains to the *cpuset* controller):

it is about setting a constraint on the maximum amount of time that tasks can spend running on the allocated cores. Please note that there is a subtle difference between allocating and limiting CPU bandwidth. The *cpu* controller does not allocate CPU time to applications: its goal is to make sure that each application does not use more than the "target" CPU bandwidth in a given time frame.

The bandwidth allocated to an application is specified using a *quota* and a *period*, both expressed in microseconds. During each *period*, the application is allowed to run up to "*quota*" microseconds. We define the ratio between *period* and *quota* as "equivalent cores usage". For instance, let us consider an application that runs on four cores but can use them no more than half of the time: in this case, given that in Linux systems the default period is 100ms, the total CPU time provided by four cores is 400ms per period (each core provides 100ms worth of execution time). An application that uses those cores only half of the time has a quota of 200ms, which means $\frac{\mathrm{quota}}{\mathrm{period}} = \frac{200\mathrm{ms}}{100\mathrm{ms}} =$ 2 equivalent cores.

To keep numbers small, all our examples will be expressed in milliseconds.

In the example presented above, using a 50ms period and a 100ms quota would have resulted in the same usage. This is due to the fact that a single constraint—that is, CPU bandwidth—is set by using two knobs: period and quota. The motivation behind this choice is that, on equal *equivalent cores usage*, long periods result in increased burst capacity, i.e. the application can run more without being interrupted; conversely, short periods ensure a consistent latency response, i.e., the application will not suffer from long interruptions.

Figure 7 details how the *cpu* controller enforces bandwidth. Time is divided into periods, and the accounting of CPU time is re-set at the end of each period. A managed application is free to use the allocated cores as long as its cumulative running time within a period does not reach the target quota. Then, the *cpu* controller throttles the application and prevents it from running until the next period.

The application from Figure 7 is CPU bound and features a constant CPU demand (i.e., the light and dark gray slopes overlap); however, such applications are not common: usually, CPU usage is at least control and data-dependent. Moreover, the enforcement is poorly predictable due to the decoupling between bandwidth demand and accounting (i.e. *Execution Cycles* and enforcement periods are not synchronized).

Concerning the first point, Figure 8 shows the effects of different periods on applications whose CPU usage is not constant (8a). The *cpu* controller does not take into account the fact that applications could under-use the allocated CPU time during certain periods (which is the difference between allocating and limiting CPU bandwidth). Hence, when using short periods, the resulting usage could be lower than expected (8b). Large periods may be employed to avoid

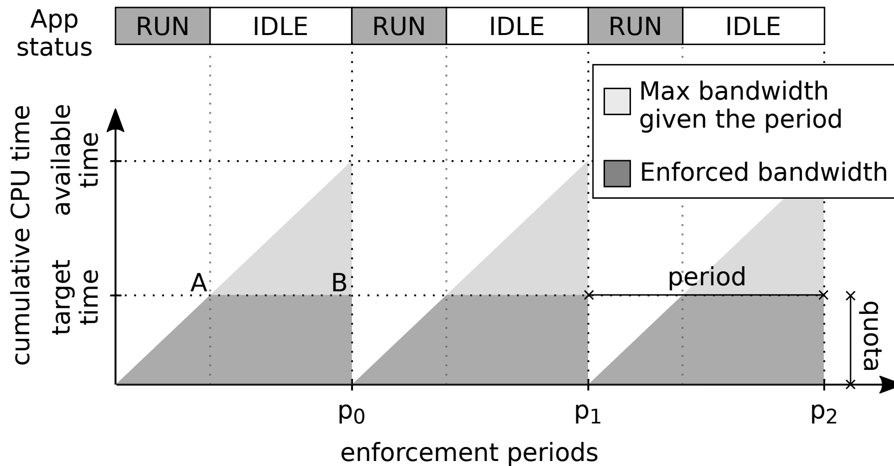


Figure 7: *Cpu* controller bandwidth enforcement. During each period, the application running time is cumulated until it reaches the target value (A); then, the application is throttled and it is not allowed to run until the next period, when the accounting is re-set (B).

under-usage; however, they could lead to several over-usage occurrences inside the period (8c).

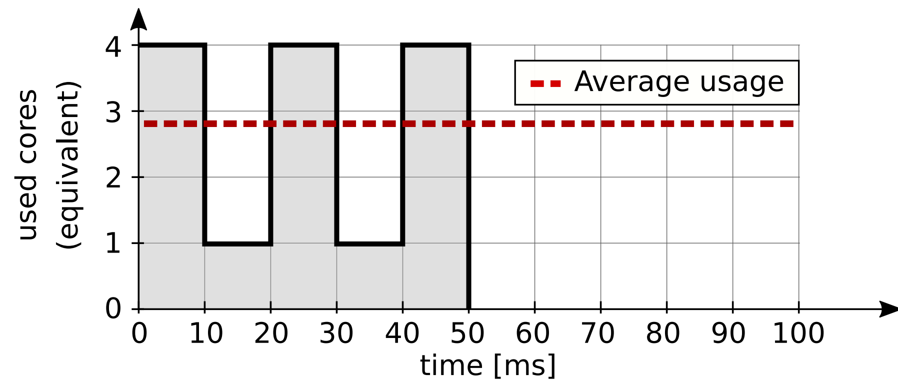
Unfortunately, larger periods do not always lead to an accurate enforcement. Figure 9 shows how the decoupling between bandwidth demand and accounting affects the enforcement: especially for applications whose CPU usage is constant (9a), the resulting usage could be quite higher than expected (9b). For instance, bursts that happen across consecutive periods are able to use both quotas, resulting in high equivalent cores usages.

Simulating CPU bandwidth enforcement

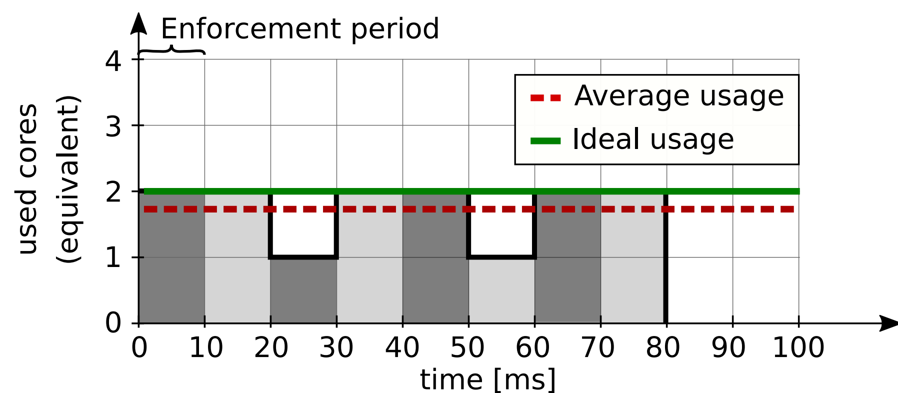
Regardless of how an application is divided into *Execution Cycles*, synchronizing CPU bandwidth accounting and *ECs* is not trivial, inasmuch as it would require to modify either (or both) kernel or applications source code and it would also introduce synchronization overheads. Therefore, we chose to investigate whether it is possible to achieve an accurate CPU bandwidth allocation by selecting a suitable enforcement period for each application. To do that, we developed a *cpu* controller simulator that benefits from the following simplifications:

- the applications execution is split in *ECs*, and the accuracy of bandwidth allocation is computed at *EC* level;
- the CPU usage of the *ECs* is discretized into arbitrarily small constant-usage slots. This does not decrease the accuracy of our simulator, given that the accounting performed by cgroups is also discretized;

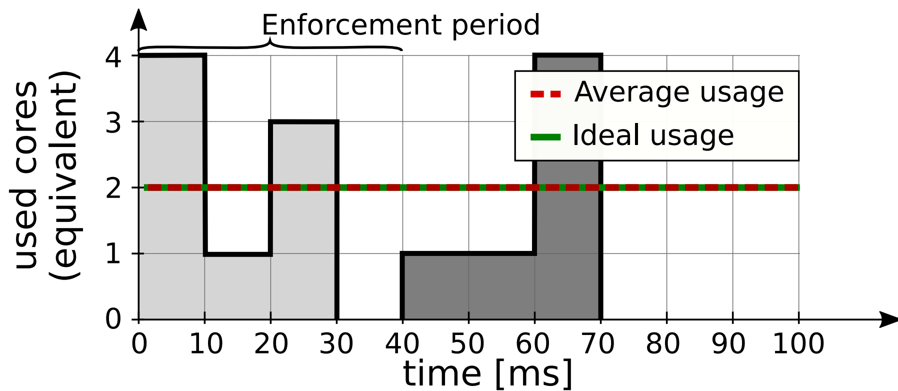
Slots size can be set via procfs: /proc/sys/kernel/sched_cfs_ bandwidth_slice_us.



(a) No quota enforcement.

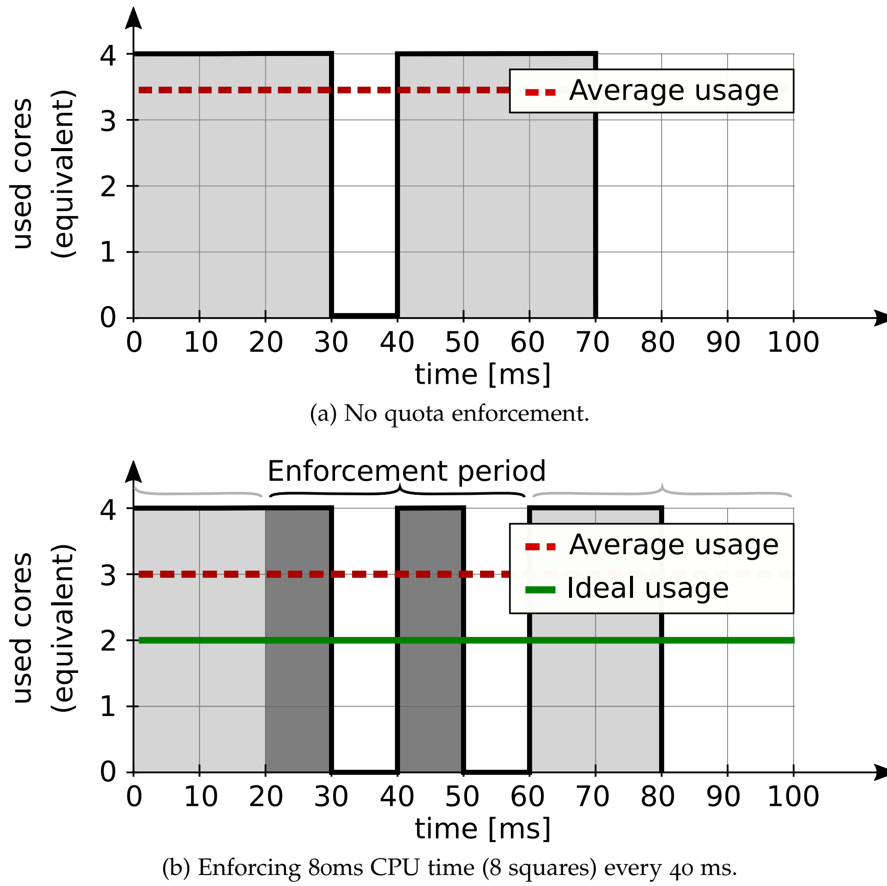


(b) Enforcing 20ms CPU time (2 squares) every 10 ms.



(c) Enforcing 80ms CPU time (8 squares) every 40 ms.

Figure 8: Enforcing an average usage of two equivalent cores by using different periods. The CPU demand of the application is not constant (8a); hence, short periods lead to an average resource under-usage (8b). Large periods, conversely, lead to an average optimal usage but also to several instantaneous over-usages, i.e. first, third and last 10-ms slot (8c). Shaded areas indicate consecutive enforcement periods.



(a) No quota enforcement.

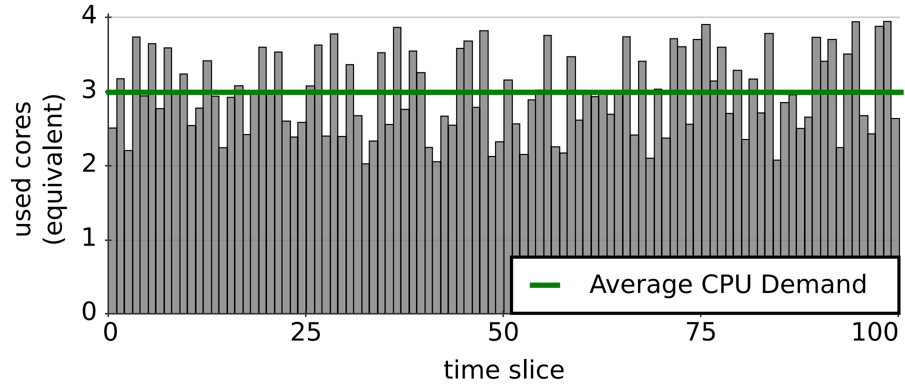
(b) Enforcing 80ms CPU time (8 squares) every 40 ms.

Figure 9: Enforcing an average usage of two equivalent cores. The application CPU demand is composed by bursts (9a). In case of large periods, lack of synchronization between enforcement and processing leads to resource over-usage (9b). Shaded areas indicate consecutive enforcement periods.

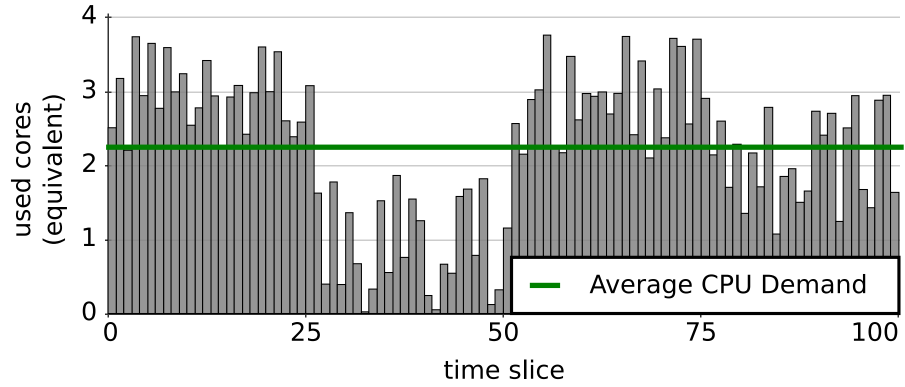
- *ECs* of the same application feature the same CPU demand and execution time. This allows us to analyze the enforcement mechanism with disregard of any data variability-induced effect;
- nothing happens between *ECs*: management overheads are negligible with respect to the *EC* execution time. This is typical of resource management scenarios.

Simulation setup

We set up the simulator according to the following scenario: a multi-threaded application that is isolated on four cores. For the sake of simplicity, let us assume that the application uses four threads, which means that, even when no constraints are applied, its average equivalent cores usage is always less than four by construction (each thread cannot use more than one core at a time). We refer to the average unconstrained usage as γ. Please note that we assume the *ECs* to be



(a) Homogeneous CPU usage profile.



(b) Heterogeneous CPU usage profile

Figure 10: Execution Cycle usage profiles used during the simulation. The first one (10a) features an homogeneous demand, i.e., the demand is often close to the average one. The second one (10b), conversely, is heterogeneous: the CPU demand is often far from the average.

equal in terms of CPU usage; hence, γ is also the average usage of each *EC*. We run three simulations, enforcing a maximum CPU usage equal to γ, $\frac{3}{4}\gamma$ and $\frac{1}{2}\gamma$, respectively. The output of each iteration is the average CPU usage of the application until the end of each *EC* up to the tenth.

We performed the simulation using two randomly generated CPU usage traces, which are shown in Figure 10. The first one features a homogeneous CPU demand, whose average value is 3.0 equivalent cores (10a). The second, conversely, features a heterogeneous CPU demand with an average value of 2.2 equivalent cores (10b).

Simulation results

Results for the homogeneous and heterogeneous scenarios are shown in Figures 11 and 12, respectively. Each Figure shows the percent allocation error for the three scenarios, i.e., the enforcement of γ, $\frac{3}{4}\gamma$ and $\frac{1}{2}\gamma$ CPU bandwidth. Negative errors indicate resource underusage, which means that the application uses less bandwidth than ex-

pected; on the contrary, positive errors indicate resource over-usage. The length of enforcing periods is normalized to the *EC* execution time.

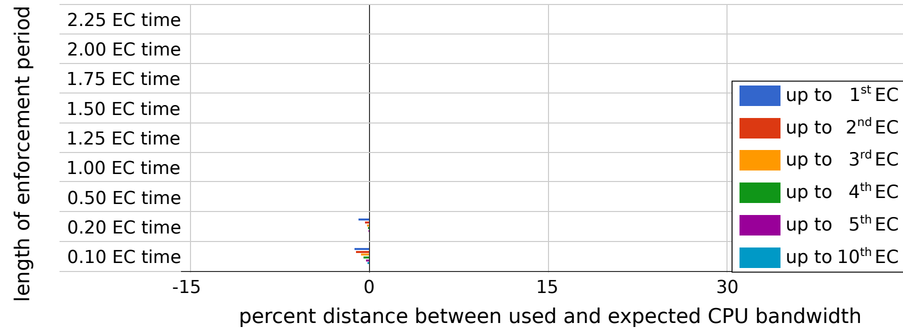
Some considerations apply to both scenarios: first of all, resource over-usage is not possible when enforcing a CPU bandwidth equal to γ. This is always true, because the average CPU usage of an *EC* does not exceed γ by construction. Second, please note that errors are represented as "up to the Nth *EC*", which means "average value of all the errors up to the N^{th}one". Hence, after a few cycles, the union of under and over-usage errors caused by the decoupling between periods and *ECs* makes the errors converge to low values even if the error of each *EC* does not; on the contrary, since they are compulsory, under-usage errors do not converge.

In case of homogeneous bandwidth demand (Figure 11 on page 26), the under-usage effect caused by short enforcing periods is present only when the controller enforces a CPU bandwidth equal to γ (Figure 11a). This is not surprising: as shown in the usage trace (Figure 10a), the bandwidth demand of the application is indeed always greater than 2 equivalent cores, i.e., $\frac{3}{4}\gamma$: when the enforced CPU bandwidth is greater than that, resource under-usage is not possible by construction. Conversely, as already anticipated by the example from Figure 9 on page 23, large enforcement periods cause resource over-usage: while the average CPU usage of the application does not exceed γ by construction (11a), in case of lower bandwidth (11b, 11c) the allocation gets more inaccurate as the period increases. Overall, the shorter periods with respect to the *EC* execution time, the more accurate the bandwidth enforcement.

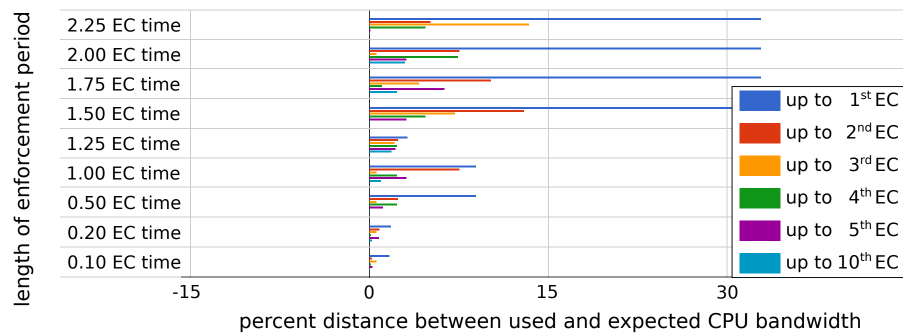
In case of heterogeneous bandwidth demand (Figure 12 on page 27), conversely, short enforcement periods lead to compulsory resource under-usage. We already addressed this problem in the example from Figure 8 on page 22: the *cpu* controller resets the usage statistics after each period; hence, the CPU bandwidth that the application is not able to use during a period is permanently lost. Obviously, this effect gets minimized in the test from Figure 12c because the instantaneous CPU usage of the application (Figure 10b) is almost always greater than $\frac{1}{2}\gamma$ (1.1 equivalent cores). Even in the heterogeneous scenario, large enforcement periods lead to over-usage. As shown in Figure 12a, this is not a problem when enforcing a CPU bandwidth equal to γ. In case of smaller allocations (12b, 12c), instead, the trend is similar to the one observed in the homogeneous scenario. Overall, periods close to the *ECs* execution time lead to more accurate results.

Discussion

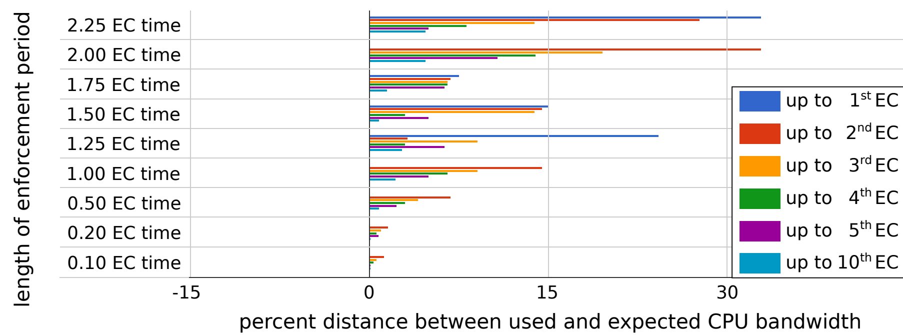
Simulation results confirm that, to maximize the enforcement accuracy of the *cpu* controller, each application should be subject to a different enforcement period.



(a) Enforced bandwidth = γ.

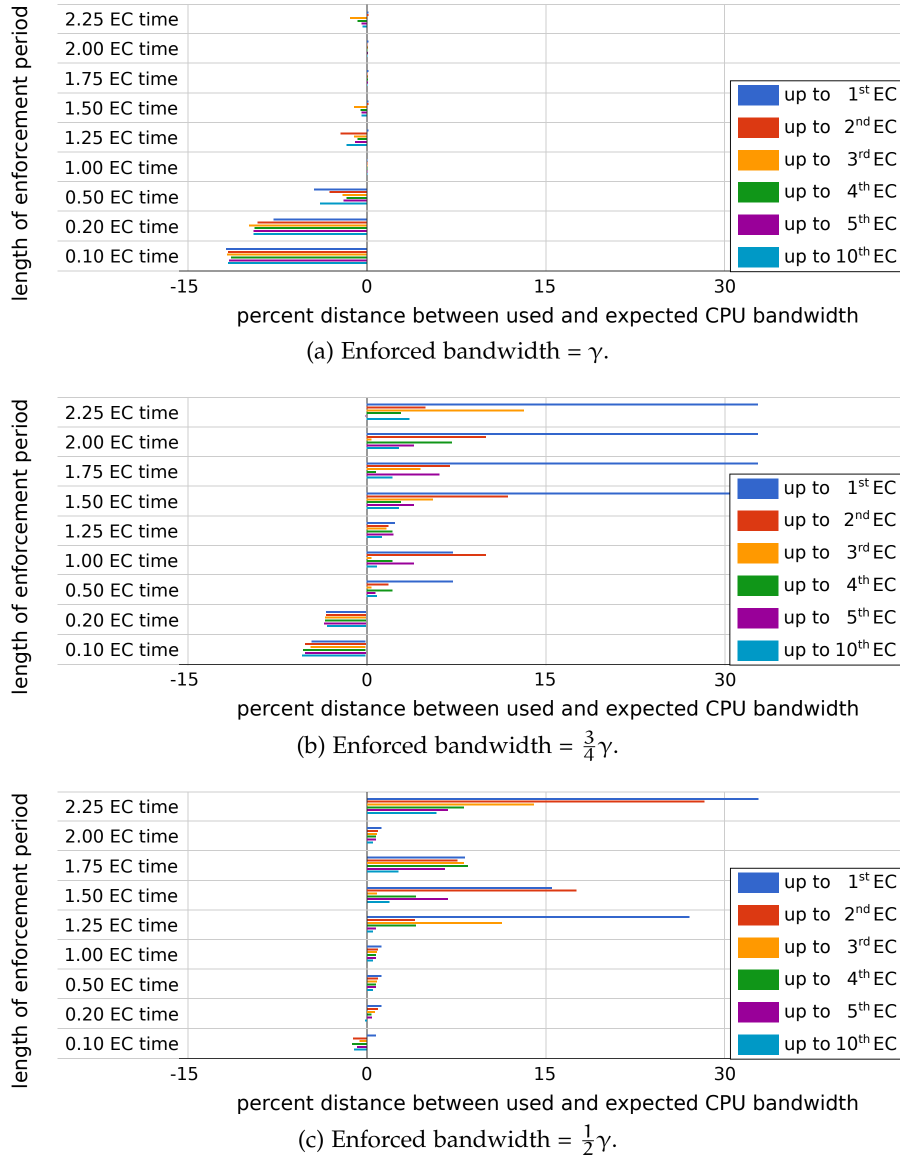


(b) Enforced bandwidth = $\frac{3}{4}\gamma$.



(c) Enforced bandwidth = $\frac{1}{2}\gamma$.

Figure 11: Homogeneous scenario: allocation error (percent) when limiting CPU usage using different enforcement periods. Period length is normalized to EC execution time. γ is the average CPU usage of the application when not using the *cpu* controller.



(a) Enforced bandwidth = γ.

(b) Enforced bandwidth = $\frac{3}{4}\gamma$.

(c) Enforced bandwidth = $\frac{1}{2}\gamma$.

Figure 12: Heterogeneous scenario: allocation error (percent) when limiting CPU usage using different enforcement periods. Period length is normalized to EC execution time. γ is the average CPU usage of the application when not using the *cpu* controller.

Periods that are larger than the average *Execution Cycle* lead to CPU usages that are typically bigger than the expected ones. In case of coarse-grained allocations (i.e. when having accurate CPU bandwidth enforcement in the long term is enough), this is not necessarily a problem because the average error tends to decrease over time. However, it is worth noticing that, given the runtime-variable delay between bandwidth demand and accounting, CPU bandwidth enforcement is accurate only on average, while the enforcement performed in each single period can be quite inaccurate. An application that over-uses the available resources is potentially stealing bandwidth from its co-runners and, even in case of isolation, it is causing more contention than expected on shared resources.

On the other hand, periods that are shorter than the average *Execution Cycle* often lead to accurate allocations. However, the *cpu* controller resets the accounting statistics after each enforcement period (and rightly so, because this minimizes overheads and ensures fairness in multi-application scenarios); hence, applications whose CPU demand is often lower than the enforced one will not be able to completely exploit the allocated bandwidth even if, by using larger periods, it would indeed be possible. This phenomena could trick the resource manager into thinking that an application is not able to exploit the allocated resources. Differently from over-usage, whose average effect gets smaller over time, under-usage is compulsory; hence, it would lead to inaccurate enforcements also in case of coarse-grained allocations.

Validation

According to the simulation results, the effect of CPU bandwidth enforcement on the CPU usage of applications is greatly affected by the enforcement period length. In particular, the optimal period must be smaller than the *EC* execution time, but not short enough to cause bandwidth under-usage. In this subsection, we validate the results using real applications.

We reproduced the simulation scenario: a multi-threaded application that runs in isolation on four cores and uses four threads. This time, the resource manager caps the allocated CPU bandwidth to three equivalent cores. That is, the new allocation consists in four cores, but this time they can be used at most 75% of the time.

The applications used during the tests are described in Table 1.

We selected a subset of the PARSEC benchmark suite [37], focusing on applications that represent different domains. We carried out the tests without characterizing the CPU demand of applications because:

- CPU demand can be data-dependent, which, in turn, makes the ideal enforcement period data-dependent;

Table 1: Applications used during the tests. We divided the processing part of each application into Execution Cycles, for which we list description and average execution time when using four threads.

Name	Description	EC Division	Average EC duration [ms]
Blackscholes	Option pricing with Black-Scholes PDE	Each thread computes a single option value	77
Bodytrack	Body tracking of a person using multiple cameras	Process a single frame from each camera	96
Ferret	Content-based similarity search server	A single step in the parallel search pipeline	120
Fluidanimate	Animate fluid dynamics with SPH method	Produce a frame	118
Streamcluster	Online clustering of an input stream	Find a center in a fixed-size chunk of points	104
Swaptions	Prices a portfolio of swaptions using a MC simulation	Price one swaption per thread.	180
x264	H.264/AVC (Advanced Video Coding) video encoder	Process a burst of 5 frames	135

- according to the simulation, heterogeneous CPU demand can be simply inferred from the presence of bandwidth under-usage in case of short enforcement periods;
- addressing uncharacterized workloads widens the applicability of this study.

Please note that EC length is computed during runtime by the resource manager.

We performed the tests on a Linux-based system that features an Intel i7-6700K CPU (4 Simultaneous Multi-Threading cores @ 4.00GHz, for a total of 8 hardware threads). The operating system is Arch Linux kernel 4.9.11.

Figure 13 shows our hardware setup: we used cgroups to divide the available hardware threads into two subsets of four threads each. The first subset hosted the operating system along with all the tasks not related to the tests, while the second one hosted the tests. We chose to operate this partition in order to run the managed applications on four hardware threads while allowing each thread to exploit its own private cache. This way, we minimized the effects of cache contention on the applications execution.

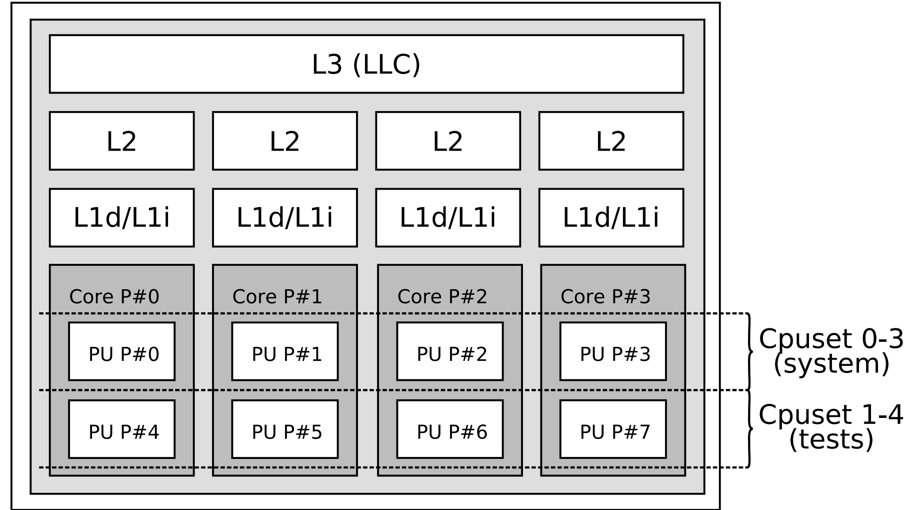
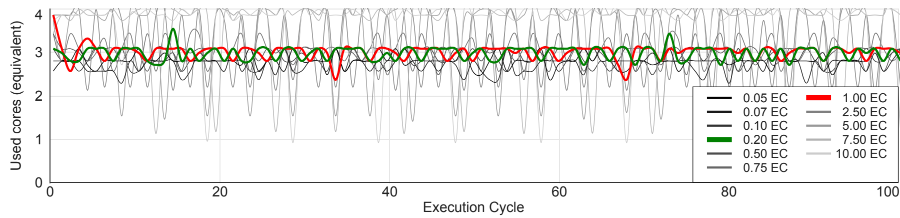


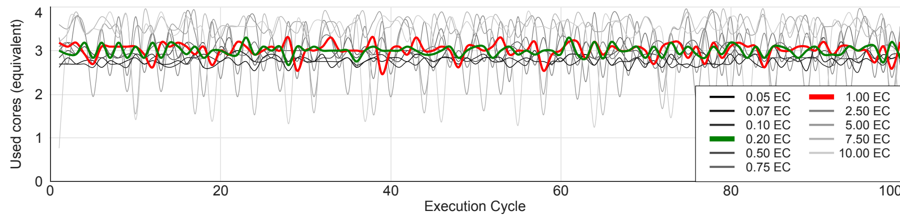
Figure 13: Hardware setup. The processor is partitioned in two cgroups cpusets. The former (cpuset $0-3$) is used to host the operating system, while the latter (cpuset $4-7$) is used to host the tests in isolation.

Figures 14 and 15 respectively report the CPU usage of each single *EC* and the average CPU usage of the whole applications. In both figures, we use increasingly dark gray lines to indicate increasingly small enforcement periods. Moreover, we use colored lines to indicate periods that are equal to $\frac{1}{5}\mathtt{EC}_{\mathtt{length}}$ (dark green) and $\mathtt{EC}_{\mathtt{length}}$ (light red). According to the simulation results, those periods are indeed most suited in case of homogeneous and heterogeneous CPU bandwidth demand, respectively.

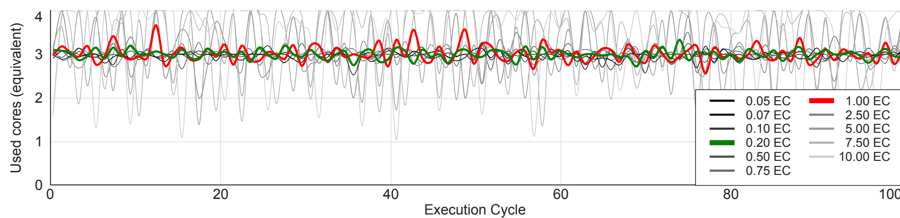
The effect of large enforcement periods on the CPU bandwidth of each *EC* is clearly shown in Figure 14 on page 31. In those cases (light gray lines), each period is large enough to fit several *Execution Cycles*. This has negative effects on bandwidth usage: during a single period, the application is allowed to execute multiple *ECs* using all the available resources, followed by a single *EC* whose CPU usage is extremely limited because there is not any more bandwidth to be used in that period. This causes the CPU bandwidth to wildly oscillate around the expected one, alternating under and over-usage. In case of power/thermal-related constraints or applications co-running on a set of cores, the positive effects of bandwidth allocation are totally lost: applications never use the allocated bandwidth because they always use either all or almost none of the available CPU time. The effect of short enforcement periods (dark gray lines) is instead clearly shown in Figures 14a, 14b, 14d and 14f: in those cases, the observed CPU bandwidth seldom surpasses the enforced one (i.e., 3 equivalent cores); however, given that CPU time accounting does not span over multiple periods, bandwidth usage is often lower than that, thus leading to under-usage.



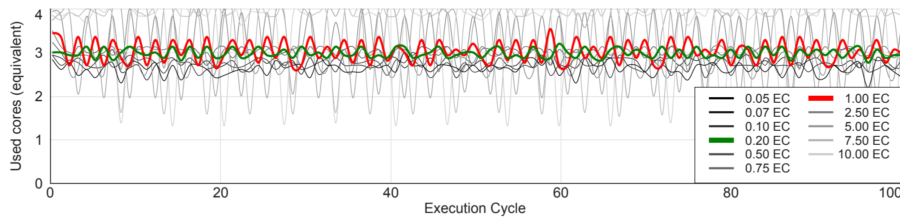
(a) Blackscholes.



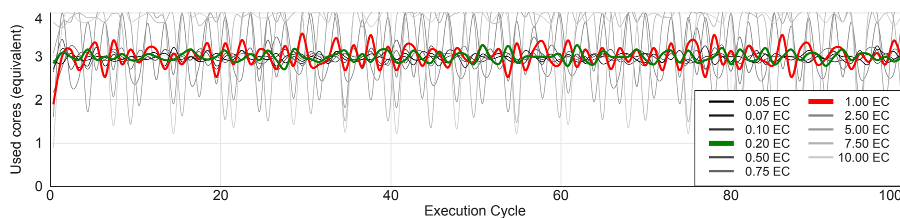
(b) Bodytrack.



(c) Ferret.

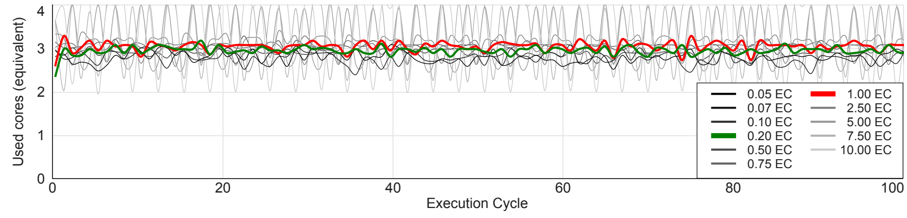


(d) Fluidanimate.

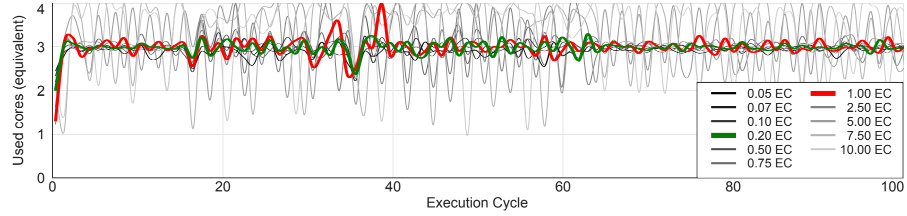


(e) Streamcluster.

Figure 14: Experimental results. For each application and for each period, we list the equivalent core usage of all the *Execution Cycles* up to the 100$^{\text{th}}$ one. Accurate allocations are those that are closer to 3 equivalent cores. Thick lines indicate the ideal period bounds as found using simulation: a fifth of *EC* length (light green) and *EC* length (dark red). Increasingly dark lines indicate increasingly small enforcement periods.



(f) (cont.) Swaptions.



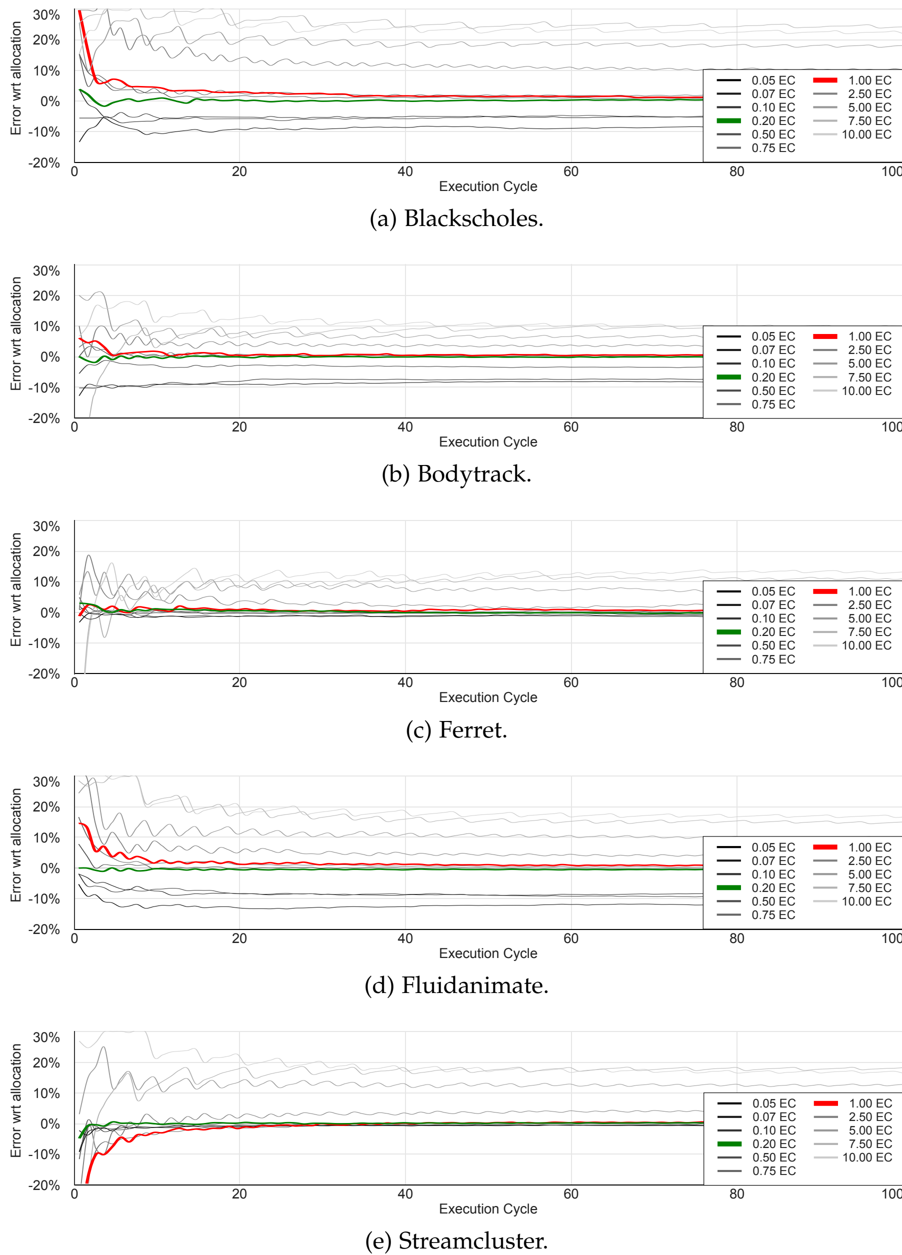
(g) (cont.) x264.

Figure 14 (cont.): Experimental results.

Overall, Figure 14 confirms the simulation results: enforcement periods whose length is equal to EC_{length} (red lines) lead to an accurate enforcement, which, however, may be subject to oscillations due to the runtime-variable delay between *ECs* and enforcement periods. Moreover, periods whose length is equal to $\frac{1}{5}EC_{length}$ (green lines) lead to quite stable results, and, in some cases, oscillations are also mitigated (see Figures 14d and 14e). Finally, periods that are too short or too large with respect to the above mentioned boundaries respectively lead to under and over-usage.

Figure 15 on page 33 offers a different view on the aforementioned results: the average bandwidth usage of the applications over a hundred *Execution Cycles*, that is, in a time frame of several seconds. Unfortunately, even in this case, all the enforcement periods that are larger than EC_{length} (light gray lines) lead to average CPU bandwidths that are higher than the enforced ones (up to 25% error). This means that the observed CPU bandwidth is over-sized not only at *EC* granularity, where the measured bandwidth wildly oscillates under and over the enforced one, but also on average. Even in case of coarse-grained allocations, applications may use more bandwidth than expected despite the cgroup-based enforcement. Similarly, periods that are shorter than $\frac{1}{5}EC_{length}$ (dark gray lines) often lead to bandwidth under-usage, as clearly shown in Figures 15a, 15b, 15d and 15f (up to 10% error).

Overall, even in this case, simulation results are confirmed: enforcement periods whose length is equal to EC_{length} (red lines) lead to an accurate average enforcement (less than to 1% error). At the same time, these periods (see the instantaneous views from Figures 14 and 14) hide an oscillatory behavior: the enforcement is accurate only on average, but the real usage oscillates between slight under and over-usage. On the other hand, periods whose length is equal to



(a) Blackscholes.

(b) Bodytrack.

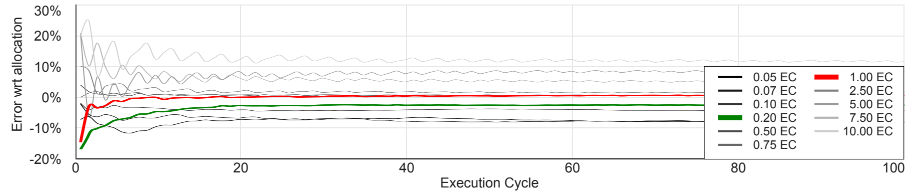
(c) Ferret.

(d) Fluidanimate.

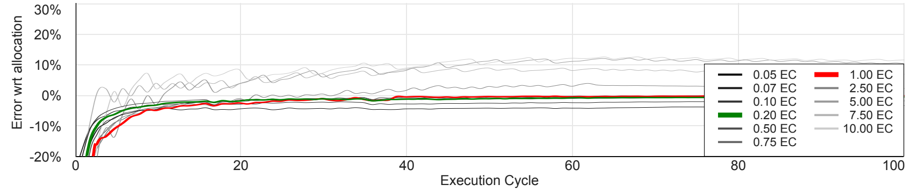
(e) Streamcluster.

Figure 15: Experimental results. For each application and for each period, we list the average allocation error measured at the end of the first 100 *Execution Cycles*. The error is defined as the percent distance between the allocated CPU bandwidth and the measured one. Thick lines indicate the ideal period bounds as found using simulation: a fifth of *EC* length (green) and *EC* length (red) Increasingly dark lines indicate increasingly small enforcement periods.

$\frac{1}{5}EC_{length}$ (green lines) lead to results that are accurate in the average case (less than 1% error) but also in the instantaneous one. Indeed, $\frac{1}{5}EC_{length}$ is not a magic number: as also shown by the simulation results (see Figure 12 on page 27), the smallest enforcement period



(f) (cont.) Swaptions.



(g) (cont.) x264.

Figure 15 (cont.): Experimental results.

that provides a good accuracy while avoiding over-usage is application dependent. In the case of *swaptions*, $\frac{1}{5}\mathsf{EC}_{\mathtt{length}}$ is too short and makes the application suffer from bandwidth under-usage (Figure 15f).

Conclusions

Whereas Control Groups are mostly used to perform isolation, i.e., they are used to run applications on a subset of the available CPU cores (*cpuset*), they also allow CPU usage to be throttled over time inside the allocated cpu set, hence enabling both application-specific and system-wide optimizations. While *cpuset* allocation is accurate by definition (because a thread either can or cannot be scheduled on a given core), CPU time enforcement is performed through CPU bandwidth control, which does not guarantee that applications will really exploit the allocated resources.

So, why is accuracy so important? The answer is strightforward: once a resource manager allocates resources, it must be sure that its decision will be accurately actuated. If not, creating refined scheduling policies would make no sense at all.

In this section, we studied how to achieve an accurate CPU bandwidth enforcement. First of all, we analyzed how the cgroups *cpu* controller limits bandwidth. We discovered that, in certain scenarios, the CPU bandwidth that is effectively exploited by the applications could potentially be not only lower than the expected one, but also notably higher.

To accurately enforce bandwidth usage on applications, we proposed to split each application into computation units, so that bandwidth usage could be individually monitored for each unit. We called these units *Execution Cycles*. We then developed a *cpu* controller simulator to understand the relationship among applications CPU demand, enforcement period length and enforcement accuracy, and we discovered that selecting the enforcement period while taking into

account the *Execution Cycles* length leads to a very accurate enforcement.

Finally, we performed a validation using real applications. We confirmed that, depending on the enforcement period, cgroups-based CPU bandwidth control may be inaccurate not only on *Execution Cycle* granularity, but also on average, i.e., throughout the entire application execution. This issue can be tackled quite effectively by employing a period that is shorter than the *Execution Cycle* length, but not so short that it leads to bandwidth under-usage.

Our approach, which does not require applications to be profiled and can therefore be easily adopted by any cgroups-based resource manager, led to a very accurate bandwidth enforcement ($< 1\%$ error) with respect to the period-agnostic scenarios (more than 25% error in the worst case).

3 RESOURCE ALLOCATION: SYSTEM-WIDE VS APPLICATION-SPECIFIC

In this chapter, we focus on the most common high-end computing systems. That is, we address devices that feature a multi-core heterogeneous processor and, optionally, an accelerator. Multi-core systems feature computational resources that are limited in number—usually a few cores plus the memory hierarchy—and must hence be shared among multiple applications. Conversely to many-core systems, where the problem of CPU resource allocation can be often reduced to "how many processing elements will be allocated to each task", multi-core systems must often face the problem of co-running multiple tasks on the same processing elements. Indeed, in this case, allocating resources also means choosing "how much time will a task spend on the available processing elements with respect ot the other tasks".

Isolating an application in a subset of the avaiable computing cores also means allowing it to exclusively access the associated portion of the cache hierachy. Unfortunately, it is not possible in small systems.

First of all, we study how to take into account the characteristics of applications to optimize resource allocation. Second, we will show that the resource manager, which has a global view of the system, can rely on application-specific auto-tuners, which lack the aforementioned global view but got a specific knowledge of the application goals, in order to make applications exploit the allocated resources in an optimal way.

TAILORING ALLOCATION TO APPLICATIONS

Concurrently running multiple threads on a set of shared resources leads to resource contention, which, in turn, negatively effects both performance and energy efficiency [30, 38, 39, 40]. A common way to tackle this issue is to employ *applications-aware co-schedulers*.

Viewing applications as black boxes does reduce the management complexity; however, it also leads to sub-optimal scheduling choices.

Standard schedulers operate choices that are mostly based on *fairness*, i.e. they allocate a fair amount of resources or computing time to each thread; and *load balancing*, i.e. they evenly distribute threads among the available resources [41]. Conversely from standard schedulers, co-schedulers mitigate resource contention by identifying the threads that are best suited to be concurrently executed on the same set of resources.

From brute force prediction to application characterization

Identifying the optimal co-scheduling (i.e., the allocation of threads that minimizes resource contention) is a feasible task: it can be done

by evaluating the performance degradation resulting by all the possible combinations of threads on all the available subsets of cores. This process is often referred to as *brute force prediction*, and, unfortunately but not unexpectedly, theoretical analyses proved it to be *NP-complete* [42]. Obviously, as the number of applications to be run gets higher, this kind of approach is not viable at all.

An NP-complete problem is both NP (nondeterministic polynomial) and NP-hard (cannot be reduced to NP in polynomial time).

Near-optimal predictions can be instead achieved with a lower complexity. *Tian et al.* [43] propose two approaches—*A*-cluster* and *local-matching*—that approximate the optimal schedules with good scalability. The *A*-cluster* algorithm treats the optimal co-scheduling problem as a *tree-search problem*, applying the well-known *A*-search* algorithm to efficiently find optimal co-schedule combinations. Clustering techniques are exploited to reduce the frequency of re-scheduling by initiating a co-schedule only when an entire cluster of applications has terminated, which avoids the generation of sub-schedules that are similar to one another, thus shrinking the number of possible mappings at the cost of accuracy. The time complexity is $O(N)$, where N is the number of remaining jobs. The clustering is based on execution time, i.e., jobs with a similar execution time are clustered. More efficiency is achieved with the *local matching* algorithm, where the problem of co-scheduling is solved with a graph-based approach that allows to find local optimum choices. Such approach does not involve clustering, thus incurring in a complexity of $O(N^4)$. Moreover, on systems with more than two cores per chip, the approach is once again *NP-complete*.

The term tree traversal (or tree search) refers to the process of analysing each node in a tree in a systematic way, without visiting a node more than once.

A following study by *Jiang et al.* [44] proposes an Integer Programming (IP) co-scheduling algorithm that is based on the observation that co-scheduling N jobs into M chips is equivalent to partitioning N jobs into $M = N/U$ sets, where each set contains U jobs and the total co-run degradation of all jobs has to be minimized. Once again, the complexity of the algorithm is not polynomial on systems with more than two cores per chip; nevertheless, its lower bound can be computed through the *Linear Programming* (LP) version of the model. The same study presents two of heuristics-based algorithms. The first one, which is called *hierarchical matching algorithm*, generalizes the IP algorithm by partitioning jobs in a hierarchical way with a complexity of $O(N^4)$. The second algorithm is called *greedy algorithm*; it schedules jobs giving priority to the most sensible to cache contention, with a complexity of $O(N\binom{N}{U})$.

In an IP problem, the complexity is reduced by restricting some variables to be integers.

In an LP problem, the objective function and the non-integer constraints are linear.

Given the increasing number of cores and concurrent applications, in addition to the growing number of threads per application, *brute force prediction* proves to be too complex to be effectively exploited. Moreover, executing a workload on a new architecture or inserting a new application into a workload would be costly due to the high number of tests to be performed.

In recent years, several works proposed low-complexity algorithms that allow co-scheduling decisions to be computed in a more reasonable time. Each application is analyzed singularly to extract metrics that represent its resource usage; the resulting information is then exploited by co-schedule algorithms to compute co-scheduling choices that are *sub-optimal* yet effective. The analysis process is often referred to as *application characterization* and has a complexity of $O(N)$, thus providing a trade-off between prediction accuracy and characterization effort [45, 46, 47]. *Application characterization* can be performed either through runtime monitoring [48, 49, 50, 51] or, if the applications are known a priori, through an off-line analysis [52, 46, 53].

Selecting metrics to characterize resource contention

The term *resource contention* is rather vague. Co-schedulers make decisions according to the *resource usage* of the running threads, but which resources are to be taken into account varies from architecture to architecture. Given the wide variety of applications and architectures, this is not surprising: previous studies have demonstrated that substantial speed-ups can be achieved by characterizing threads basing on *L1* usage [54, 55]; *last level cache* (*LLC*) usage (where the LLC is either an *L2* [56, 57, 58] or *L3* [59, 60] cache); available bandwidth at each cache level [61]; data locality [62]; off-chip bandwidth [63, 64]; or micro-architectural events such as front side bus stalls [65, 38, 66], floating point units usage [52, 67], and stall cycles [68].

Blagodurov et al. [45] show that contention on shared *floating point units* leads to substantial performance degradations and provide numerical results for floating point unit contention of SPEC CPU2006 benchmarks running on a UltraSPARC T1 microprocessor. The substantial performance degradation induced by co-scheduling *floating point intensive* applications is imputable to the fact that T1 processors feature eight cores which share a single floating point unit. While this is clearly a limitation that is not the norm in modern processors, such scenario is nonetheless emblematic: the problem of resource contention is not architecture independent, nor is its solution. Despite the great variety of metrics, it is clear that one the most severe and architecture-independent bottlenecks to be taken into account when co-scheduling applications is the memory.

There also approaches [69, 64] that treat the problem of contention in an indirect way: whereas all the other characterization-based approaches exploit resource usage (the *cause* of resource contention) as a metric to achieve effective co-scheduling decisions, in this case the chosen metric is the performance degradation suffered by co-running threads (the *effect* of resource contention). Such approaches are exclusively based on the performance degradation experienced by each thread when executed in a high resource contention environment.

Such metric is computed by co-running each thread with benchmarks that stress a number of chosen resources. The *sensitivity* of the thread to resource contention, i.e. the slowdown caused by co-running, can be computed by comparing the execution time of the thread when co-running with the benchmarks and when running alone.

ENERGY-EFFICIENT CO-SCHEDULING USING PERFORMANCE COUNTERS

The contents of this section are partially published in [67]. You may want to consult Appendix A before venturing forth.

In this section, we show how taking into account the characteristics of applications can improve the allocation choices of a resource manager. This work serves two purposes: first of all, we will give you an idea of the effort that is needed to suitably characterize applications on a target architecture; second, we will show that allocating resources is only half of the problem: applications that are aware of their own resource allocation and tune their parameters accordingly are indeed able to further optimize their resource usage. That is, applications should cooperate with the resource manager to enable an optimal resource management.

Our contribution is two-fold: first, we developed a Design Space Exploration (DSE) flow that is integrated with the resource manager and automatizes the characterization of applications. Second, we designed a resource mapping policy that exploits characterization information to select which processing elements will be allocated to each application.

The resource manager we used and extended in this work is the Barbeque Run-Time Resource Manager (BarbequeRTRM), which is described in Appendix A.

A performance-counters-aware BarbequeRTRM

The BarbequeRTRM bases resource allocations on a set of "golden configurations". That is, each managed application must be analyzed during an off-line characterization phase whose output is a set of pareto-optimal allocation choices. Each choice is called Application Working Mode (AWM) and is described by a set of resources—e.g., an AWM could require "two processing elements and a GPU"—and by an integer number that is used to explicitly order the AWMs from the least to the most performing. The list of AMWs of an application is contained in an XML file that is called *recipe*.

An allocation is pareto-optimal if there are not alternative allocations that improve one objective without worsen any of the others.

The main idea behind this work is to characterize memory usage and energy consumption of applications and to insert the obtained information into the AMWs description, so that the BarbequeRTRM allocation policy is able to take that information into account during the process of AWM mapping (i.e., when mapping the allocated resources on the hardware). The work consisted in three steps:

PERFORMANCE COUNTERS SELECTION
: The goal of this step is to find out which performance counters are most related with memory contention and energy consumption on the target architecture.

OFF-LINE APPLICATIONS CHARACTERIZATION
: This step consists in performing an automatized characterization of applications in order to create an enriched application recipe. The Design Space Exploration (DSE) engine computes which are the best configurations for the application. Moreover, it characterizes each of the selected allocations by monitoring both the obtained quality level and the values of performance counters.

RUNTIME RESOURCE ALLOCATION
: During runtime, the BarbequeRTRM selection policy is able to perform a contention and energy-aware resource mapping by exploiting the additional information that is contained in the enriched recipes. We extended the BarbequeRTRM allocation policy by implementing a resource mapping heuristic that tries to evenly spread energy consumption over the whole chip while minimizing memory contention.

Performance counters selection

In order to monitor the values of performance counters, we relied on the monitoring facilities offered by the BarbequeRTRM Runtime Library (see Subsection A.2.1). Regarding energy consumption, we instead used the Likwid tool [70].

When correlating performance counters to energy, the analysis process should be performed anew for each target architecture, new application and new dataset.

We performed the analysis by executing 10 applications from the Parsec benchmark suite [71] on a system that featured an Intel Core i7-2670QM quad-core SMT processor. Each core had independent L1 and L2 caches, while the L3 cache was shared between all the cores.

The relationship between performance counters and memory contention is well known: as already discussed in literature, performance degradation induced by memory contention is correlated to the number of last level cache misses generated by applications [72, 52]. Regarding energy consumption, instead, we compiled a list of suitable counters, and, starting from that list, we used a correlation test to extract the counters that are the best suited to characterize energy consumption. Figure 16 reports the correlation results for the most suitable performance counters: Last Level Cache misses, resource stalls, retired/issued micro-operations, retired instructions and floating point operations. In order to minimize the number of counters and to avoid biasing, we further shrunk the list of selected counters to *resource stalls*, *retired instructions* and *floating point operations*.

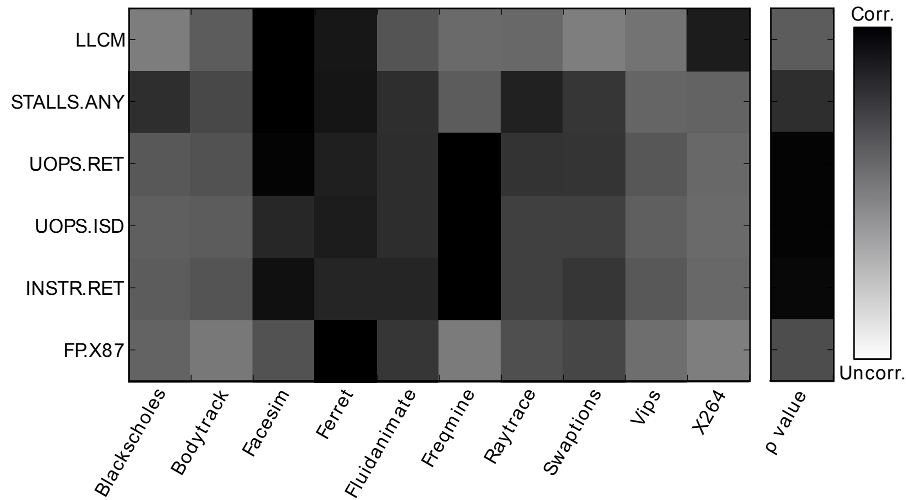


Figure 16: Spearman's rank correlation coefficient for each performance counter, along with correlation ρ for each specific application. We show the results for the most correlated counters: Last Level Cache misses, resource stalls, retired/issued micro-operations, retired instructions and floating point operations.

Off-line applications characterization

We created a Design Space Exploration flow to detect and profile the best resource allocations of applications. The output of the exploration is an enriched recipe that can be then used by the BarbequeRTRM to perform resource allocation at runtime.

The exploration schema is shown in Figure 17: the main component is the DSE engine, which uses ad-hoc management APIs to drive resource allocation in place of the BarbequeRTRM scheduling policy. The DSE engine runs the target application multiple times to test

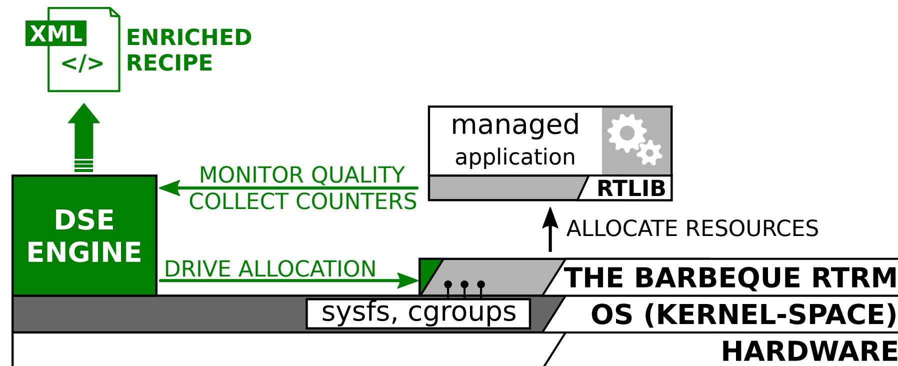


Figure 17: Off-line applications characterization. The DSE engine drives the resource allocation and collects the characterization information from the Runtime Library. At the end of the exploration, the DSE engine selects the best resource configurations and translates them into the enriched AWMs that will compose the application recipe. Our contributions are highlighted in green.

different resource allocations. During each execution, it also collects the characterization information that is computed by the Runtime Library. Finally, after the exploration process is complete, the engine computes which are the best allocations in terms of memory pressure, Quality of Service and energy consumption, and it translates them into enriched AWMs.

Performance is often a proxy to quality: the more performing an application is, the more computation it can perform during a given time frame.

In the context of this work, we chose to use the throughput of applications as the main metric to compute Quality of Service. Figure 18 shows the relationship between parallelism level, allocated CPU time and performance for *bodytrack* and *ferret*, which respectively exploit data parallelism (i.e. it process data using multiple parallel threads) and task parallelism (i.e. it distributes the computation on a pipeline whose stages are implemented by one or more threads). In this experiment, the applications run on a single multi-threaded core, i.e., on two processing elements that share L1 cache, L2 cache and part of the pipeline. We run bodytrack with a number of threads ranging from 1 to 5. We instead run ferret with a number of threads-per-pipeline-stage equal to 1 and 2, which respectively mean 4 and 8 threads in total. It is worth noticing that, in both applications, communication overheads and cache contention cause performance to drop when the allocation passes from one to two processing elements (i.e., when the allocated CPU bandwidth gets slightly higher than 100%). This effect is especially severe in the case of bodytrack, which is more memory bound than ferret.

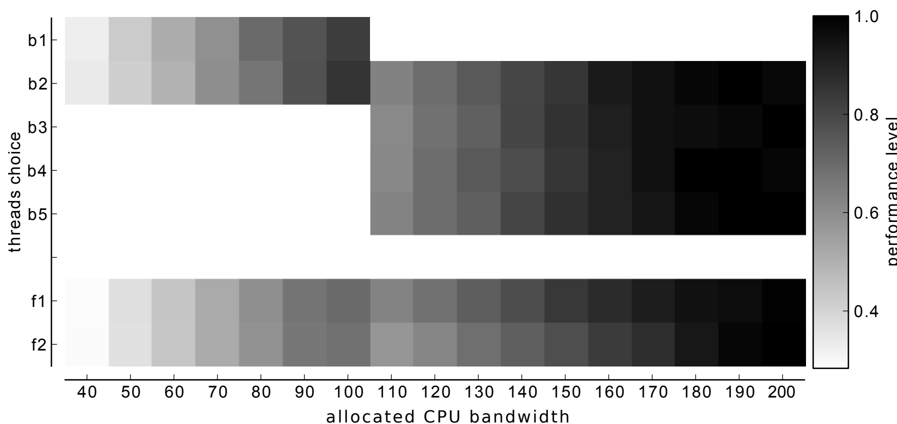


Figure 18: Off-line applications characterization: performance of bodytrack and ferret for different parallelism levels and allocated CPU bandwidth. The selected level of parallelism is referred to by using the notation AX, where A is the first letter of the application name and X is the number of threads. CPU bandwidth is instead expressed in percentage, e.g., 150% bandwidth means "one processing element and a half" worth of CPU time.

Runtime resource allocation

We extended YaMS, which, at that time, was the default scheduling policy if the BarbequeRTRM [73], in order to endow it with a energy- and memory contention-aware resource mapping support. We called the extension CoWs, which stands for "CO-scheduling WorkloadS".

CoWs has its own name because it is an independent extension module that can be plugged into the YaMs policy.

YaMS implements a heuristic that aims at (sub-optimally) solving the Multidimensional Multiple choice Multiple Knapsack problem (MMMKP) [74]. Basing on a multi-objective evaluation that takes into account performance, fairness and load balance, YaMS computes which AWM will be selected for each application. The goal of the CoWs extension is to perform a clever mapping of the selected AWMs on the available resources. In particular, after the AWMs have been selected, CoWs maps the allocated cores and memory nodes on the hardware while trying to spread both energy consumption and memory pressure evenly throughout the entire chip.

Experimental Results

In order to exhaustively evaluate our approach, we chose to carry out the tests using all the workloads that feature at least three applications and are composed by 0 to 4 instances of bodytrack and 0 to 4 instances of ferret. The execution time of the workloads spans from 30 to 60 seconds.

A workload is a group of applications that must be executed (possibly in parallel) on the target system. Its execution time is the time elapsed between the start of the first application and the end of the last one. Its size is the number of applications it contains.

We performed two different set of tests: first of all, we compared CoWs (i.e., YaMs using the CoWs extension) with YaMs in terms of workload execution time, energy consumption and Energy Delay Product. Second, we extended the comparison to the plain Linux case.

Figure 19 compares YaMs and CoWs in terms of execution time, energy efficiency and EDP. We tag workloads according to their composition: `BX FY` indicates a workload that is composed by `X` instances of bodytrack and `Y` instances of ferret. Being aware of the memory pressure that is generated by each application, CoWs is able to perform a memory contention-aware resource mapping, which leads to performance speedups up to 19% with respect to YaMs. However, given that CoWs tries to aggressively isolate memory intensive applications on subsets of the available resources, the speedups tend to decrease as the number of applications that compose the workload increases. In the case of `B4 F1` and `B3 F4` (5 and 7 applications, respectively), the isolation proves to be excessively aggressive and leads to a slight performance degradation. Overall, the average performance speedup induced by CoWs equals to 5.63%. From the energy efficiency perspective, CoWs outperforms YaMs in all scenarios. In this case, the improvement is quite homogeneous, and it is positive even in the congested scenarios. Overall, the average energy consumption speedup equals to 7.43%. Regarding EDP, which equally takes into account ex-

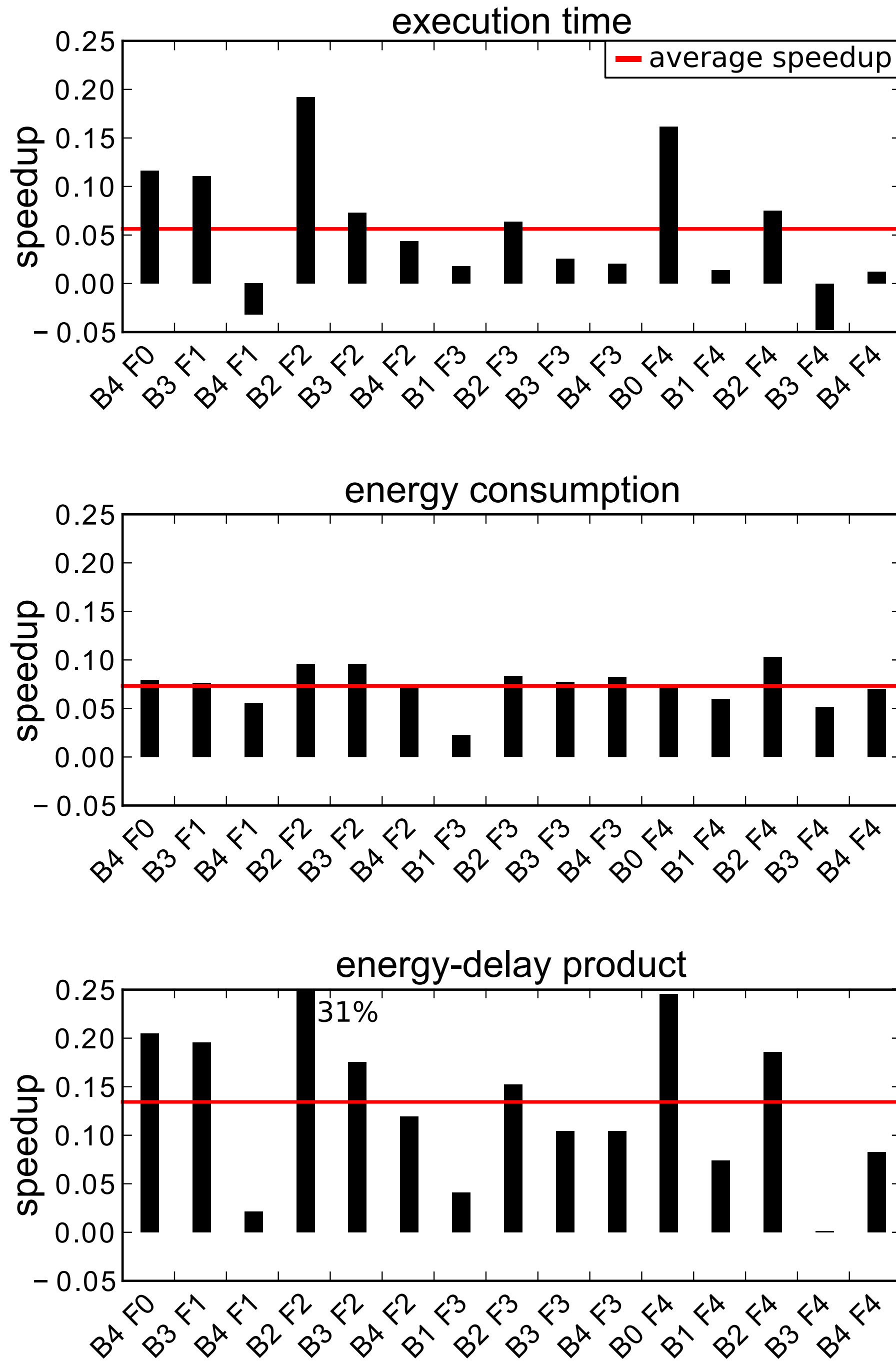


Figure 19: Execution time, energy consumption and EDP speedups induced on the CoWs extension on YaMS. Workloads are tagged according to their composition: BX FY means "X instances of bodytrack, Y instances of ferret".

ecution time and energy, CoWs provides an average speedup equal to 13%, with a peak of 30% in the B2-F2 scenario. Overall, the results show that characterizing the applications in terms of power consumption and memory intensiveness allows scheduling policies to allocate resources more efficiently than in characterization-agnostic scenarios.

Before presenting the comparison between the managed approaches (i.e., those that employ the BarbequeRTRM) and the plain Linux one, it is worth remarking that, in the latter case, applications do not rely on a resource manager and are therefore unable to configure their parallelism level according to the system load. Unmanaged applications suppose to be running alone; hence, they usually spawn one thread per active core, which can possibly result in a system congestion if the workload is composed by many applications. In order to perform a fair comparison between managed and unmanaged approaches, we chose to split the Linux scenario in three cases, each one employing a different approach to select the parallelism level of applications:

AGGRESSIVE (AGGR)
: each application supposes to be running alone on the system and hence spawns one thread per online processing element;

CONSERVATIVE (CONS)
: applications are aware of the workload size and hence select a threads number equal to the number of online processing elements divided by workload size;

ORACLE (ORAC)
: we manually selected an optimal number of threads for each application according to the workload composition.

Please note that, in the Linux scenarios, applications select a parallelism level and are not able to change it throughout their entire execution. Conversely, in the managed scenarios, applications are aware of the amount of resources that they have at their disposal and are therefore able to tune their parallelism level accordingly.

Figure 20 shows the comparison results. We plotted performance, energy consumption and EDP of the workloads for each scenario. Moreover, in order to provide a high level view of the effects of resource management, we also plotted the average results for managed (YaMs and CoWs) and unmanaged (AGGR, CONS and ORAC) scenarios.

From a high level perspective, managed scenarios lead to better results than unmanaged ones. The main advantage of letting Linux orchestrate the resource management is that applications are potentially able to better exploit the available CPU time. Indeed, each of the threads that compose the workloads is free to migrate to any unused processing element, thus minimizing resource under-usage. However, this approach also has a disadvantage: allowing threads to freely migrate on any processing element induces cache thrashing phenomena, which in turn negatively affect performance and hence energy consumption. Relying on a resource manager leads instead to results that consistently outperform those of the unmanaged scenarios. In fact, selecting a suitable parallelism level for applications is only half

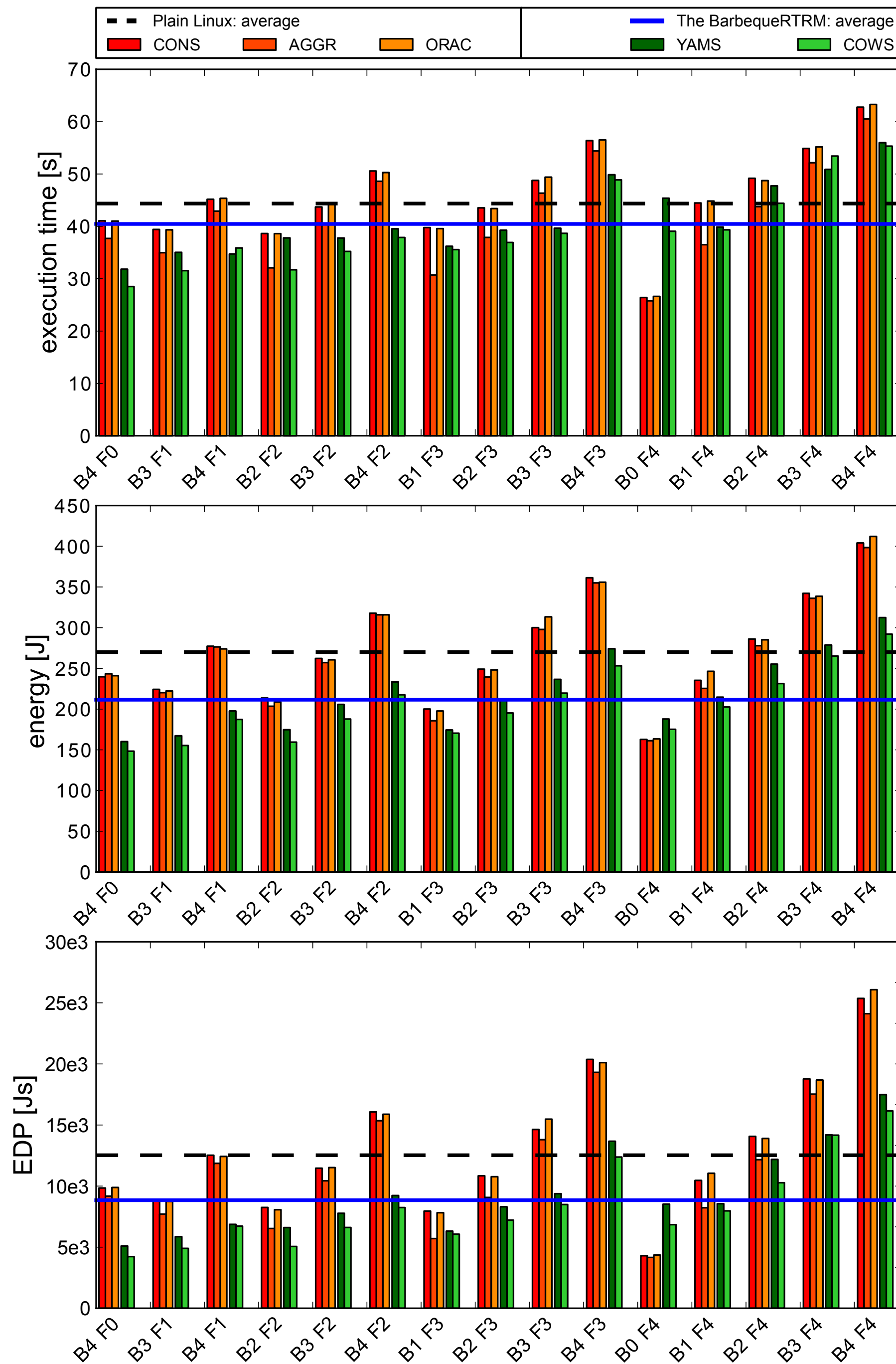


Figure 20: Linux scheduler compared to CoWs with respect to workload execution time, system wide energy consumption and energy-delay product. We employed three parallelism levels (AGGR, ORAC and CONS) for the standard Linux experiments, and we compared them with both YaMs and the CoWs.

of the problem: mapping threads to the most suitable processing elements is also paramount in order to avoid cache thrashing and to allow threads from the same applications to share the same caches. This in turn leads to an average 12% speed-up in workload comple-

tion time and to a corresponding 32% reduction on the system-wide energy consumption. Even more important, this resulted in a 50% improvement of the EDP metric.

Finally, it is worth noticing that, although it possibly leads to a system congestion, selecting the applications parallelism level using an aggressive approach leads to a lower workload execution time with respect to both the oracle and conservative scenarios. The reason is straightforward: given that the applications that compose the workloads feature different execution times, some applications are bound to terminate before the others. Using a high number of threads indeed leads to a system congestion; however, as the number of running application decreases, applications that spawned a high number of threads and are yet running benefit from a substantial performance boost with respect to the ones that limited their parallelism level. Taking into account the system load when selecting the parallelism level is therefore useful only if applications are able to dynamically tune it as the composition of the workload changes. It follows that, in those cases, using a resource manager is paramount to guarantee an optimal parallelism selection.

Conclusions

In this work, we employed Design Space Exploration techniques to characterize resource usage and energy consumption of applications. We extended YaMs, which is a multi-objective scheduling policy of the BarbequeRTRM, in order to provide it with memory-contention and energy consumption-aware mapping capabilities. Experimental results show that, by exploiting the characterization information to suitably map applications, the resource manager is indeed able to optimize its scheduling decisions.

We also stressed the fact that the relationship between performance counters and energy consumption primarily depends on the target architecture and on the workload that we must execute. Hence, using performance counters as a proxy to energy is possible, but it also requires a significant effort on the characterization side.

Finally, we shown that, in order to maximize the Quality of Service that can be squeezed out from the available resources, applications need to constantly update their software parameters in order to continuously adapt them to the system status.

MAKING APPLICATIONS ADAPT TO ALLOCATIONS

In the previous section, we tried to understand how resource managers can exploit the characteristics of applications to optimize the resource allocation. However, there are still open issues: first of all, in order for the characterization problem not to be NP-complete, each

The contents of this section is partially published in [75]. You may want to consult Appendix A before venturing forth.

application should be characterized separately from the others. This means that, although the characterization information provides at least a general idea of how an application will behave when running on shared resources (e.g. a memory bound application will suffer from cache sharing), we cannot know a priori how the application will behave at runtime. Second, even if an application is running alone on the available resources, using different input data or even changing the frequency of some cores may dramatically change how the application will exploit the allocated resources. Third, while a resource manager exactly knows which are the system-wide goals that must be complied with (e.g., minimization of power consumption, energy consumption and number of thermal hot-spots), it is not always clear what applications expect from the resource manager. Not surprisingly, the satisfaction of running applications is usually referred to with the umbrella-term "Quality of Service", which does not necessarily mean "the more allocated resources, the better". Given a sub-optimal resource allocation, managed applications can complain with the resource manager, which, as already stated, may not be completely aware of what each application really wants; or they can try to tune their behavior (e.g., parallelism level or accuracy of the results) in order to maximize their Quality of Service despite the sub-optimal allocation.

Given the premises, is it feasible to move part of the optimization effort to the applications side?

The term "dynamic auto-tuning" refers to the ability of applications to dynamically change their behavior in order to adapt to the runtime-variable system status. Basically, some of the static software parameters of applications (e.g. number of threads or accuracy of the results) can be transformed into dynamic knobs that can be manipulated to maximize the Quality of Service of the application despite a runtime-variable resource availability [76]. The concept of auto-tuning is also a central component of approximate computing: by dynamically changing their configuration, applications are able to select the optimal trade-off between accuracy of the results and performance [77].

As already discussed in Section 3.2, allowing applications to autonomously compute their configuration could lead to instability. For instance, two applications that run on the same processor could initially choose to employ one thread per core. Then, noticing that the processor is congested, both of them would scale down their threads number. Finally, noticing that the system is now less loaded, each of them would be tricked into thinking that it can again scale the number of threads up, thus creating a periodical oscillation between resource under-usage and congestion. This kind of problem is well known, and it derives from the fact that application-specific auto-tuners lack a global system view and therefore suppose that the ap-

Achieving an optimal system-wide resource usage withouth recurring to an arbiter is difficult if not unfeasible.

plication that they are managing is running alone on the available resources – which, by the way, is usually a feasible assumption, since auto-tuning approaches are mostly employed in HPC scenarios.

In this Section, we will study how a resource manager and an application auto-tuner can cooperate in order to achieve an optimal resource utilization on a multi-core platform. Whereas a resource manager has a system-wide view of both resources and applications and, most importantly, is able to allocate a set of exclusively owned resources to each application, the auto-tuner is application-specific and is therefore able to configure applications so that they efficiently exploit the allocated resources.

We propose a framework that is based on the combination of runtime resource management and OpenCL application auto-tuning. We show that application auto-tuning, which, in this case, is based on a design-time analysis, can become synergistic with run-time resource management. In the proposed framework, the system-wide resource manager is in charge of allocating resources; however, each application autonomously takes runtime decisions in order to optimally exploit the available resources.

Motivation

Modern architectures expose ever increasing parallelism capabilities. On one hand, the number of processing elements on the same chip is constantly growing. For instance, the Intel Xeon Processor E7 v4 Family features up to 24 cores [78], while the Nvidia Titan Xp GPU leverages 3840 CUDA cores [79]. A similar trend can be also observed for embedded computing platforms, e.g. Nvidia Tegra X1 [80] or the Adapteva's Parallela board [81]. On the other hand, it is also common to integrate different types of accelerators on the same platform, thus providing better energy efficiency and a higher throughput to the application developers. This trend is common to different architectures that target embedded systems on one side, and high-performance computing (HPC) on the other.

In order to facilitate the exploitation of the aforementioned parallelism capabilities, applications developers are supported by specialized programming models. Although the industrial players proposed custom paradigms that target their own many-core platforms, the convergence of architectures is now pushing towards more generic and portable programming models, and, among those, the OpenCL industry standard is regarded as one of the most established solutions [82].

Indeed, the increasing computational power of many-core accelerators and the availability of portable parallel programming models enable a new set of challenging possibilities at the software level. In particular, from a system-level perspective, it is now possible to con-

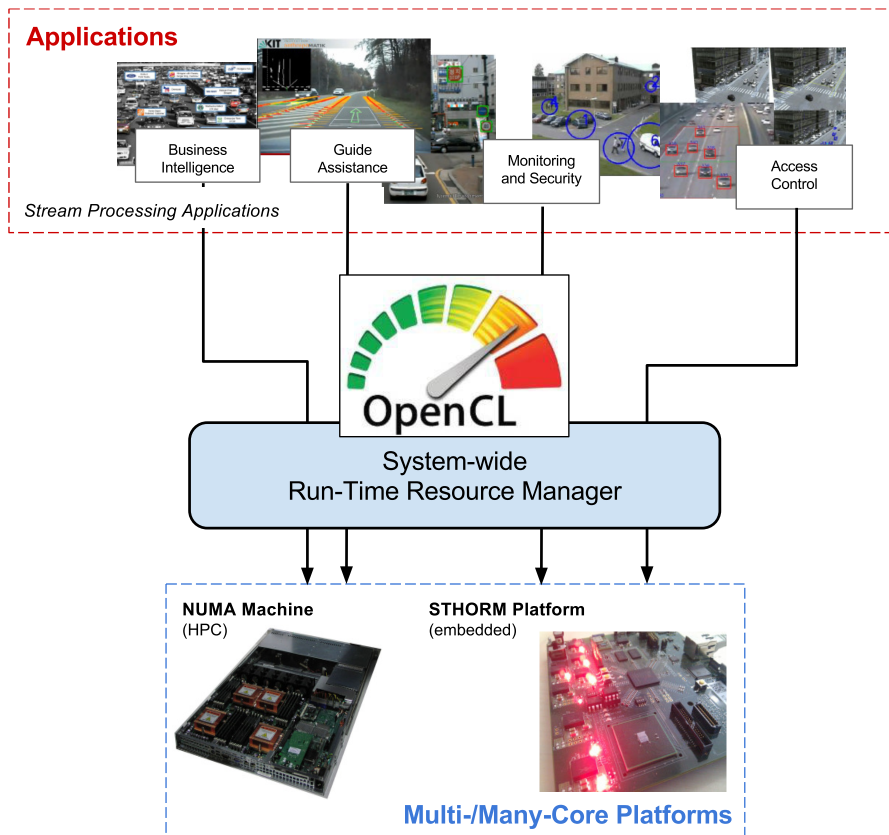


Figure 21: Application and platform domain of the proposed methodology. The OpenCL runtime is leveraged on top of a run-time resource manager, for efficient allocation of platform resources.

currently run multiple applications that have different priorities and requirements, and this enables multi-application scenarios where all types of applications compete for the usage of computing resources. This is true even in the case of mixed-critical systems on embedded computing platforms: in this case, whereas the workload is usually known at design-time, its run-time variability is still unpredictable. It follows, then, that in order to enable a system-level optimization, resource allocation and application auto-tuning should take into account the dynamic resource and quality demands of each application.

For the application domain (see Fig. 21), we target "stream processing applications" such as video processors and augmented reality. In this kind of scenarios, the set of possible applications is known at design-time, while the run-time workload mix is unpredictable: each application could start and terminate at different time instants. Moreover, each application can exhibit a different criticality, thus being associated to a corresponding priority level or, depending on the system status or application-specific resource requests, being subject to time varying requirements.

Methodology

In order to address both system-wide and application optimization, we propose an approach based on the synergy between design-time and run-time techniques. System-wide resource management is operated by the Barbeque Run-Time Resource Manager (BarbequeRTRM), which is described in Appendix A.

At design-time, we exploit Design Space Exploration (DSE) techniques to profile the behavior of applications that run on the target platform and to identify a set of optimal application configurations. The main idea behind the exploration is that, for any given resource allocation, the application may be able to achieve different trade-offs between performance and accuracy by dynamically changing its configuration. Accordingly, the output of the DSE is a BarbequeRTRM application recipe (see Section A.3), and each of the Application Working Modes defined in the recipe is associated to a set of optimal configurations. The DSE follows the schema presented in Figure 22. First, the application is executed multiple times under different configurations; second, each configuration is evaluated against the others in order to identify the most efficient configurations for a given set of quality metrics (e.g., performance and accuracy). Finally, the efficient configurations are clustered basing on their resource usage, thus identifying the AWMs that will compose the application recipe.

At run-time, the methodology is based on the cooperation of the BarbequeRTRM with an Application-Specific auto-tuner [83]. Depending on the resource allocation, i.e., on the Application Working Mode selected by BarbequeRTRM, the auto-tuner is able to dynamically tune the application software parameters by picking one of the configurations that are associated to the selected AWM.

Experimental Results

To evaluate our approach, we implemented an OpenCL version of the area-based local stereo matching algorithm described in [84]. We designed the application so that it exposed a set of software parameters that affects both application and platform metrics. Basically, the algorithm computes "stereo disparity", i.e., the difference in position between corresponding points in multiple images; hence, we used the percent disparity error as a proxy for assessing the quality of the results.

First, we performed an experiment to evaluate the capability of applications to auto-tune their configuration in order to achieve a constant throughput (at the cost of accuracy) despite a runtime variable resource availability. This is a common practice when dealing with multimedia streaming applications: a typical example is that of a web-based movie-player that reduces the frame resolution in

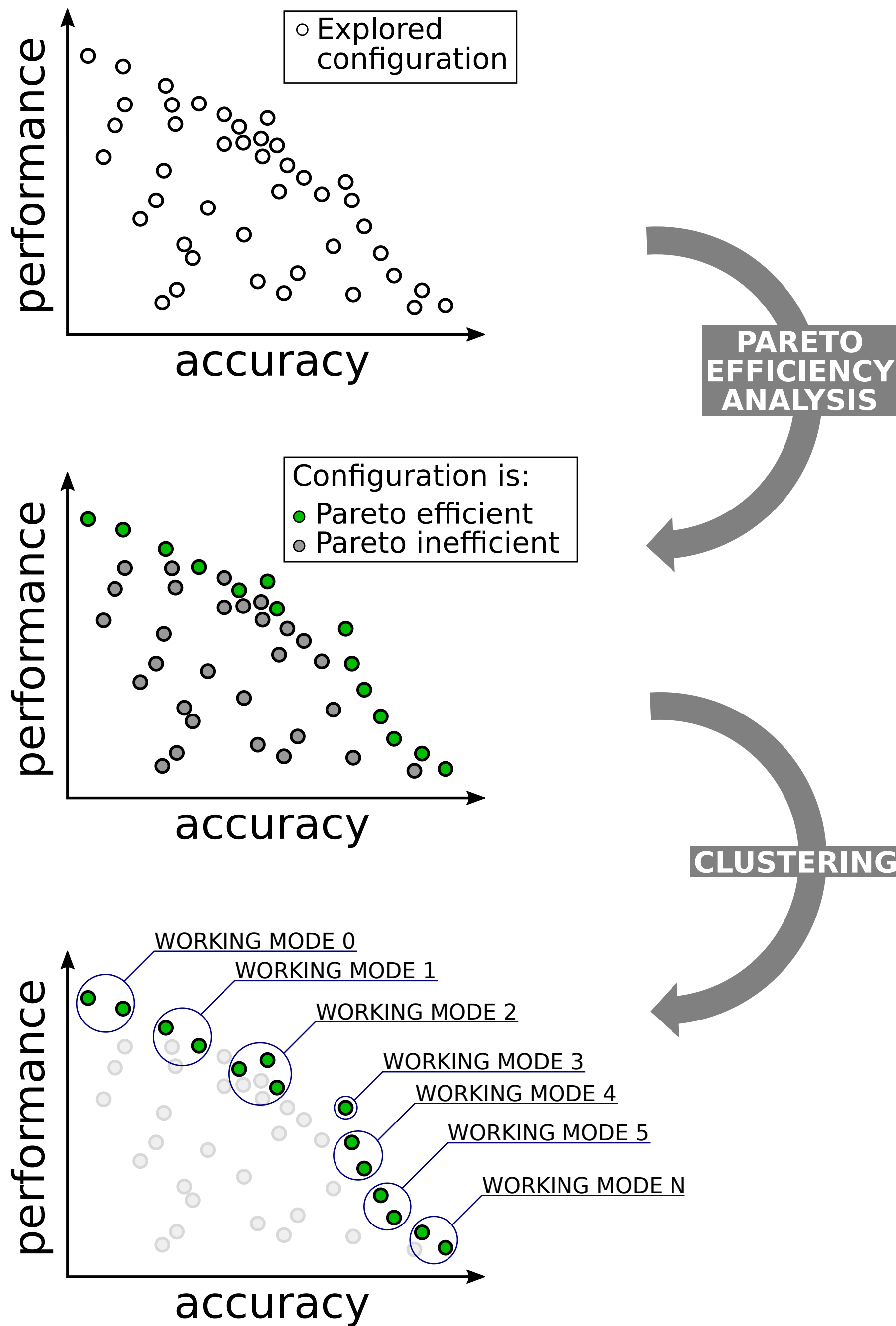


Figure 22: Design Space Exploration phase. The application is executed using multiple times using different configurations. Each configuration is then checked for Pareto efficiency. The resulting configurations are then clustered basing on the resources they require, thus creating the Application Working Modes that will compose the BarbequeRTRM recipe for the target application.

order to maintain a frame-rate that is comfortable for the user. We started multiple instances of the Stereo-Matching application at different time instants: 0, 20, 60 and 100 seconds. We performed the test in two configurations: in the first one, the applications employed a

constant configuration and were able to exploit all the available resources as in a plain Linux scenario. In the second one, conversely, applications employed our framework.

The results are shown in Figure 23. In the plain Linux scenario (23-a), the applications employed a constant configuration; therefore, the accuracy of their results was constant (approximately 45%) and their throughput varied according to the system load: as can be clearly seen in Figure 23-a, the throughput of applications wildly changes as the number of active instances of Stereo-Matching increases. On the contrary, in the runtime-managed scenario (23-b), the resource manager and the application auto-tuner collaborate to achieve an optimal throughput for each instance (in this case, 4 frames per second). Each time an application starts or terminates, the resource manager updates the resource allocation and the auto-tuner, which is aware of the selected allocation, tunes the application configuration in order to comply with the throughput goal.

In the second experiment, we analyzed how the framework fares when managing dynamic workloads. By dynamic workload, we mean a set of applications with different schedules (start time), amount of data to process (number of frames in Stereo-Matching) and performance requirements (frame rate). This experiment aims at mimicking

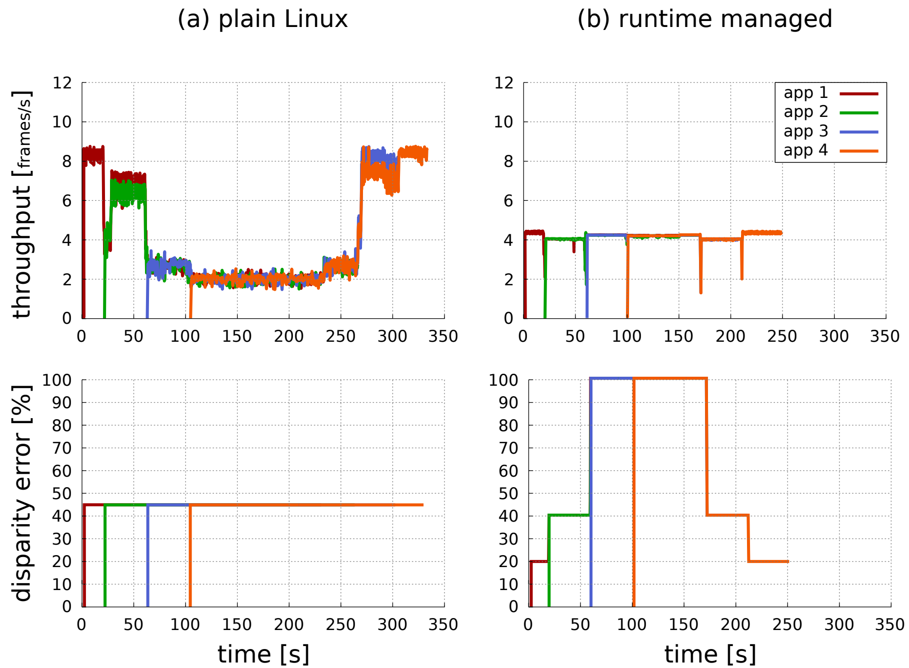


Figure 23: Throughput and percent disparity error for a plain Linux implementation and for the proposed run-time management strategy. In the Linux scenario, the applications run with a constant quality and a resource-availability-dependent throughput; in the managed scenario, instead, the applications are able to achieve a constant throughput by trading off quality with performance.

the typical workload of resource consolidation scenarios, where multiple heterogeneous applications are confined in a small subset of the system computational resources [85]. Although we use only one type of application (Stereo-Matching), we mimic a dynamic workload by exposing the following parameters:

START DELAY
: the start time of an application instance;

AMOUNT OF INPUT DATA
: the number of frames to be processed;

FRAME-RATE GOAL
: the target throughput, as demanded by the user.

Figure 23 shows the results of a scenario where we executed four concurrent instances of the Stereo-Matching application. In the Linux scenario, the applications run with a constant quality and a resource-availability-dependent throughput; in the managed scenario, the applications are instead able to achieve a constant throughput by trading off quality with performance.

Finally, Figure 24 shows the results with a number of concurrent instances that ranges from 1 to 6. The first plot shows the average percent distance from the throughput goal (the lower the better). Due to resource contention, indeed, the distance from the performance goal increases with the number of active applications. However, by exploiting the trade-off between accuracy and performance, the managed approach delivers a higher throughput with respect to the plain Linux one. The second plot focuses instead on performance predictability. Given that the managed scenario relies on a resource manager to ensure that each application is able to use its own set of resources, contention is limited to inherently shared resources such as interconnect and last level cache. Moreover, applications that run in the managed scenario feature a tunable configuration, so that the application-specific auto-tuner is able to make them execute at a constant throughput. It follows, then, that the managed approach is able to deliver a very high performance predictability with respect to the plain Linux scenario. The last plot reports the average error of the applications results. As expected, the plain Linux scenario is generally characterized by a higher accuracy (i.e., a lower error) with respect to the managed scenario. However, in scenarios that feature up to two concurrent applications, the available resources are enough for applications to tune their configuration in order to achieve a higher accuracy than in the plain Linux scenario, while complying with their throughput goal at the same time.

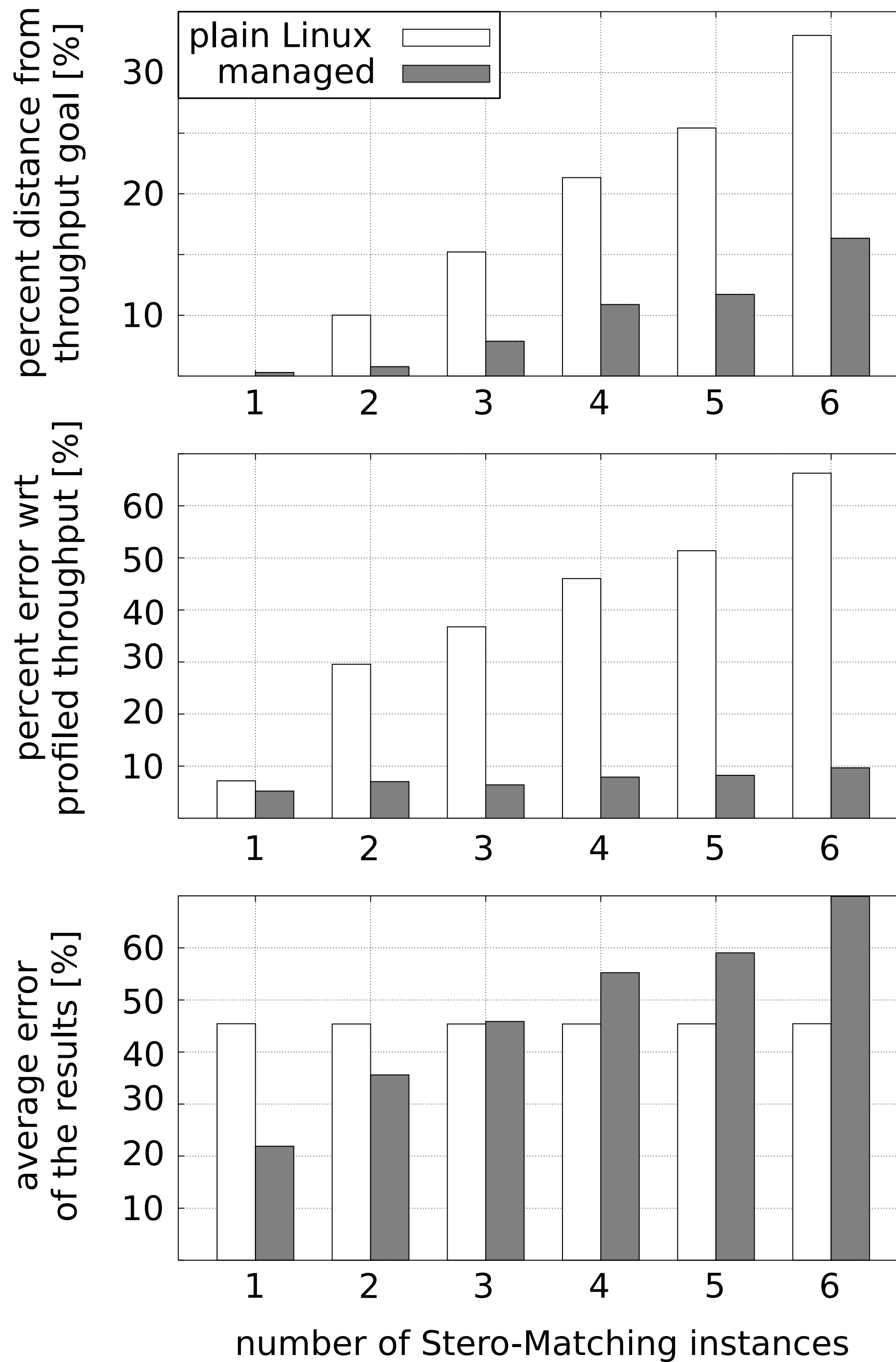


Figure 24: Dynamic workload analysis under a variable number of Stereo-Matching instances.

Conclusions

In this section, we shown the benefits that come from allowing a system-wide resource manager and an application-specific auto-tuner to work in a synergistic way. The main idea behind this work is that, whereas resource managers allocate resources to applications according to precise and known system-wide optimization goals, they are

often unaware of what applications (and users) really need. That is, each application defines its Quality of Service in its own way. We tackle this problem by moving part of the management complexity to the applications side: each application relies on an application-specific auto-tuner, i.e., a component that is specifically configured for the target application and is able to tune the application software parameters at runtime in order to make it comply with its quality of service goal despite a runtime-variable resource availability.

Our approach proved to be very effective in allowing applications to comply with their Quality of Service goal, which, in this case, consisted in maintaining a constant throughput at the cost of accuracy, with respect to a plain Linux scenario.

4 SINGLE-ISA HETEROGENEOUS PROCESSING: BIG.LITTLE ARCHITECTURES

As already mentioned in Chapter 3, multi-core systems must face the problem of co-running multiple tasks on the same processing elements. Heterogeneity introduces yet another degree of complexity, since tasks are allowed to execute on processing elements that have different characteristics. In this context, we will deal with Single-ISA heterogeneous processing, which, from a resource management perspective, is the most challenging flavor of heterogeneity: given that all the processing elements feature the same Instruction Set Architecture, tasks are able to migrate between them during runtime.

We will specifically address big.LITTLE architectures, which are multi-core systems that feature two clusters of cores: the one called "big", which is performing; and the one called "little", which is energy efficient. First, we will study how to dynamically migrate threads among the two clusters in order to maximize the usage of the big cluster while minimizing the performance losses that are induced by resource contention. Second, we will use the two clusters of cores as a heterogeneous OpenCL device. In this context, we will present a mechanism that forces the OpenCL runtime to view the big.LITTLE processor as a custom set of heterogeneous devices instead of viewing it as a single device.

My current Smartphone features a MediaTek helio X20 deca-core processor, which is "big-medium-little". That is, it features three clusters of cores. Indeed, heterogeneity proved to be a good asset for energy-efficient devices.

RESOURCE CONTENTION IN BIG.LITTLE ARCHITECTURES

The contents of this section are partially published in [35].

Single-ISA heterogeneous processors allow operating systems to exploit the trade-off between power and performance. An example is given by ARM *big.LITTLE* architectures, which exploit two clusters of cores: the LITTLE cluster, which features low performance, low power cores; and the big cluster, which features high performance, power hungry cores. Exploiting cores from either the *big* or LITTLE cluster according to the requirements of the running tasks allows *big.LITTLE* architectures to achieve energy efficiency [86, 87].

From now on, with "big" and "little" cores we will refer to cores from the high performance and low performance cluster, respectively.

In heterogeneous processors, different clusters are characterized by different processing capabilities; and different memory hierarchies in terms of cache size and, possibly, coherency protocols [88]. Moreover, as each cluster is itself a multi-core processor, it incurs in the well known problem of resource contention, which induces negative effects on performance and energy efficiency [89, 90]. Thus, heterogeneous schedulers must deal with two activities: *thread-to-cluster allocation* (i.e. mapping threads to clusters), which has a great impact

on both performance and energy efficiency [57, 91]; and *thread-to-core allocation* (i.e. mapping threads to cores inside the chosen cluster), which should mitigate resource contention by suitably co-scheduling threads on shared resources [61, 92].

We propose an approach to optimize co-scheduling on heterogeneous processors. In detail, we introduce the concept of *stakes function*, which represents the trade-off between *isolation* and *sharing* of resources. We demonstrate that isolating an application in a suitable subset of cores leads to benefits in terms of mitigation of resource contention and optimization of resource usage. Finally, we exploit *stakes functions* to drive a resource allocation policy that imposes constraints to the Linux HMP scheduler, achieving speed-ups in terms of both performance and energy efficiency.

Multi-core architectures feature few cores. Is isolating an application on a couple of cores preferrable to making it contend for all the resources?

We validated our approach using an ODROID-XU3 development board, which is based on the ARM big.LITTLE architecture and features two clusters of cores: a big cluster composed by four ARM CORTEX A15 cores; and a little cluster composed by four ARM CORTEX A7 cores.

Related Works

In a heterogeneous multi-threading scenario, the process of scheduling can be defined as a two-fold process:

- THREAD-TO-CLUSTER ALLOCATION: scheduling each thread on the most suitable cluster of cores
- THREAD-TO-CORE ALLOCATION: allocating each thread to the most suitable core in the selected cluster

Thread-to-core allocation is also typical of homogeneous multi-core processing and is a NP-complete problem[93]; however, low complexity algorithms compute sub-optimal but effective allocations by estimating performance degradation when multiple known applications run on a set of shared resources [94, 95, 96].

To do this, each application has to be characterized. Several previous works characterize applications using performance counters. Resource usage of applications can be computed using the number of *Last Level Cache misses* [97, 98], *L2 cache misses* [57], *L1 cache misses* [61], or micro-architectural events such as *Front Side Bus stalls*, *branch misses*, *stall cycles* [99, 95, 100, 96]. A given set of performance counters, however, may not always be optimal to characterize the resource usage of an application: for example, *L1 misses* may be preferred to *Last Level Cache misses* in case of particular architectures [61].

To tackle this problem, several works characterize applications using performance degradation [69, 64]. Each application is co-run with resource-hungry benchmarks that purposely create contention on multiple resources; the resulting performance degradation represents the

sensitivity of the application to resource contention. The estimated resource contention is used by co-scheduling algorithms to perform a resource-aware *thread-to-core allocation*.

Thread to cluster allocation is typical of heterogeneous processors, where the resources consist in multiple clusters of cores, each cluster with different capabilities.

Several previous works compute thread-to-cluster allocation decisions using retired instruction rate as a metric: big cores are exploited to run threads that have the highest average *Instructions per Cycle* value[101] or the highest local *Instructions per Second* value[91, 102], while the other threads run on the little cores.

A previous work proposes fairness as an optimization target for co-scheduling choices [103]. They propose two scheduling algorithms: *equal-time scheduling*, where each thread runs on each cluster in a round-robin fashion; and *equal progress scheduling*, where the threads that are experiencing the higher slowdown are dynamically allocated to the big core.

The work presented in [104] proposes a task scheduler that consists of a history-based task allocator and a preference-based task scheduler. The history-based task allocator allocates tasks with heavy workloads to fast cores, and tasks with a light workload on slow cores. The task-to-cluster allocation is static and takes into account the historical statistics that are collected during the execution of each application. The preference-based task scheduler dynamically adjusts the allocation to ensure load balance and to correct sub-optimal scheduling choices.

To the best of our knowledge, co-scheduling algorithms for heterogeneous processors are still too focused on the choice of the best cluster on which executing each thread. Thread-to-core allocation, which was a central issue in multi-core co-scheduling policies, is giving ground to thread-to-cluster allocation; however, as extensively shown in previous literature[105, 106, 107, 61], mitigating resource contention also at cluster level (i.e. performing thread-to-core optimizations) leads to benefits in terms of both performance and energy efficiency, both of which are essential in heterogeneous processors.

Methodology

Our co-scheduling policy sets constraints on the Linux HMP scheduler, allowing it to perform scheduling choices that are implicitly *resource* and *energy-aware*. The scheduling policy provides *thread-to-cluster allocation* and only a partial *thread-to-core allocation*: it does not allocate threads to cores; instead, it allocates threads to *subsets* of cores.

In Figure 25 on page 63 we present an overview of the proposed flow. The core of the approach is the concept of *stakes function*, which

is exploited during runtime by the co-scheduling policy to dynamically compute the size of the subset of cores in which each application will be isolated. Once selected a subset of cores, *thread-to-core allocation* is performed, as usual, by the standard Linux HMP scheduler. During design time, we analyze each application separately and build its *stakes functions*, one for each cluster. The characterization process is composed by two phases—CPU demand analysis and memory sensitivity analysis—which characterize the application in terms of both required CPU bandwidth and sensitivity to memory contention.

We define *CPU demand (γ)* as the average CPU bandwidth usage of an application.

The performance of an application is strongly dependent from its γ and the number of cores at its disposal: for example, an application with $\gamma = 1.5$ CPUs can reach its maximum performance if scheduled on two cores. Conversely, it would suffer if scheduled in a single core, regardless of memory contention or similar side-effects: a single core is not enough to provide that bandwidth to the application.

The CPU demand analysis profiles the *CPU demand* of each application A_t (application A, t threads), both on *big* and *LITTLE* clusters. During the analysis, we run A_t *alone* on the chosen cluster (*solo run*), while we isolate all the other applications in the unused cluster. The profiled CPU bandwidth, which we compute using performance counters, is the *CPU demand* of A_t (γ_{A_t}).

The computational complexity of this first characterization step is $O(NT)$, where N is the total number of applications, T is the average number of configurations (number of threads) per application.

The memory usage behavior of an application is called *memory intensiveness*. Memory intensive applications bring a lot of data in the caches, an this can lead to memory contention with the co-runners (if any). An application is *memory sensitive* if it experiences a substantial performance degradation when co-running with memory intensive applications. In the next sections, μ will refer to the *memory sensitivity* of applications.

During the memory sensitivity analysis, we co-schedule each application A_t with a synthetic memory intensive benchmark that performs a set of memory accesses denoted by a high memory accesses count and a poor cache line reuse. The benchmark we implemented is very simple: it continuously and randomly accesses all the cache lines of the Last Level Cache, contending memory with A_t.

Even if isolated on different cpusets, applications will contend on the LLC, hence the stress isolated run.

Weexecute each test in two configurations: *a*) *stress run*, where A_t and the benchmark can migrate on any core of the cluster; and *b*) *stress isolated run*, where the A_t and the benchmark are isolated each on a subset of cores, thus incurring in memory contention only in the Last Level Cache, which is shared among all the cores of each cluster.

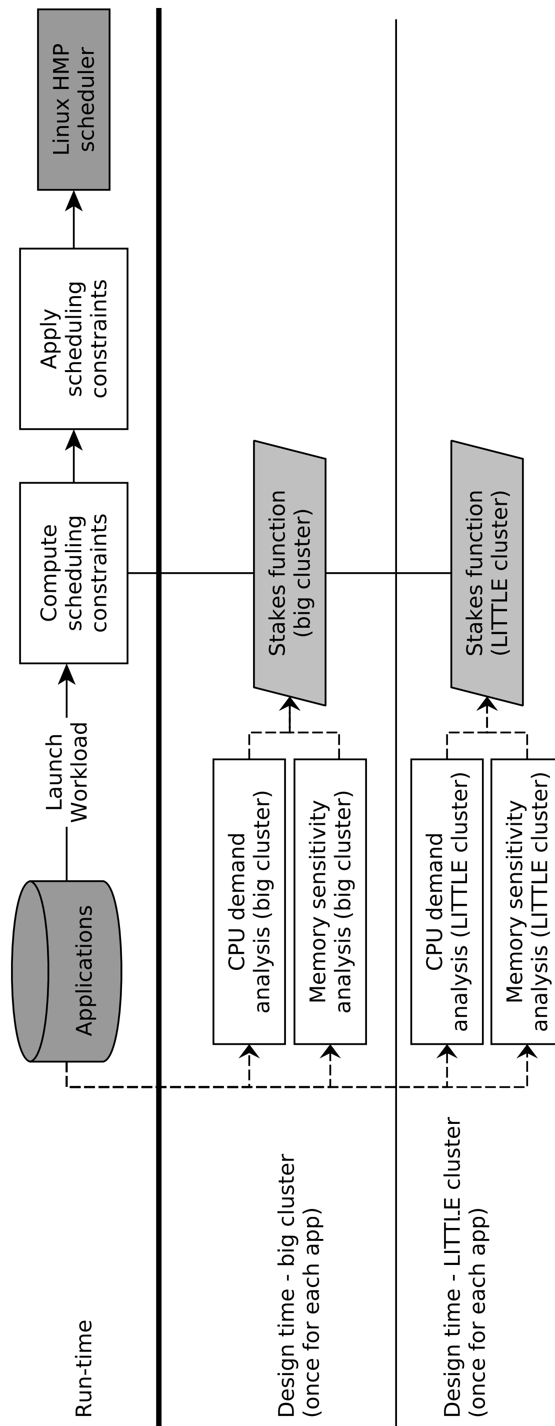


Figure 25: The proposed approach: each application is analyzed separately to build its *stakes functions*, one for each cluster. The characterization process, which is carried out at design time, is composed by two phases: CPU demand analysis and memory sensitivity analysis. Stakes function are exploited during runtime by the co-scheduling policy, which set constraints to the standard Linux HMP scheduler.

The maximum and minimum *memory sensitivities* ($\mu^{M}_{A_t}, \mu^{m}_{A_t}$) are the performance degradations experienced by A_t during the two *stress runs*, with respect to the results of the *solo run*.

We perform the analysis once for each application, both on the big and the LITTLE cores. The computational complexity of this second and last characterization step is again $O(NT)$.

To better understand how we compute stakes functions, we first introduce two concepts: *dynamic bandwidth* and *exposure to memory contention*.

DYNAMIC BANDWIDTH The dynamic bandwidth of an application A_t ($\hat{\gamma}_{A_t}$) is the fair quantity of CPU quota that we allocate to A_t during runtime. We compute dynamic bandwidth for an application A_t as shown in Equation 1, where N_c is the number of cores in the current cluster (big or little) and Γ_c is the total CPU bandwidth of the applications running on the cluster, including the bandwidth of A_t.

If a resource manager decides that multiple applications will run concurrently, it expects all of them to comply with their quality requirement. In this context, "fair" does not mean "all the applications have the same amount of resources": it means "all the applications are equally content with their allocation".

$$\hat{\gamma}_{A_t} = \min\left(\gamma_{A_t}, \frac{\gamma_{A_t}}{\Gamma_c} N_c\right) \quad (1)$$

Example. Let $N_c = 4$, $\gamma_{A_t} = 2.5$ and $\Gamma_c = 5.5$, that is, we are computing $\hat{\gamma}$ on a quad-core cluster, for an application that requires 2.5 CPUs and runs in a workload that requires 5.5 CPUs in total. Given that the cluster features 4 CPUs, we allocate to A_t only $\frac{N_c}{\Gamma_c}\gamma_{A_t} = \frac{2.5}{5.5} * 4 = 1.81$ CPUs.

EXPOSURE TO MEMORY CONTENTION Exposure to memory contention (E) represents how an application is exposed to memory contention when scheduled on a subset of cores of size γ_s. We define exposure of an application A_t as the CPU bandwidth that is offered by the subset of cores but is not used by A_t, with respect to the case where A_t is not isolated at all. Exposure to memory contention is computed as shown by Equation 2, where N_c is once again the cluster size.

$$E = \max\left(0, \frac{\gamma_s - \hat{\gamma}_{A_t}}{N_c - \hat{\gamma}_{A_t}}\right) \quad (2)$$

Example. The application from the previous example (A_t) will be allocated 1.81 CPUs. If A_t is not isolated, its co-runners will use $4 - 1.81 = 2.19$ CPUs, possibly triggering memory contention on all the caches. In this case, $E = \frac{4-1.81}{4-1.81} = 1$. Conversely, isolating A_t on a set of two cores, its co-runners will use only $2 - 1.81 = 0.19$ CPUs from the subset, leading to an exposure of $E = \frac{2-1.81}{4-1.09} = 0.08$. In this case, the data of A_t will be concentrated in few caches, and few applications will execute in the same subset of cores due to the fact that the CPU bandwidth offered by the subset of cores is mainly used by A_t.

COMPUTING STAKES FUNCTIONS *Stakes functions* evaluate the possibility to isolate each application in a subset of the available cores, instead of leaving it free to run on any core. The higher the function value, the better is the choice of the subset size.

Stakes functions do not take into account the *memory intensiveness* of the specific workload (i.e. the applications that are co-running with A_t): they represent the risks A_t is incurring into if running on a subset of cores while other applications are co-running on the same processor. In other words, *stakes functions* give a hint on how much an application would suffer, in the worst case, if scheduled on a subset of cores of a given size.

We compute stakes functions as shown in Equation 3. $S_{A_t,c}$ is the *stakes function* for application A_t, cluster c ("big" or "LITTLE"). The arguments of the function are γ_s, which is the CPU bandwidth quota under evaluation (i.e. the size of the core subset under evaluation), and Γ_c, which is the total CPU bandwidth required by the applications running on the cluster c.

$$S_{A_t,c}(\gamma_a, \Gamma_c) = \underbrace{\min\left(\frac{\gamma_s}{\hat{\gamma}_{A_t}}, 1\right)}_{A} \underbrace{\left(1 - \hat{\mu}_{A_t}\right)}_{B} \tag{3}$$

The stakes function is composed by two distinct contributions. The first contribution (A) estimates how performance is affected by isolating the application on the subset of cores, regardless of memory contention. The numerator is the number of cores that will be allocated to A_t, while the denominator is the dynamic bandwidth, that is, the number of cores that should be allocated to A_t.

Example. The application from the previous example (A_t) can use 1.81 CPUs. If allocated on two cores, A_t will be able to reach its maximum performance ($\min(\frac{2.0}{1.81}, 1) = 1$). Conversely, if allocated on a single core, the performance of A_t can be estimated as $\min(\frac{1.0}{1.81}, 1) = 0.55$. That is, regardless from memory contention, A_t will be experience at least 45% of performance losses due to resource underassignment.

The second contribution (B) represents how the performance of A_t is affected by resource sharing in the worst case scenario. $\hat{\mu}_{A_t}$ is the *expected memory sensitivity* of A_t, and represents the performance degradation of A_t in case of memory intensive workloads. As a consequence, $1 - \hat{\mu}_{A_t}$ estimates the performance of A_t when sharing resources in case of high memory contention.

The expected memory sensitivity is computed as shown in (4), and is a function of the *exposure to memory contention*. Each application A_t suffers the maximum performance degradation when it is totally

exposed to memory contention ($E = 1$), while it suffers the minimum performance degradation when totally isolated ($E = 0$).

$$\hat{\mu}_{A_t} = \left(\mu^M_{A_t} - \mu^m_{A_t}\right)E + \mu^m_{A_t} \quad (4)$$

We already know the minimum and maximum performance degradation of A_t from the *memory sensitivity analysis*: $\mu^m_{A_t}$ and $\mu^M_{A_t}$, respectively. For the sake of simplicity, we assume the degradation to be linear between $E = 0$ and $E = 1$. The assumption is certainly not accurate, but please note that $\hat{\mu}_{A_t}$ is in any case always greater than $\mu^m_{A_t}$ and lower than $\mu^M_{A_t}$ by definition.

Example. From the previous example, isolating A_t on two cores would give $E = \frac{2-1.81}{4-1.09} = 0.08$. Let the performance degradation of A_t (worst case) be always greater than $\mu^m_{A_t} = 5\%$ and lower than $\mu^M_{A_t} = 25\%$. The estimated performance degradation of A_t is $\hat{\mu}_{A_t} = (0.25 - 0.05)0.08 + 0.05 = 6.6\%$.

The proposed policy is based on two concepts: *applications acceleration* and *resource contention mitigation*. The first concept is straightforward: the big cluster is usually exploited as an accelerator, while most of the applications execute on the *LITTLE* cluster for energy efficiency purposes. However, a sub-optimal usage of the big cores may not be the most energy efficient choice. As shown in Table 2, co-running multiple threads on the accelerator leads to a lower power consumption per thread; therefore, we propose to allocate the accelerator to as much threads as possible, provided that the consequent performance degradation does not lead to energy inefficiency.

Table 2: Average Power [W] and energy consumption [J] of applications from the PARSEC benchmark suite on an ODROID-XU3 development board (big cluster). Number in parentheses in the first column indicate the number of threads. The last column indicates the energy saving achieved by co-running the applications instead of running them sequentially.

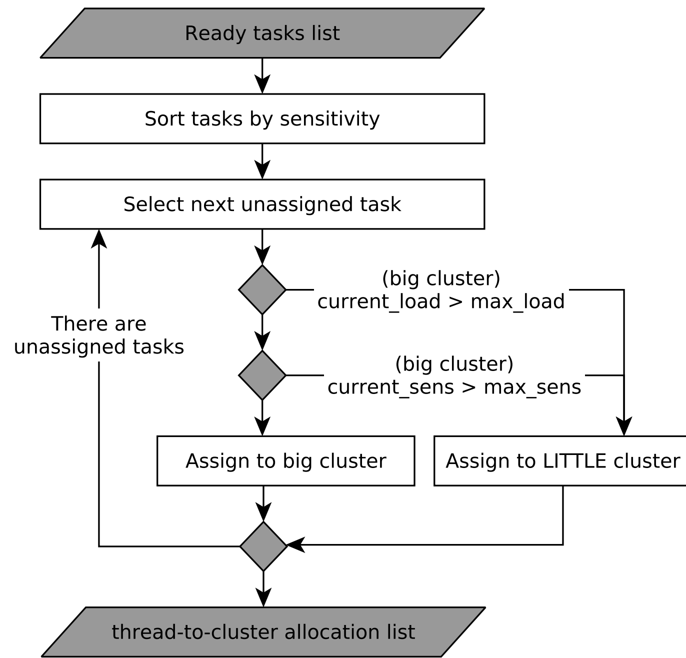
	Solo run		Co-run		
Application	power	energy	power	energy	energy speed-up
bodytrack (1)	3.011	56.364	5.343	278.489	9.82%
fluidanimate (2)	4.876	222.123			
swaptions (1)	4.925	252.137	5.526	388.752	17.15%
facesim (2)	3.771	136.614			
freqmine (3)	4.030	24.939	5.458	130.992	13.22%
blackscholes (3)	5.147	126.001			

Concerning *resource contention mitigation*, we propose to isolate the applications with high memory sensitivity into CPU partitions (subsets of cores), leaving the Linux HMP scheduler free to make scheduling decisions inside each partition. By doing so, the data of the applications is concentrated into a subset of the cache hierarchy and is less sensible to cache trashing induced by co-runners. On the other hand, applications with low memory sensitivity do not require to be isolated, and are completely subject to the Linux scheduler choices. This fosters a better utilization of the processing resources. The amount of cores to be allocated to an application comes from the configuration reporting the highest *stakes function* score.

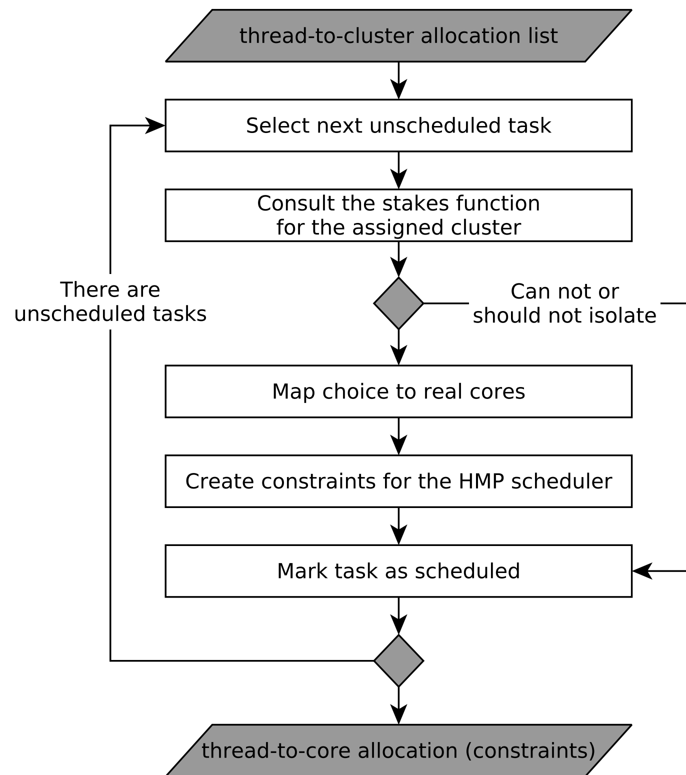
The policy is activated each time a task starts or terminate. The first phase is called *thread-to-cluster allocation* and provides *application acceleration*. We want to avoid unacceptable performance degradations, and we do this by allocating tasks to the big cluster until a *load* or *sensitivity threshold* are reached. The *load threshold* limits the number of threads concurrently running on the big cluster to avoid congestion. The *sensitivity threshold* represents the number of threads that have been scheduled on the big cluster and are sensible to memory contention: due to the limited cluster size, only few tasks can be isolated at the same time.

The *thread-to-cluster allocation* phase is detailed in Figure 26a on page 68. Ready tasks are sorted by memory sensitivity to ensure that the first tasks to be served will be the ones that most benefit from isolation. Each task is then allocated to the big cluster or, if one of the two thresholds is reached, to the little cluster. Due to the sorting process, the complexity of this phase is $O(n \log n)$, where n is the number of ready applications.

The policy then proceeds with the *thread-to-core allocation* phase, where we achieve *resource contention mitigation* by isolating all the memory sensitive tasks. Note that, while congestion is easily avoided in the big cluster, this is not necessarily true for the *LITTLE* one. In case of congestion in the *LITTLE* cores, only the most sensitive applications are isolated. The *core allocation* phase is illustrated in Figure 26b on page 68. Each task is assigned a CPU partition, whose size is the one reporting the highest *stakes function* score. If there are not enough resources or the selected partition equals the entire cluster, the task is not isolated. Otherwise, the partition is mapped to the real hardware and the resulting set of cores is set as a constraint to the Linux scheduler when scheduling the task. Due to the stakes function computation, the complexity of this phase is $O(n_c\ c_c)$, where n_c and c_c are the number of applications and the number of cores in the cluster c, respectively.



(a) Thread-to-cluster allocation phase.



(b) Thread-to-core allocation phase.

Figure 26: Flowchart describing the co-scheduling policy.

Experimental Results

We validated our approach on real Hardware using a ODROID-XU3 development board, which features a Samsung Exynos5422 Octa Core System-on-Chip. The board is an example of ARM big.LITTLE architecture: the big cluster features four Cortex A15 cores (2.1Ghz, 32KB L1 cache, 2MB L2 cache), while the little cluster features four Cortex A7 cores (1.5Ghz, 32KB L1 cache, 512KB L2 cache). When commenting the results, the LITTLE cores will be numbered from 0 to 3, while big cores will be numbered from 4 to 7. The board provides sensors for monitoring the CPU power consumption at cluster and memory level, and we used them to monitor power consumption during each test. Some of the tests take place on a single cluster (i.e. only big or only little); in that case, we only report power consumption for that cluster and for the memory.

Regarding applications, we used *blackscholes*, *bodytrack*, *facesim*, *ferret*, *fluidanimate*, *freqmine*, *swaptions* and *vips* from the PARSEC benchmarks suite 3.0 [108].

We implemented the co-scheduling algorithm as a user-space process that exploit the Linux Control Groups framework [109] to enforce the exclusive assignment of cores and the maximum CPU bandwidth available to the applications.

Regarding the parameters of the policy, we used a load threshold of of 1.5 threads per core, and we allowed a maximum of four memory sensitive threads to concurrently run on the accelerator. We defined a memory sensitive thread as a thread that suffers more than $\mu^M = 5\%$ degradation due to memory contention.

We characterized each application in both clusters to build their stakes functions. Then, we performed two tests: first, we ran workloads separately on the two clusters to demonstrate the benefits of *thread-to-core* allocation. Second, we run workloads on the entire processor to show also the benefits of resource-aware *thread-to-cluster* allocation. During the last test, we isolated all the processes that were not involved in the analysis on cores 0 and 4 to minimize interferences; therefore, we exploited only three cores per cluster: 1 to 3 for the LITTLE cluster, 5 to 7 for the big cluster.

APPLICATION CHARACTERIZATION We executed each application with a number of threads that ranges from 1 to 3, for a total of three configurations per application. The only exceptions are *fluidanimate*, whose number of threads is required to be a power of 2; and ferret, which exploit pipeline parallelism and therefore was analyzed in two configurations: 1 and 2 threads per stage. Note that most applications use additional threads for synchronization and output collection; with number of threads we refer only to the threads that perform the actual execution.

Table 3: Results of the application characterization: CPU demand (γ) and minimum/maximum memory sensitivity percentage (μ^m, μ^M). We report applications in red if their execution consumes more energy on the big cluster than on the little cluster.

A_t		big cluster			little cluster		
Application	threads	γ_{A_t}	μ^m	μ^M	γ_{A_t}	μ^m	μ^M
blackscholes	1	1.00	0.79	1.47	1.00	0.30	0.35
blackscholes	2	1.86	1.02	7.93	1.80	0.33	0.40
blackscholes	3	2.63	4.91	5.01	2.42	0.00	0.01
bodytrack	1	1.01	15.87	35.05	1.01	16.54	30.17
bodytrack	2	1.81	8.00	27.50	1.90	9.91	17.83
bodytrack	3	2.35	2.36	11.02	2.65	6.32	12.79
facesim	1	0.99	12.09	38.57	1.00	2.96	7.66
facesim	2	1.67	7.07	30.12	1.79	2.00	8.73
facesim	3	2.17	11.66	16.15	2.39	1.27	15.86
ferret	1	1.11	3.97	5.15	1.13	11.03	14.10
ferret	2	2.20	11.04	11.27	2.22	8.01	8.25
fluidanimate	1	1.00	4.98	10.72	1.00	2.12	2.74
fluidanimate	2	1.95	10.29	20.45	1.96	1.53	2.32
freqmine	1	1.00	7.82	23.46	1.00	4.54	11.55
freqmine	2	1.62	8.84	20.92	1.67	5.89	9.82
freqmine	3	2.05	2.87	3.01	2.03	7.03	8.00
swaptions	1	1.00	1.64	10.23	1.00	3.09	5.84
swaptions	2	2.00	9.89	15.63	1.99	1.27	2.46
swaptions	3	2.97	7.11	7.61	2.95	0.38	2.72
vips	1	0.90	8.74	24.79	0.97	1.98	4.25
vips	2	1.60	9.85	21.90	1.83	0.00	0.01
vips	3	2.04	6.90	14.01	2.58	0.00	2.88

Results are summarized in Table 3 on page 70. It is very interesting to note that during solo run, *executing applications on the big cluster is usually more energy efficient*. We reported exceptions in bold: the only applications whose execution is more energy efficient on the little cluster are the ones that: *a*) use only one thread. This result validates our observations from Table 2 on page 66, according to which the energy efficiency of the big cluster substantially improves with the number of scheduled threads; *b*) use resources from the little core in an efficiency way, e.g. use one thread and have $\gamma \sim 1.0$. This validates the basic idea underlying our scheduling policy, according to which an optimal usage of the accelerator is crucial to achieve en-

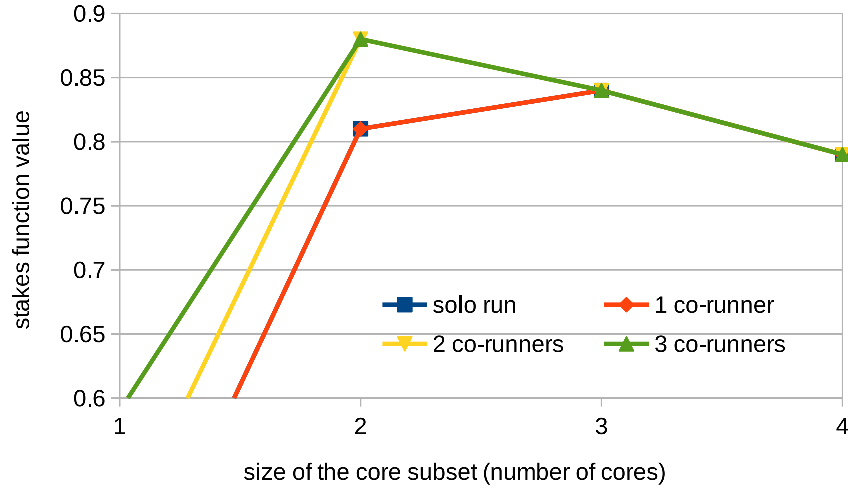
ergy efficiency; and *c*) are more memory sensitive on the big cluster than on the little cluster, meaning that an efficient memory usage is also crucial to achieve energy efficiency. This last point is not true for *ferret*, but please note that *ferret* exploits pipeline parallelism and its memory behavior is therefore different from those of the other applications.

Regarding memory intensiveness, it is worth to notice that some applications are more sensible to memory contention when running on a certain cluster (either *big* or *little*), with respect to the other. The reason behind this phenomena is that the sensitivity of an application is correlated to the number of cache misses and the entity of the cache miss penalties. It is well known how this is dependent from the processor operating frequency and the parameters of the cache hierarchy.

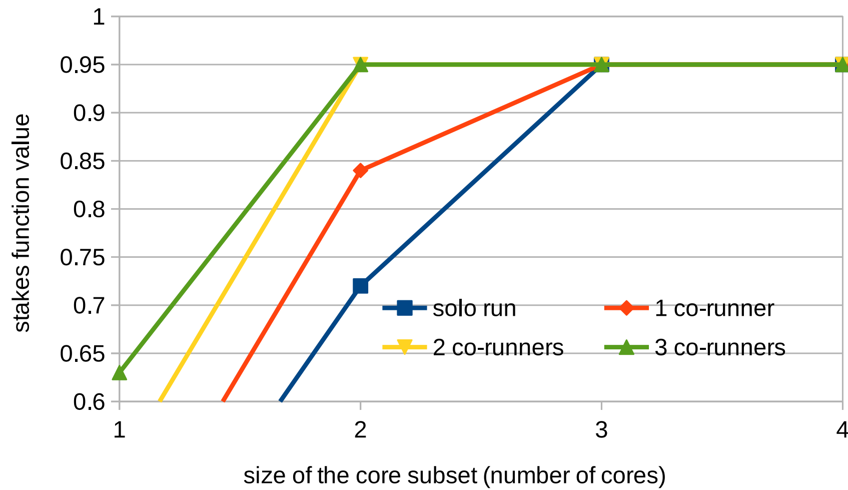
Figure 27 on page 72 shows two examples of stakes function on the *big* cluster, both in 3-threads configuration. *facesim* (Figure 27a on page 72) is *memory sensitive*: even if its *CPU demand* is 217%, therefore needing at least three cores to meet the maximum performance level, an allocation of three cores is advantageous only if there is at most one other thread running on the cluster. Otherwise, the optimal subset choice would be 2 cores. Conversely, *blackscholes* (CPU Demand 263%) is less *memory sensitive*: as shown in Figure 27b on page 72, the application can run in 4 cores even in high congestion scenarios without incurring in serious performance degradations.

CO-SCHEDULING POLICY VALIDATION Instead of testing all the possible combinations of applications, we decided to focus on a limited number of randomly created cases. This allows us to show, along with the results, all the choices taken by the policy. Moreover, to make the descriptions more understandable, we chose to start all the applications belonging to the same scenario at the same time. In this way, the starting status of the system is known. Being the applications characterized by different execution times, the dynamism of the policy is nonetheless shown because the termination of each application causes the policy to be reactivated and to modify its previous decisions.

The scenarios are summarized in Table 4 on page 73 and Table 5 on page 74. We named each scenario according to the cluster involved in the test: *big* and *LITTLE* scenarios involve *big* and *little* cores respectively, while *big-LITTLE* scenarios involve the whole device. For each scenario we list the applications belonging to the workload, along with their number of threads, from the most to the least memory sensitive. For each scenario we also report the degree of congestion induced on the device, i.e. the ratio between number of running threads and number of available cores.



(a) *facesim*, 3 threads configuration.



(b) *blackscholes*, 3 threads configuration.

Figure 27: Stakes function examples on the *big* cores, with increasing number of co-running threads. *facesim* and *blackscholes*, both in three threads configurations ($\gamma = 2.17$ and $\gamma = 2.63$ CPUs respectively), run with an increasing number of co-runners. The co-runners are instances of *swaptions* in 1 thread configuration ($\gamma = 1.00$ CPUs each).

The *big.LITTLE 3* scenario offers a good example of how the policy works: the applications involved in the test are ordered by memory sensitivity, then each application is allocated to the big cluster until the sensitivity or load thresholds are reached. In this example, being the load threshold equal to $1.5\frac{\mathrm{Threads}}{\mathrm{Core}}$, only four threads can be assigned to the big cluster. According to their stakes functions, *bodytrack* has to be isolated on one core while *vips* can run on the remaining ones. The other applications are assigned to the little cluster where, according to their stakes function, *swaptions* is isolated on one core

Table 4: Summary of the big and LITTLE test scenarios. $M_1 \xrightarrow{\alpha} M_2$ means that an application mapping is changed from M_1 to M_2 after application α has terminated.

	Description of the workload			Cores allocation	
Scenario	Application	Threads	$\frac{\text{Threads}}{\text{Cores}}$	HMP	HMP w/policy
LITTLE 1	ferret†	1	1.00	$0-3$	0
	vips	3		$0-3$	$1-3 \xrightarrow{\dagger} 0-3$
LITTLE 2	freqmine†	2	1.25	$0-3$	$0-1$
	blackscholes	3		$0-3$	$0-3 \xrightarrow{\dagger} 0-3$
LITTLE 3	bodytrack†	3	1.25	$0-3$	$0-1$
	facesim	2		$0-3$	$0-3$
LITTLE 4	facesim	3	1.50	$0-3$	$0-1 \xrightarrow{\dagger} 0-3$
	blackscholes†	3		$0-3$	$2-3$
big 1	vips	3	1.00	$4-7$	$4-5 \xrightarrow{\dagger} 4-7$
	ferret†	1		$4-7$	$6-7$
big 2	freqmine†	2	1.25	$4-7$	$4-5$
	blackscholes	3		$4-7$	$6-7 \xrightarrow{\dagger} 4-7$
big 3	facesim	2	1.25	$4-7$	$4-5$
	bodytrack	3		$4-7$	$4-7$
big 4	facesim	3	1.50	$4-7$	$4-5 \xrightarrow{\dagger} 4-7$
	blackscholes†	3		$4-7$	$4-7$

while *blackscholes is not isolated* When *bodytrack* terminates, the policy is reactivated and *vips* and *blackscholes* are assigned to the big cluster. The policy then re-allocates cores according to the stakes functions of each application, and the entire process is repeated until there are ready applications in the workload.

We show the benefits of our co-scheduling policy in Table 6 on page 75, where we report the energy consumption for all the scenarios. Given that this work is focused on energy efficiency but also on performance, we reported also the execution time, along with its standard deviation. Finally, we report the Energy-Delay product (EDP) for all the scenarios. EDP is a well known metric that rewards the configurations that bring benefits in terms of both execution time (delay) and energy, and equals the product of energy and execution time.

Our policy *does not degrade* the performance of the Linux HMP scheduler. Even workloads that exhibit a small number of threads benefit from this support. For instance, in *big 2* scenario the policy isolates the two applications into separate core sets. Indeed, according to their stakes functions, a not negligible degradation would occur in case of co-scheduling. Even if there is plenty of CPU bandwidth for each thread to execute, isolating the two applications leads to performance and energy efficiency improvements.

Moreover, our policy shrinks the list of processors where each thread can be scheduled, thus making the scheduling process more deterministic; therefore, the execution times present a lower standard deviation with respect to the HMP scheduler alone.

The results validate the proposed approach, showing that the *big* processor can be exploited to accelerate concurrent applications with good speed-ups in terms of both performance and energy efficiency. Imposing few constraints to the Linux HMP scheduler, the policy achieves up to 12.88% performance speed-up, 13.65% energy speed-up and 28.29% EDP speed-up with respect to the standard Linux HMP scheduler.

Conclusions

In this Section, we introduced the concept of *stakes function*, which represents the trade-off between *exclusive allocation* and *sharing of resources* in multi-core processors. We introduced a co-scheduling policy that exploits *stakes functions* as a metric to take co-scheduling decisions on heterogeneous processors.

We validated the policy on an octa-core big.LITTLE processor, achieving up to 12.88% performance, 13.65% energy and 28.29% EDP speed-up with respect to the standard Linux HMP scheduler.

Our approach has still open limitations: *a*) it relies on a design time characterization, and therefore can not manage unknown applications or data dependent performance variability; *b*) stakes functions address the worst case scenario and are therefore conservative; and *c*) the usage of the big cluster could be farther optimized by allowing some applications from the little cluster to partially run on the big cluster while they wait for their turn to be accelerated.

While the last limitation can be addressed by implementing a more refined policy, the first two limitations could be addressed by employing online learning techniques to compute CPU bandwidth and memory sensitivity of applications dynamically during runtime.

A HETEROGENEITY-AWARE OPENCL SUPPORT

Heterogeneous System Architectures (HSAs) [110] are nowadays an attractive solution to exploit the trade-off between performance and

The mechanism described in this subsection is part of the work published in [33], which has a broader scope. Here, we will only focus on our contribution.

energy efficiency. Those architectures feature different kind of resources, such as Central Processing Units (CPUs)—possibly integrating heterogeneous cores—Graphic Processing Units (GPUs), Digital Signal Processors (DSPs) and other kinds of hardware accelerators. Examples of HSAs are the Samsung Exynos 5 Octa [111], which hosts an ARM big.LITTLE asymmetric octa-core CPU and a Mali GPU; and the Xilinx Zynq [112], which features an ARM dual-core CPU and a reconfigurable Field Programmable Gate Array (FPGA) unit.

The increase of heterogeneity comes at the cost of programmability: exploiting multiple kinds of processing elements implies dealing with different type of programming languages and models, and this introduces new challenges in implementation and integration. Moreover, this abundance of resources has to be properly managed and allocated, since each type of processing unit delivers a different level of performance/power efficiency to each application.

In 2009, the Khronos Group, which includes Apple, ARM, Samsung and many other industrial partners, has defined OpenCL [82], a cross-platform programming model that leverages the Single Instruction Multiple Thread (SIMT) computational paradigm in order to exploit the data parallelism capabilities of heterogeneous accelerators.

OpenCL is implemented as an extension of the C/C++ language and allows application developers to program and use a large variety of processing units using a single programming model. However, although it provides functional portability between different processing units, the OpenCL API still requires the application developers to explicitly select and configure the resources that will be used to execute applications.

Especially when executing on CPUs, OpenCL applications can gain performance advantages by carefully choosing which computational units will be used. In oder to allow that, the OpenCL 1.2 specification introduced the concept of Device Fission [113]. Basically, Device Fission allows application developers to partition an OpenCL device (i.e., a multi-core CPU, a GPU or a hardware accelerator) into multiple sub-devices. On Intel CPUs, moreover, the partition can be performed by name. That is, developers can manually select which processing elements will be part of the same partition.

Indeed, the Device Fissure feature is especially useful on CPUs, e.g. to deal with cache contention or to minimize memory overheads in NUMA systems. However, this feature could also be handy in big.LITTLE architectures: in this case, applications would indeed be able to execute OpenCL kernels on a wide variety of devices: GPU, big cluster, little cluster or on a mix of big and little processing elements. However, this goal poses multiple challenges: *a*) big.LITTLE architectures feature ARM cores, which, at the time of writing, do not support named partitions; and *b*) as already mentioned in the previous sections, in order to address multi-application scenarios, the

Due to the lack of named partitions support, it is not possible, say, to select all the big cores. It is possible to create partitions, but it is up to the OpenCL runtime to chose which processing elements will be selected.

resource manager should be in charge of actually choosing (and configuring, e.g. using DVFS) which subset of processing elements will be allocated to each application. In this scenario, named partitions would be useless unless the applications source code is changed in order to retrieve the IDs of the allocated cores from the resource manager before actually requesting the device fissure.

Figure 28 summarizes our approach to OpenCL sub-device allocation. The lower layer represents the big.LITTLE processor, which consists in two heterogeneous clusters of Processing Elements (PEs). The abstraction performed by the Operating System, however, exposes a single cluster of processing elements, and this is exactly what the OpenCL runtime detects when not managed by the resource manager. In managed scenarios, conversely, the resource manager uses the Linux Control Groups to enforce a specific system view on the application (and hence, on the OpenCL runtime). This means that managed applications that exploit the OpenCL runtime to retrieve

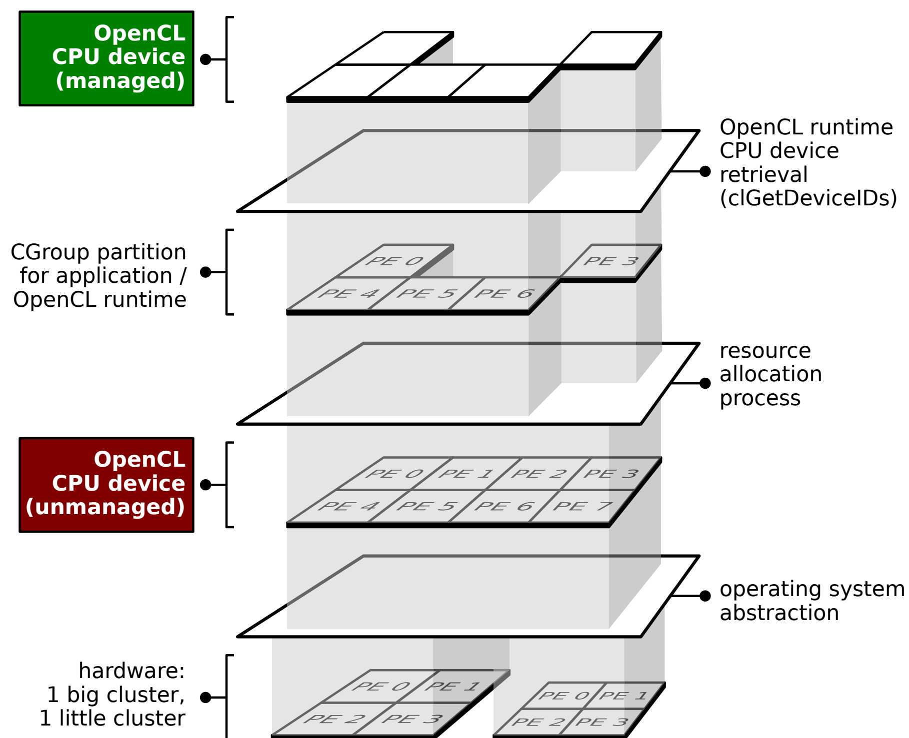


Figure 28: Enforcing a custom system view on the OpenCL runtime. The lower layer represents the big.LITTLE processor, which consists in two heterogeneous clusters of Processing Elements (PEs). The abstraction performed by the Operating System, however, exposes a single cluster of processing elements, and this is exactly what the OpenCL runtime detects when not managed by the resource manager. In managed scenarios, conversely, the resource manager uses the Linux Control Groups to enforce a specific system view on the OpenCL runtime, thus tricking it into detecting only the selected processing elements.

the available platform devices will see a single CPU device that is composed by only the processing elements that have been allocated to the application.

Although the approach presented in this Section is extremely simple and is not based on novel tools and frameworks, it is based on a clever insight: given that the Linux Control Groups framework enforces a custom system view on applications, it can be also used to trick the OpenCL runtime into detecting only the processing elements that the resource manager allocated to an application. Our novel contribution can be therefore summarized as follows:

- we enabled a CPU device partitioning process that is akin to device fissure but is performed at resource manager level instead of at application level, hence enabling OpenCL multi-application managed scenarios;
- we provided the resource manager with the ability of performing named device partitions (i.e., OpenCL sub-devices that contain only the selected processing elements) on any type of systems, including ARM-based ones.

Table 5: Summary of the big.LITTLE test scenarios. $M_1 \xrightarrow{\alpha} M_2$ means that an application mapping is changed from M_1 to M_2 after application α has terminated.

	Description of the workload			Cores allocation	
Scenario	Application	Threads	$\frac{\text{Threads}}{\text{Cores}}$	HMP	HMP w/policy
big.LITTLE 1	blackscholes	2	1.00	$0-3,4-7$	$1-3 \xrightarrow{\dagger} 5-7$
	swaptions	1		$0-3,4-7$	5
	facesim	3		$0-3,4-7$	$5-7$
big.LITTLE 2	vips	3	1.17	$0-3,4-7$	$5-6 \xrightarrow{\dagger} 5-7$
	swaptions	1		$0-3,4-7$	7
	blackscholes	3		$0-3,4-7$	$1-3$
big.LITTLE 3	freqmine †	2	1.33	$0-3,4-7$	$5-6$
	ferret ‡	2		$0-3,4-7$	$5-7 \xrightarrow{\dagger} 5-6$
	swaptions	1		$0-3,4-7$	$1 \xrightarrow{\dagger} 7$
	blackscholes	3		$0-3,4-7$	$1-3 \xrightarrow{\ddagger} 5-7$
big.LITTLE 4	freqmine †	1	1.17	$0-3,4-7$	5
	fluidanimate ‡	2		$0-3,4-7$	$6-7 \xrightarrow{\dagger} 5-6$
	facesim	3		$0-3,4-7$	$1-2 \xrightarrow{\ddagger} 5-7$
	ferret	1		$0-3,4-7$	$3 \xrightarrow{\dagger} 7$
big.LITTLE 5	facesim †	3	1.66	$0-3,4-7$	$5-7$
	ferret	1		$0-3,4-7$	$5-7 \xrightarrow{\dagger} 5-6$
	swaptions	3		$0-3,4-7$	$1-3 \xrightarrow{\dagger} 5-7$
	freqmine	3		$0-3,4-7$	$1-3$
big.LITTLE 6	bodytrack †	1	1.83	$0-3,4-7$	5
	vips	3		$0-3,4-7$	$6-7 \xrightarrow{\dagger} 5-6 \xrightarrow{\ddagger} 5-7$
	blackscholes ‡	1		$0-3,4-7$	$1-3 \xrightarrow{\dagger} 7$
	swaptions	3		$0-3,4-7$	$1-3$
	freqmine	3		$0-3,4-7$	$1-3$

Table 6: Experimental results: for each scenario, we present execution time [s] and Energy [J]. For execution time, we also present the standard deviation (25 runs). The last three columns present the achieved speed-up in terms of time, energy and Energy-Delay.

	Execution time				Energy		Speed-up %		
	HMP	std dev	w/policy	std dev	HMP	w/policy	Time	Energy	EDP
LITTLE 1	420.19	7.07	413.18	3.59	306.74	299.55	1.70	2.40	4.14
LITTLE 2	73.48	0.34	72.68	0.35	53.27	52.69	1.10	1.10	2.20
LITTLE 3	80.35	1.06	75.54	0.62	43.53	44.72	6.37	1.82	8.31
LITTLE 4	40.60	0.44	40.02	0.13	25.64	24.99	1.45	2.64	4.12
big 1	226.00	7.55	218.55	2.37	1107.4	1066.55	3.41	3.83	7.37
big 2	42.31	1.86	41.15	0.18	223.82	213.30	2.82	4.93	7.89
big 3	37.74	1.07	35.88	0.65	162.02	156.32	5.17	3.64	9.00
big 4	26.16	0.47	24.37	0.28	138.36	127.97	7.31	8.12	16.02
big.LITTLE 1	106.84	1.93	101.98	1.98	446.58	428.32	4.76	4.26	9.23
big.LITTLE 2	249.35	4.03	233.02	2.64	1241.76	1183.74	7.01	4.90	12.25
big.LITTLE 3	118.15	1.77	104.66	0.51	473.77	416.88	12.88	13.65	28.29
big.LITTLE 4	71.66	0.67	65.40	0.15	375.05	348.04	9.58	7.76	18.08
big.LITTLE 5	66.57	1.10	64.44	0.41	342.15	327.36	3.30	4.52	7.96
big.LITTLE 6	250.51	6.11	229.00	1.87	1227.51	1140.43	9.39	7.64	17.74

Part II

MULTIPLE-COMPUTING-NODES SYSTEMS

The second part of this dissertation deals with systems composed by *multiple computing nodes*. In particular, we will focus on HPC systems.

With the term HPC we refer to the practice of using a massive degree of computational parallelism in order to accelerate the execution of applications. The typical users of HPC systems are scientific researchers, academic institutions and some government agencies, e.g. the military.

Given the introduction of HPC-oriented cloud platforms such as the Amazon EC2 Cluster Compute Instances (CCIs) [114], cloud systems are becoming increasingly attractive (but not yet fully comparable to in-house solutions) even in the case of tightly-coupled applications such as MPI programs [115, 116].

As can be guessed by most readers, one of the biggest problems that must be tackled by HPC resource management falls in the area of RAS, which stand for "Reliability, Availability and Serviceability".

Detection and tolerance of faults, however, are not the only problem that must be tackled by resource managers. The costs of cooling and supplying are today a non-negligible part of the costs associated to HPC systems, cloud and data centers [117, 15]. In HPC scenarios, it is therefore paramount to minimize power consumption, which directly affects the cost of power supplying, and to avoid thermal hot-spots, which have negative effects on the life expectancy of the chips—by both inducing hardware faults and accelerating the chip aging process—and on the cooling costs.

First of all, we perform and interesting study on how the freeze/restore-based process migration of MPI applications, which is usually performed at node granularity to address faults, can be made fine-grained in order to migrate only parts of the application on a different computing node. This allows resource managers to perform optimizations such as load balancing, resource consolidation, or also to counteract the effects induced on the hardware by aging (e.g. by avoiding to use faulty cores). Then, we present a resource management approach that exploits the trade-off between power consumption and performance when executing HPC applications that must comply with runtime-variable Quality of Service requirements. Finally, we perform the first steps towards an unified runtime management support for deeply heterogeneous HPC systems.

5 ENABLING A TRANSPARENT PROCESS MIGRATION IN OPEN MPI

The contents of this subsection are partially published in [118].

Migrating a process among cores of the same processor does not pose any specific issue. When going distributed, however, migration becomes a complex operation. First of all, all the processes that are communicating with the migrating one must be temporarily blocked. Second, the process data, which resides in the memory and can possibly be huge, must be moved from the target to the destination computing node.

Before dealing with allocation policies, we therefore chose to focus on inter-node process migration. In particular, we want to provide a mechanism that allows processes to be transparently migrated by a resource manager in a totally transparent fashion.

We chose to address MPI applications because MPI is the most widespread distributed programming model.

This chapter introduces `mig`, a framework that enables the transparent migration of MPI applications—or even just a subset of their processes—among different nodes of a distributed HPC system. This framework, which is implemented as a module of Open MPI, provides mechanisms that can be exploited by resource managers to react to hardware faults, to counteract performance variability, to improve resource utilization or to perform a fine-grained load balancing and power/thermal management.

Conversely to the other state-of-the-art approaches, `mig` does not require changes in the applications source code. Moreover, it is highly maintainable, since its implementation required very few changes in the already existing Open MPI modules.

MOTIVATION

Given the wild evolution of High-Performance Computing (HPC) and silicon technology, modern and future parallel systems must deal with an increasing number of computing nodes and, also due to the end of Dennard's scaling (see Section 1.1), with the subsequent power-related issues. These two aspects are posing new challenges in terms of performance scaling, efficient utilization of the nodes, power and thermal management, reliability and fault-tolerance.

As the Mean Time Between Failures (MTBF) of current supercomputing systems is already way below 100 hours [119, 120, 121], relying on fault-tolerance techniques is nowadays paramount. In this regard, a very common approach is to employ *Checkpoint/Restart (C/R)* approaches: the execution state of a managed application is periodically saved (check-pointed) so that, in case of faults, it can be resumed

(restarted) from the last consistent state. An alternative use of C/R is to migrate the execution of applications from a faulty to a reliable set of nodes. This is a versatile technique: a system can benefit from task migration support not only to react to faults, but also for resource management purposes, e.g. to perform load balancing or to relieve an overheated node.

Concerning nodes utilization, the efficient exploitation of an extremely parallel HPC system usually relies on programming models, e.g. MPI, that allow applications to seamlessly execute in a distributed environment. As already discussed in the previous chapters, in order to enable multi-applications scenarios and to perform an effective system-wide management, the aforementioned programming models may in turn rely on entities—e.g., resource managers or job schedulers—that are in charge of driving the task placement over the wide set of available computing resources.

Our framework allows resource managers to dynamically change the set of computing nodes that is allocated to a running MPI application. This kind of support enables a wide range of possibilities, such as reacting to faults, adapting the resource assignment to the time-varying performance requirements of applications, or performing system-level load balancing in order to evenly spread heat and power consumption among the available resources.

We implemented a system-level migration schema based on the idea of partitioning the MPI application processes that are running on the same node into multiple migratable entities. This schema allows us to perform transparent fine-grained migrations, i.e., to migrate a part of an MPI application onto a different node while the remaining processes are still running.

RELATED WORKS

The state-of-art literature already offers some examples of Checkpoint/Restart [120]. Most of the C/R implementations rely on the *Berkley Lab's Checkpoint/Restart* kernel-space tool (BLCR) [122] and the *libckpt* user-space library [123].

C/R based approaches usually adopt the following schema:

1. the processes that belong to an application are forced to reach a global consistent state;
2. the application execution state is check-pointed;
3. the application is resumed;
4. after some time, if no faults are detected, return to point 1.

In case of faults, all the running processes are killed and the application execution is resumed from the last checkpoint.

C/R mechanisms can be managed either at *application* or at *system-level*. In the former case, also known as user-level, the application itself is in charge of synchronizing the execution of its own processes and performing the checkpoint. This is typically done by calling suitable library functions. In the latter case, instead, C/R is accomplished by the run-time system that controls the application life-cycle, e.g. the resource manager or the programming model runtime.

Hursey *et al.* [124] extend the Open MPI stack with additional layers that provide C/R capabilities in a network-agnostic fashion. The application processes can be stopped and then restarted on a different set of nodes that is potentially characterized by a different network topology. This solution introduces notable code dependencies between internal Open MPI modules. Moreover, it induces significant overheads, since it copies the process state images on an external storage server, which therefore becomes the real bottleneck for the system performance. This drawback, along with the poor maintainability of the software, led the Open MPI developers to disable these additional layers since Open MPI version 1.7.

In this work, we used OpenMPI version 1.10.

As already noted for the work presented by Hursey *et al.*, the common limitation of C/R-based approaches is the overhead introduced by performing periodical checkpoints. In some use cases, this overhead impacts dramatically, even doubling the execution time of applications [121]. Moreover, the overhead increases exponentially with the system size, i.e. with the number of computing nodes. Considering a large HPC system with thousands of nodes and not negligible power supply costs, the overhead must be carefully evaluated not only in terms of performance loss, but also in terms of additional energy consumption [125].

In order to minimize the overheads induced by periodical checkpointing, some authors propose to employ *task migration* techniques. The main idea behind these approaches is that, provided that some entity is able to predict the imminent fault of a computing node, all the processes that are running on that node can be pro-actively check-pointed, migrated on a healthy node and there restored. This approaches are quite advantageous, inasmuch as that they does not require applications to be periodically check-pointed.

Task migration techniques can be classified on a granularity basis: migration can be performed either at *Virtual Machine*, *container* or *process-level*.

The first two classes are very common, since an easy workload management and the guarantee of isolation is very appealing in the case of MPI applications. However, it is worth remarking that the lack of shared memory communication between processes on different virtual machines heavily impacts on the performance of applications, and this problem is especially exacerbated in case of I/O intensive workloads [126]. Although many authors proposed approaches that

tackle this problem, the gap between native and and hosted (VM-based) execution is still wide, and it leads to latency increments up to 16x for communication intensive operations [127].

The third class of techniques, i.e. process-level migration, is instead preferable when the main goal is to optimize resource usage: process-level resource allocation is way more flexible than that of containers or Virtual Machines. Concerning this class of techniques, the most promising solution has been proposed by Wang *et al.* [119]. The basic idea of the authors is to try to minimize the number of performed checkpoints by using the aforementioned proactive approach: each computing node is constantly monitored and, in case of imminent fault, all the processes that are executing on that node are migrated on a healthy one. As usual in the C/R approaches, the proposed framework, which is implemented in LAM/MPI (predecessor of Open MPI), requires all the processes pertaining to the same application to synchronize in order for the checkpoint to be performed.

According to our literature review, we chose to focus on the following goals:

MAINTAINABILITY
: The migration framework must be self-contained.

TRANSPARENCY
: The migration support must not require the source code of applications to be changed.

FINE GRANULARITY
: It must be possible to migrate just a part of the application, i.e. a subset of its processes. This allows resource managers to perform load balancing or also to isolate only a subset of a faulty node from the rest of the system.

ASYNCHRONISM
: Processes that are not going to be migrated must be allowed to continue executing.

INTEGRABILITY
: The migration framework must provide APIs that allow resource managers to drive the migration of applications.

The solution we propose addresses all the aforementioned issues:

- it is a process-level migration mechanism whose granularity can be tuned by the resource manager;
- it does not require any change to the applications code;
- the migration is almost completely transparent with respect to the application execution;

- migration can be triggered by a resource manager through a suitable API.

We implemented the proposed task migration approach as a non-invasive Open MPI module that we called `mig`.

DESIGN AND IMPLEMENTATION

As already mentioned in the previous subsections, we designed `mig` as a mechanism that allows resource managers to trigger migrations. Hence, this work will not deal with with scheduling policies; instead, we will focus on how to suitably carry out the migration itself.

The main idea behind our approach is that the resource manager can signal a migration request to the Open MPI runtime by sending a `(source node, destination node)` pair via the already existent socket channel. Then, the Open MPI runtime (in particular the `mig` module) performs the sequence of actions that are needed to actually migrate the application processes.

In order to perform C/R on the selected processes, we employ the Checkpoint/Restore In Userspace (CRIU) tool [128], which is a C/R tool that is gaining a lot of interest in virtualization environments [129]. The big advantage of CRIU is that it exploits mechanisms that have already been integrated in the Linux kernel. CRIU does not require additional kernel modules to be loaded, and it can be used in user-space.

Open MPI architecture

Open MPI is a popular implementation of MPI. Its structure, which is based on the Modular Component Architecture (MCA) [130], is composed by three kinds of entity:

- **MCA**: the Modular Component Architecture backbone, which is in charge of instantiating all the Open MPI modules and initializing them according to the run-time parameters;
- **Modules**: the main functional parts of Open MPI. Each module is devoted to a specific task, e.g., managing the processes life cycle or forwarding input/output. Please note that, according to the Open MPI jargon, modules are referred to as *frameworks*; since this term is already used multiple times in this dissertation, we chose to avoid ambiguity and call them “modules”;
- **Components**: a specific implementation of a module. For instance, depending on which component is loaded at runtime, communication can be based on different protocols (e.g., TCP or Infiniband).

In order to logically separate the different functional areas, the modules are bundled into three layers:

- **OpenMPI (OMPI)**: modules that expose the application-level API;
- **Open Run-Time Environment (ORTE)**: modules that manage the processes life-cycle and orchestrates the inter-node communication;
- **Open Portable Access Layer (OPAL)**: a library that provides OMPI and ORTE with a set of utility modules such as event manager and memory allocator.

As shown in Figure 29, the structure underlying an MPI application consists of an application-level ORTE layer, which allocates resources to the application processes according to the resource manager directives and orchestrates the communication from/to other nodes; and multiple process-level OMPI layers, which expose the MPI API to each process and manage communication among the processes that are local to the node. In case of multi-node execution, each computing node is characterized by the same structure: each node features an application-level ORTE layer and multiple process-level OMPI layers.

The command that is used to launch an MPI application is called `mpirun`. The node from which the application is started is defined as the *Head Node Process* (HNP) and manages the entire application execution.

Open MPI extension

Most of the changes involved in our extension are contained in the new `mig` module; however, in order to allow `mig` to suitably orches-

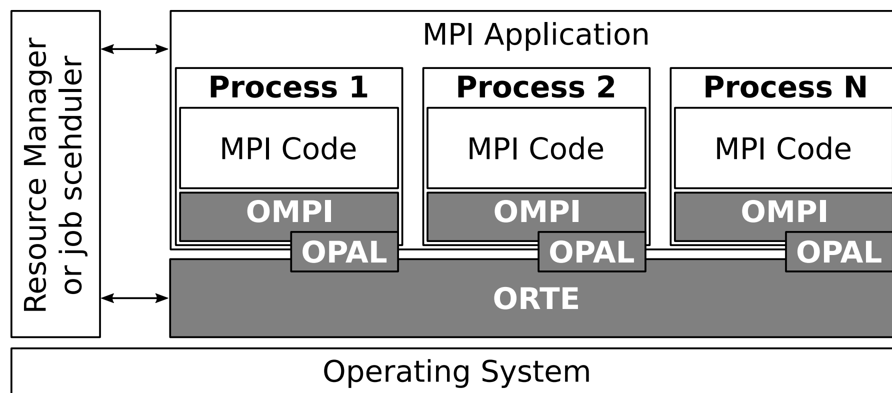


Figure 29: Architecture of Open MPI modules. OMPI exposes the MPI APIs to the processes, ORTE controls the processes life cycle, while OPAL acts as an utility library.

trate migrations, we also had to perform minor additions to the existing Open MPI modules. The modules involved in our Open MPI extension are:

- `ras` (part of ORTE): it provides the communication channel between the Open MPI runtime and the resource manager. Currently, Open MPI uses this module only during the application initialization, and it does so to retrieve the full list of available nodes. We extended the `ras` API to allow the resource manager to send migration requests during the applications execution and to be notified about the status of the requests.

- `oob` (part of ORTE): the "out-of-band" framework provides a low-level API for the communication between the ORTE daemons. This module contributes to the migration process by managing the opening and closure of the pending connections towards the migrating ORTE daemon instance.

- `plm` (part of ORTE): high-level communication between HNP and the ORTE daemons. We implemented the protocol necessary to coordinate the orted instances. We also added the `ssh` call that spawns the `orted-restore` daemon on the destination node. This daemon is in charge of resuming the processes execution once the checkpoint image transfer is completed.

- `btl` (part of OMPI): this is the application-level peer-to-peer communication module. We modified the *TCP* component to manage the opening/closure of the TCP socket connections among migrating application processes.

- `mig` (part of ORTE): this is the module we implemented to enable the migration mechanisms. The provided functionalities are controlled by `ras` on behalf of the resource manager. `mig` is in charge of coordinating the migration phases via the `plm` module by routing commands to the ORTE daemon instances that are involved in the migration. `mig` is also responsible of performing checkpoint/restore and of sending the process status image to the destination node.

As already mentioned, at computing node level, processes from the same application are usually managed by a single ORTE daemon. In order to enable a fine-grained migration support, we instead force Open MPI to instantiate a tunable number of orted instances, so that one or more of them—along with the processes that they are managing—can be migrated on a different node (See Figure 30). That is, migration happens at orted-granularity.

The number of ORTE daemons that are instantiated on a single computing node can be selected at runtime by the resource manager. The reason is simple: whereas processes that are managed by

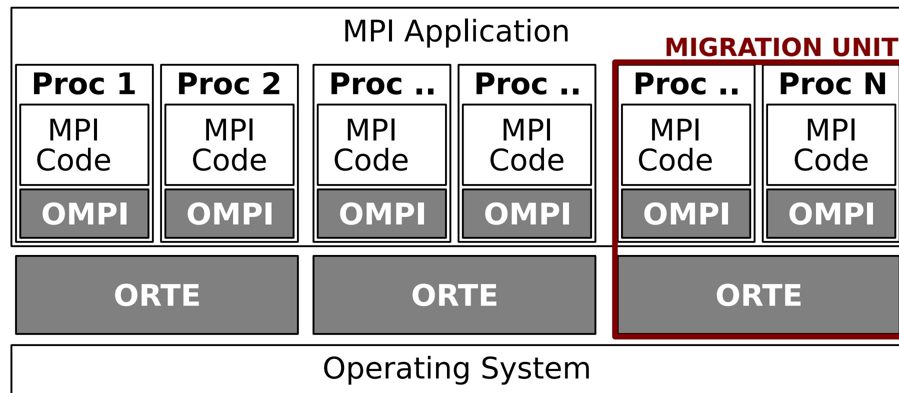


Figure 30: Open MPI modules architecture used in the `mig` approach. On each node, processes are grouped into migratable units. Each unit is managed by an ORTE daemon, as if it was running on a different node.

the same daemon communicate via shared memory, processes that are running on the same node but are managed by different ORTE instances are forced to communicate using sockets; therefore, especially in case of communication intensive applications, employing multiple ORTE daemons per node is bound to induce performance degradation. This issue can be mitigated by allowing the resource manager to trade off migration granularity (i.e., number of ORTE daemons per node) with performance.

It is worth remarking that this issue could be also solved by allowing ORTE daemons from the same node to communicate using shared memory. Since this would further augment the amount of changes to the Open MPI modules, we chose to postpone this approach to a future work. Indeed, in the context of this work, we will study the overheads that are induced by the presence of multiple ORTE daemons per node.

CRIU

The CRIU library, which is used in the CRIU C/R component of the `mig` module, is in charge of performing checkpoint/restart of a single ORTE daemon and its processes.

The checkpoint stage, which is called *dump*, freezes the processes execution and creates a collection of binary files that contain the processes state. This collection is composed by three kinds of files: *inventory*, *image*, and *auxiliary*. CRIU uses the *inventory* and *auxiliary* files in order to store the meta-data that is needed to perform the restore. The *image* files, instead, contain the memory dump of the processes and all the OS-level information (e.g., file descriptors, file-system mount-points, signal masks and ghost files).

The restart stage, which is called *restore*, reads the binary files previously generated by the *dump* stage and restarts the frozen processes. It is worth remarking that, if the restore operation does not occur on the node where the processes were dumped (i.e., in case of migration), program executables, libraries and data files must be present and identical in the destination node. Moreover, remote file-systems must be mounted, and the process identification numbers (PIDs) must be available, since the processes cannot change their PIDs after the restore. In order to guarantee the PIDs availability, we exploited the *Linux Namespaces*, a feature of the Linux Kernel [131] that allows us to isolate a set of processes in a detached environment via the `unshare` system call. In order to do this, the `orted-restore` executes the following C call:

```
unshare(CLONE_NEWNS | CLONE_NEWPID)
```

The `CLONE_NEWNS` and `CLONE_NEWPID` flags respectively detach the mount and the PID namespaces. In particular, the `orted-restore` daemon becomes the `init` process (PID=1) of the new empty PID namespace and can restart the application processes with the original PIDs. The isolated mount namespace is needed to remount the `/proc` directory and match the new process identifier configuration. At this point, the ORTE daemon instance and its children can be safely restarted.

Migration phases

As shown in Figure 31 on page 93, the migration procedure consists in five phases:

1 – COORDINATION

The `mig` module of the Head Node Process (HNP) spawns an `orted-restore` daemon on the destination node. Then, via `plm`, it notifies to all the running ORTE deamons that a migration has been triggered. The `btl` TCP component of the processes that are not involved in the migration terminates all the ongoing data transmission towards the migrating processes. Until the end of the migration, all the future data transmissions towards the migrating processes will be cached. At this point, the processes send back an acknowledgment to their own ORTE daemon instances, ensuring that no further transmissions towards the migrating processes will be performed. The ORTE daemons forward the acknowledgment to the HNP.

Indeed, processes that are not migrating can continue executing as normal. However, they will be stopped if they try to communicate with the migrating processes.

2 – CRIU DUMP

The HNP issues the `MIGRATION_EXEC` command, hence starting the migration procedure. When an application process receives that command, it waits until all the in-flight packets have been received

by the destination side. Then, all the TCP connections towards the processes involved in the migration can be safely closed, and an acknowledgment is sent back to the ORTE daemon. When the migrating ORTE daemon receives the acknowledgment, it uses the CRIU API to perform the checkpoint of its execution status.

3 – PROCESS STATE MIGRATION
The outcome of the CRIU checkpoint (or dump) is a collection of files. To simplify the transfer of such files over the network, `mig` bundles them in an archive. Optionally, the archive can be compressed. For the sake of brevity, we refer to the archive with the term "image". The image is now ready to be moved to the destination node.

4 – CRIU RESTORE
After having received – and optionally decompressed – the image, the `orted-restore` daemon on the destination node uses the CRIU API to restart the ORTE daemon and its children processes. Then, the ORTE daemon reopens the connection to the HNP and sends the `MIGRATION_DONE` message. The HNP broadcasts this message to all the other ORTE daemons.

5 – FINALIZATION
The migrated processes reopen the connections towards all the other ones and resume the execution.

Evaluation

In this subsection, we evaluate the overheads introduced by our migration mechanism. We distinguished between two types of overhead: 1) performance loss due to the execution of multiple ORTE daemons on the same computing node; and 2) time required to actually perform a migration.

Before presenting the results, let us describe our hardware and software setup.

The hardware consisted in two computing nodes. Each of those was equipped with two Intel Xeon E5-2640 octa-core hyper-threaded CPUs in a NUMA configuration (256GB of total memory). As common in HPC environments, we disabled Hyper-Threading; hence, each computing node featured a total of 16 processing elements. The operating system was CentOS 6.7, Linux kernel version 3.18.

Regarding applications, we chose to employ the NAS Parallel Benchmarks suite (NPB) [132]. In particular, we selected `IS` and `MG`, which are kernels; and `BT`, `SP` and `LU`, which are pseudo-applications. As shown in Table 7, each application can be executed using different datasets. We chose to totally exclude type `A` datasets, since they led to very short execution times. Similarly, for the pseudo-applications, we

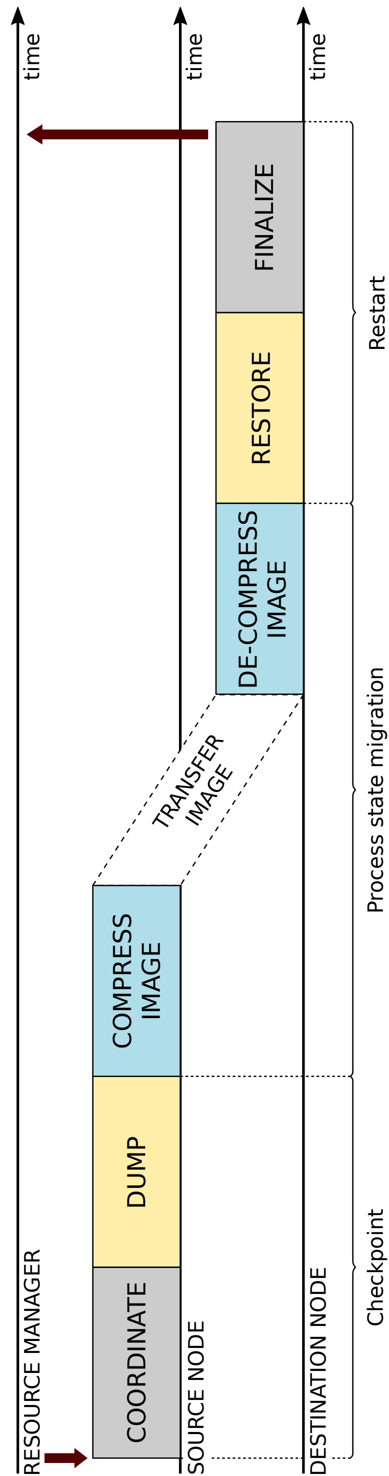


Figure 31: The mig framework: migration phases. When the resource manager triggers a migration, the application enters a coordination stage. Then, the migrating processes are dumped and the image is transferred to the destination node. There, the processes are restored, and, after a finalization phase, they can resume their execution.

Table 7: Problem data sizes (MB) for the A, B, C, and D classes of each benchmark. Application type can be either kernel (k) or pseudo-application (p-a). Green numbers indicate that we used the configuration (application - dataset) in our experiments.

		Dataset Type			
Type	Name	A	B	C	D
k	IS	64	256	1024	16793
k	MG	128	128	1024	8294
p-a	BT	2	8	32	518
p-a	LU	2	8	32	518
p-a	SP	2	8	32	518

excluded type `D` datasets, which led to prohibitively long execution times. Hence, we used datasets of type `B`, `C`, and `D` for the kernels; and datasets of type `B` and `C` for the pseudo-applications.

Overheads due to multiple ORTE daemons per node

As already mentioned, given that Open MPI employs only one ORTE deamon per node, communication between ORTE daemons is performed using TPC/IP, which is less performing than shared memory [133]. Indeed, this may become a bottleneck when multiple ORTE daemons run on a single computing node. To evaluate the overhead introduced by splitting the control of the MPI processes among multiple ORTE daemon instances, we used one of the previously described 16-core computing nodes.

We executed each benchmark using a number of ORTE daemons in $1, 2, 4, 8, 16$. Given that the benchmarks used 16 processes—i.e., one process per core—each ORTE daemon respectively managed a number of processes in $16, 8, 4, 2, 1$. It is worth noticing that using only one ORTE daemon means executing the benchmarks in the standard Open MPI scenario; hence, we took that case as a golden model.

We measured the execution time of each tuple <*benchmark, class, granularity*>, starting after the `MPI_Init` call and stopping before the `MPI_Finalize` call. We repeated the test 20 times to obtain a significant statistics; however, it turned out that the experienced standard deviation was always within 1% of the total execution time. Therefore, for the sake of simplicity, we will not report the standard deviation values.

MPI init and finalize belong to the MPI API and must be respectively called at the beginning and at the end of any MPI processing.

Figures 32, 33 and 34 provide a visual representation of the overheads for type B, C and D datasets, respectively. It is clear that the overhead increases sub-linearly with respect to the number of ORTE

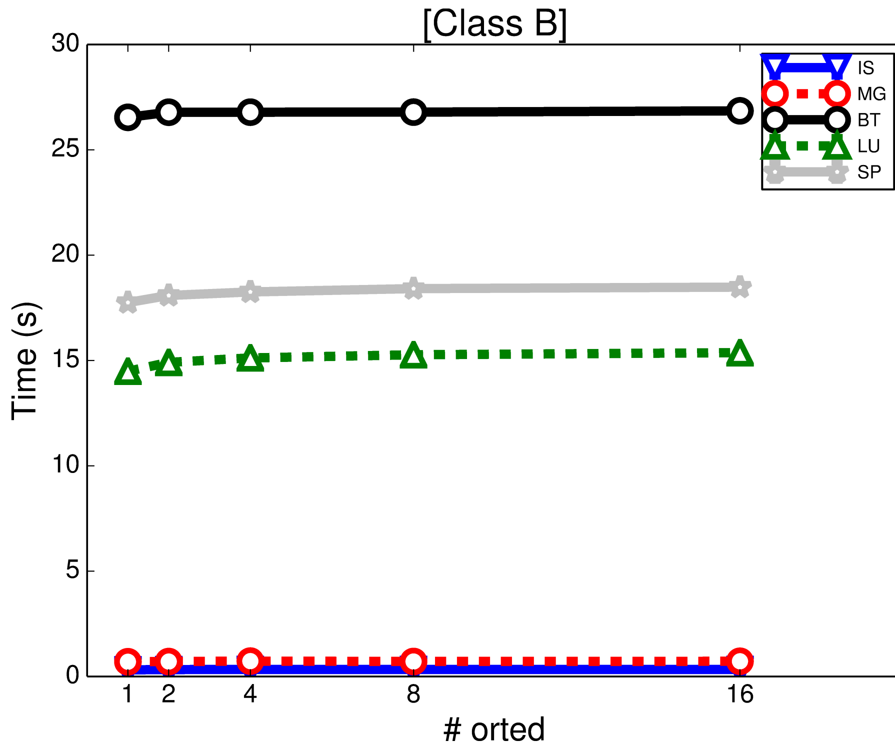


Figure 32: Execution time of each benchmark (input class = B) when running 16 processes using 1 to 16 ORTE daemons.

daemon instances, while it decreases as the problem size increases. The sub-linear increase is imputable to the fact that, once there are at least two ORTE daemons, the TCP/IP communication between MPI processes of different daemons becomes the bottleneck for communication latencies; therefore, adding more ORTE daemons does not tend to further degrade performance. The decrease of the overhead in case of increasing problem sizes, conversely, is due to the fact that increasing problem size means spending more time on computing data; therefore, the time spent in communication—which is where the overhead applies—decreases in percentage.

Tables 8 and 9 summarize the overheads for kernels and pseudo-applications, respectively. The ORTE daemons granularity poorly affects the application execution time. Considering all the test cases, the percentage of time loss is always in the $0-6\%$ range; however, as already mentioned, the overheads decrease as the dataset size increases. Considering realistic scenarios, i.e. HPC applications with large datasets, the overhead is always lower than 2%.

Finally, it is also worth remarking that, in multi-node scenarios, there are *always* at least two ORTE daemons that are forced to communicate via TCP/IP; therefore, the above mentioned overhead applies only in single-node scenarios, which are not frequent in HPC environments. This also means that migration granularity affects only the time required to perform migrations. For example, migrating 8 pro-

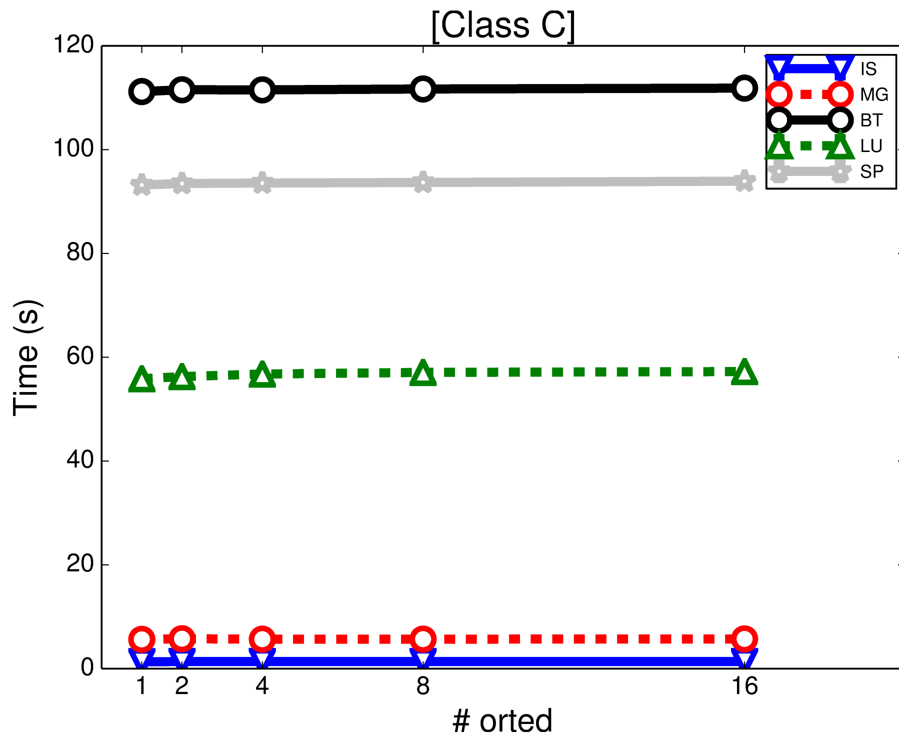


Figure 33: Execution time of each benchmark (input class = C) when running 16 processes using 1 to 16 ORTE daemons.

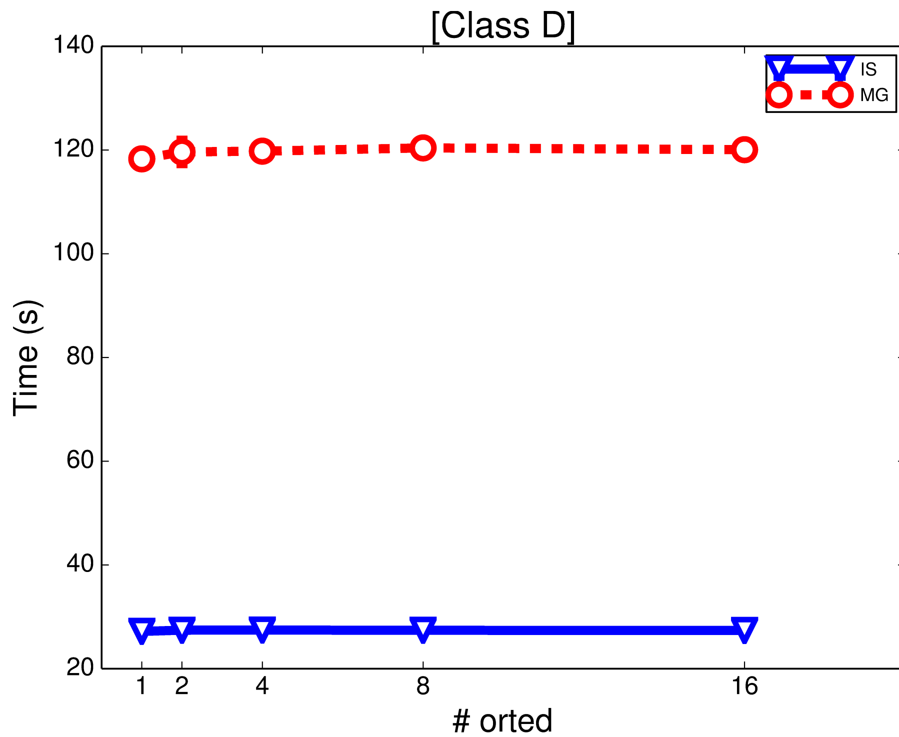


Figure 34: Execution time of each benchmark (input class = D) when running 16 processes using 1 to 16 ORTE daemons.

Table 8: Static overhead of `IS` and `MG` with increasing migration granularity, i.e. increasing number of ORTE daemons, compared with single ORTE daemon case. For each kernel, we run the tests using multiple datasets (B, C, D).

Benchmark	Class	# orted	Overhead %
IS	B	2	4.68
		4	5.18
		8	4.85
		16	4.85
	C	2	2.21
		4	2.68
		8	2.64
		16	2.64
	D	2	0.87
		4	0.96
		8	0.79
		16	0.64
MG	B	2	1.28
		4	4.36
		8	1.73
		16	3.24
	C	2	2.25
		4	0.62
		8	0.88
		16	1.57
	D	2	1.13
		4	1.25
		8	1.79
		16	1.50

cesses would require a single migration in case of 8 processes per ORTE daemon; or up to 8 migrations, as the number of processes per ORTE daemon decreases.

Table 9: Static overhead of BT, SP, and LU compared with single ORTE daemons case. For each pseudo-application, we run the tests using multiple datasets (B, C, D)

Benchmark	Class	# orted	Overhead %
BT	B	2	0.92
		4	0.93
		8	0.93
		16	1.17
	C	2	0.31
		4	0.29
		8	0.45
		16	0.60
SP	B	2	1.90
		4	2.83
		8	3.72
		16	4.12
	C	2	0.31
		4	0.40
		8	0.52
		16	0.79
LU	B	2	2.92
		4	4.37
		8	5.42
		16	6.10
	C	2	0.73
		4	1.63
		8	2.19
		16	2.48

Overheads due to migration

We characterized the migration overhead by running the benchmarks on the two 16-core computing nodes, which were connected via Gigabit Ethernet.

In this experimental scenario, which is summarized in Figure 35, we let each application execute on the two nodes employing 8 processes per node. On each node, we spawned 2 ORTE daemons; therefore, at node level, each daemon managed 4 processes. During the application execution, we triggered a migration: in particular, `mig` migrated four processes from one node to another, so that the new configuration was 4 processes on the first node and 12 processes on the second one.

To better observe the composition of the migration overhead, we split the migration time in seven contributions: coordination, CRIU dump, image migration, CRIU restore and finalization, which are the migration phases as already described in Subsection 5.3.4; and the image compression/decompression, which are optional and respectively happen before and after the image migration. The contributions are therefore the following:

COORDINATION
: The ORTE daemons get ready for the migration. All new communication towards migrating nodes are held. All other communications happen as usual;

CRIU DUMP
: The migrating ORTE daemon and its processes are dumped into an image file;

IMAGE COMPRESSION
: In order to save network bandwidth and transfer time, the image is compressed (optional);

PROCESS STATE MIGRATION
: Image is transferred to the destination node;

IMAGE COMPRESSION
: The image gets decompressed (optional, depends on whether the image was compressed before being transferred);

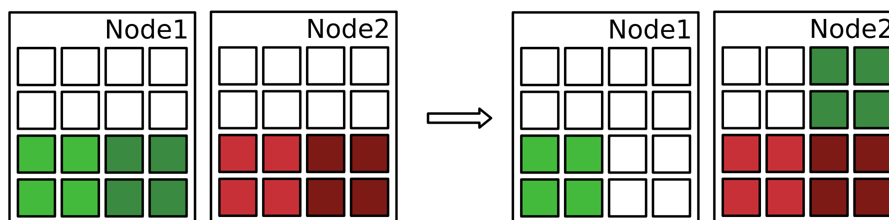


Figure 35: Experimental scenario used to asses the overheads induced by migration. An application composed of 16 processes is running on two computing nodes. The processes are equally distributed among the nodes, and, at node-level, processes are managed by two ORTE daemons (processes managed by different daemons are indicated using different colors). At some point, a group of four processes is migrated from the first node to the second one.

CRIU RESTORE
Image is restored;

FINALIZATION
The ORTE daemons get ready to restart communicating with the migrated one.

Please note that the execution time of all the aforementioned phases but coordination and finalization strongly depend from the size of the image file. In particular, the network that interconnects the computing nodes may become the bottleneck of migrations, and this is why we introduced the image compression/decompression phases.

We compressed the images using the GZIP algorithm [134]. Figure 36 shows the size of the compressed images normalized to the uncompressed one. Overall, compression is quite effective in reducing the size of the checkpoint image. The reason is straightforward: Open MPI allocates over 100MB of unused shared memory as ghost files initialized as zeros, and this makes the compressed images quite smaller than the compressed ones. In case of big datasets, this phenomenon is less evident, hence the higher size ratios of the type `C` datasets.

Figure 37 provides an overview of the measured migration times, comparing the cases with image compression against cases where no compression is applied. In case of big datasets (i.e., datasets of type C), compressing the processes images before transferring them to the destination node leads to higher execution times. Indeed, although

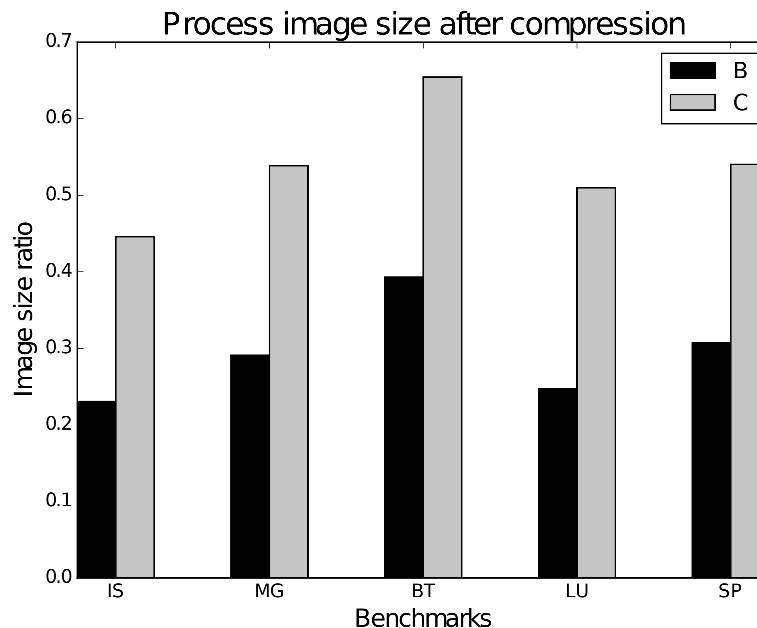


Figure 36: Size of the compressed process image normalized to the uncompressed size. For each application, we report the results for input data classes `B` and `C`.

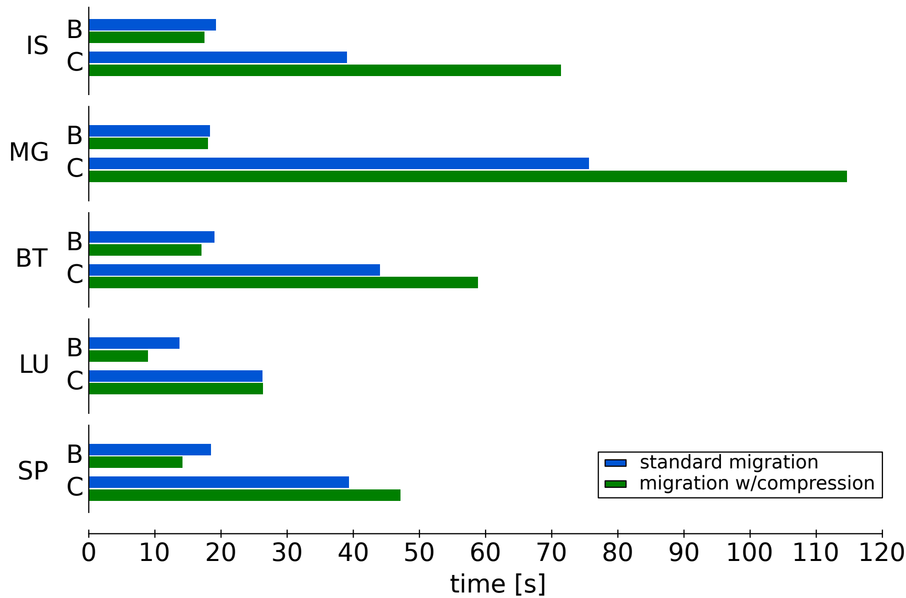


Figure 37: Time required to migrate a group of four processes. For each application, we list migration times for datasets type B and C, with and without compression.

the image takes less time to be transferred, compressing the image on the source node and decompressing it on the destination node takes time. The sum of the three components, i.e., compression, transfer and decompression, is higher than the time required to transmit an uncompressed image. In case of smaller datasets, due to the high compression ratios, compressing images is often convenient, but it does not lead to dramatic performance improvement either.

It is worth noticing that, in realistic HPC scenarios, applications are likely to feature big images, and this makes compression inconvenient in most cases. Moreover, we performed the aforementioned on two computational nodes connected via Gigabit Ethernet. Most recent HPC systems connect nodes using InfiniBand, which is much faster than Ethernet. In those cases, we do not expect compression to be needed even when dealing with small datasets: transfer time would always be in the range of seconds instead of tens of seconds.

Finally, Figure 38 shows the composition of migration times for the experiments that do not employ compression. In all the scenarios, archiving and transferring the processes images takes more than 90% of the total migration time. Moreover, although the time spent archiving the image gets higher in percentage as the datasets gets bigger, the migration time is always dominated by transfer time. This is a good result, since, as already mentioned, transfer time can be dramatically shrank by employing InfiniBand as a communication channel between computing nodes.

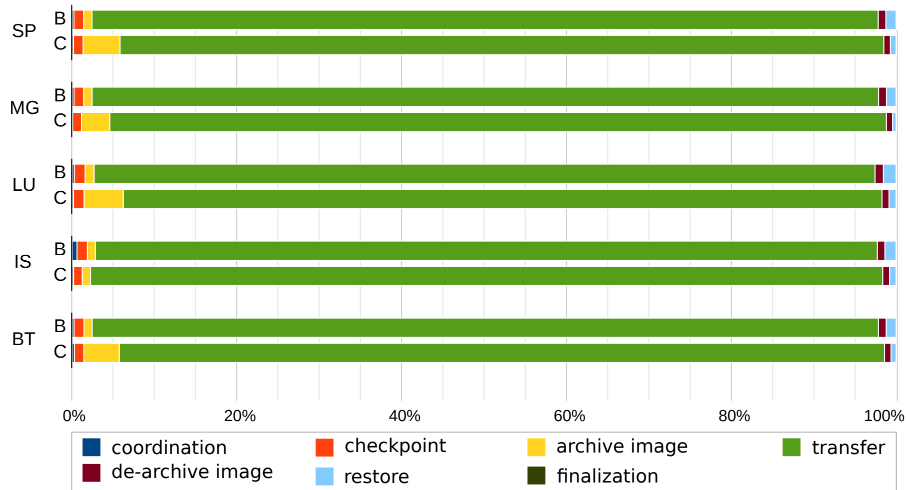


Figure 38: Migration time composition (percentage).

CONCLUSIONS

In this section, we introduced a novel approach to support process migration in the Open MPI framework. The main idea behind the approach is to execute applications employing multiple ORTE daemons per node instead of a single one. By doing this, we are able to treat each ORTE daemon, along with the MPI processes it is managing, as an entity that can be migrated without performing intrusive changes to the Open MPI framework itself. The application a part of whom is migrating is not aware of the migration. Processes that are not migrating just experience a temporary network slow-down (i.e., communications towards migrating processes are temporarily held). Communication towards non-migrating processes can instead be performed as usual.

The only negative effect of employing multiple ORTE daemons per node is that communication between processes that are in the same node but are managed by different ORTE daemons happens via socket instead of shared memory. Usually, this is not a problem: in the standard scenario, applications run on several computing nodes, and inter-node communication is the real bottleneck. Through experimental tests, we shown that applications executing on a single computing node suffer from a performance degradation that is limited (up to 5% in communication-intensive applications), gets smaller as the application dataset increases (bigger datasets means a lower communication ratio), and does not depend on the number of employed ORTE daemons. The last characteristic is quite useful, since it means that the migration granularity, i.e., the number of employed ORTE daemons, can be tuned at will without impact the performance of applications.

Compared to other state-of-the-art solutions, one of the major advantages of our approach is the maintainability. The extension introduced in the Open MPI runtime has a minimal impact on the other

Open MPI modules. Furthermore, it does not require any change on the applications code, and it does not rely on any virtualization layer. The latter characteristic enables gains in terms of performance with respect to approaches that are based on virtual machine allocation. In this regard, our proposal allows us to perform fine-grained migrations, since the resource manager can decide to migrate either an entire application or a subset of its processes. This feature also increases the controllability of the workload execution.

The major limits of the proposed process migration mechanism are similar to those of the other C/R based systems: the nodes of HPC systems must be homogeneous, i.e. the operating system (and kernel version), libraries version and application binaries must be perfectly identical. Moreover, performing the checkpoint with CRIU requires administration level permissions (root user in Linux) in all the nodes. As a future work, we may allow ORTE daemons to spawn application processes without such a requirement. From the MPI communication standpoint, the lack of InfiniBand support is currently the most important missing feature. However, the development of this component is currently ongoing.

6 CPU RESOURCE MANAGEMENT IN HPC SYSTEMS

In this chapter, we present our novel contributions in the context of the HARPA European project, which successfully terminated in 2017.

If you are interested in the HARPA project, please visit our website: www.harpa-project.eu

The goal of HARPA was to support next-generation embedded and high-performance many-cores in cost-effectively providing dependable performance, i.e., correct functionality and timing guarantees, throughout the expected lifetime of a platform and under thermal, power, and energy constraints. The HARPA novelty was in seeking synergies in techniques that have been considered to be virtually exclusive for the embedded or high-performance domains (worst-case tailored, partly proactive techniques in embedded, and best-effort reactive techniques in HPC). HARPA demonstrated the benefits of merging concepts from these two domains by evaluating key applications from both segments running on embedded and HPC platforms.

The framework proposed by HARPA is composed by two layers: the first one is low level (i.e., closer to the hardware) and is characterized by a very high responsiveness. It is called HARPA Run-Time Engine (HARPA-RTE). The second layer, which acts as a middle-ware between the RTE and applications, is the HARPA Operating System (HARPA-OS), which has a responsiveness in the order of tens or even hundreds of milliseconds and is composed by a system-wide resource manager (the BarbequeRTRM, see Appendix A) and a set of monitors and hardware faults prediction modules.

This section will deal only with the HARPA-OS layer. In particular, we will focus on the resource management part, since it was our contribution to the Project. Section 6.1 presents a feedback-based approach that allows the BarbequeRTRM scheduling policy to dynamically tune resource allocation according to the runtime-variable system status. Resource allocation is based on a set of Pareto-optimal configurations (i.e., resource allocations under which the application can optimally execute) that we identify at design time using Design Space Exploration techniques. By taking into account the feedback coming from applications, the scheduling policy is able to infer how different runtime conditions impact on the profiled relationship between allocated resources and applications quality level.

During the aforementioned work, we noticed the limitations of what we call discrete allocation, i.e., choosing a resource allocation among a limited set of pre-defined choices. Therefore, we chose to design and implement a new and more refined BarbequeRTRM scheduling policy that can allocate resources in the continuous domain. In Section 6.2, we present a scheduling policy that is again feedback-

based but does not need applications to be profiled at design time. This means that the scheduling policy is able to handle even completely unknown applications. Moreover, the scheduling policy uses the information coming from monitors and fault prediction modules in order to tackle performance variability and to level power consumption and temperature throughout the chip. Hence, it fully exploits the capabilities of the HARPA-OS in order to achieve a performance, temperature and variability-aware resource allocation.

Please note that, since the BarbequeRTRM is the central component of the HARPA-OS—which is composed by the BarbequeRTRM and a set of monitor and fault prediction modules—we will thereon interchangeably use the terms HARPA-OS and BarbequeRTRM.

HARNESSING PERFORMANCE VARIABILITY IN HPC

The contents of this section are partially published in [135]. You may want to consult Appendix A before venturing forth.

Nowadays, thanks to the progress of silicon manufacturing processes, multi-core processors can be found in High-Performance Computing (HPC) systems as well as in embedded and mobile devices. Modern laptops offer computational capabilities that are comparable to those of 70's supercomputers, and we expect this trend to hold in the future.

According to the technology road map and despite all the challenges described in Chapter 1, the silicon manufacturing will soon rely on a 10nm integration process, and such transistor density is bound to increase performance variability and the probability of silicon defects.

Due to the very high worst-case cost impact for technology nodes, the current solutions for hardware errors detection/recovering (e.g., checkpoint/restore) and the possible guard-bands to battle interference variations may not be scalable enough. For instance, in the case of HPC systems, when a computing node fails, it can be rebooted or, if the errors are persistent, it can be replaced. In the case of integrated chips, when a processor core fails, it cannot be replaced. This means that such systems must be able to properly work even though some cores are temporarily or permanently unavailable. In order to address the aforementioned issues, we need to introduce new attenuation techniques that increase the lifetime of the system.

In this Section, we present a novel approach to High Performance Computing. The main idea behind this work is to provide applications with resource and performance-awareness, so that they are able to negotiate resource allocation with a centralized resource manager. Being aware of the amount of resources that they have at their disposal (i.e., *which* as well as *how many* processing cores and accelerators), applications are able to tune their behavior in order to maximize their quality. Moreover, being able to assess their own quality, applications are able to provide the resource manager with feedbacks about the current allocation. By doing so, they are therefore able to min-

imize their resource usage while nonetheless complying with their Quality of Service goal, and this induces system-wide benefits such as optimal resource exploitation (more applications can fit the available resources), and the minimization of power, heat and aging.

Background

Given the increasing importance of performance variability, the sole maximization of throughput cannot anymore be the only concern of High Performance Computing. HPC systems must begin tackling problems that, until today, were exclusive of the embedded domain, such as optimizing the applications resource usage. This is exactly what we do in this work. In particular, our approach is based on the idea of allocating to applications the minimum amount of resources that allows it to satisfy its performance and Quality-of-Service requirements.

Similar practices are well known in scenarios such as cloud computing and multi-core based embedded systems.

In cloud computing, the term *elasticity* refers to the ability of a system to dynamically adapt to workload changes in order to match current available resources to current performance demand [136, 137].

Regarding multi-core embedded scenarios, *Pusukuri et al.* introduce ADAPT, a scheduling framework that monitors the resource usage of multi-threaded applications and dynamically selects both the scheduling and the thread-to-core mapping policies in order to reduce inter-application interference [50]. This in turn allows the framework to consolidate workloads on few resources and to put the unused resources in a low power state. The authors validated their approach on a 64-core Supermicro server, thus creating a strong link between multi-core systems—on whom resource-aware co-scheduling policies are usually focused—and High Performance Computing.

Allocating an optimal amount of resources to each application may not be enough: in order to optimize both resource usage and application performance, several previous works dynamically adapt the applications behavior to the available resources. A typical example is the concept of *adaptive parallelism* [138, 139], where the parallelism of an application (i.e., the number of threads) adapts itself to the current resource availability.

Bhadauria and McKee present the HOLISYN co-scheduling policy, which aims at optimizing resource usage of applications by using adaptive parallelism [140]. Their policy samples resource usage for each application during the initial part of their execution: during this phase, each application is executed with a varying number of threads. The applications that scale well with high numbers of threads are then executed in isolation to achieve a very efficient resource utilization. Conversely, the remaining applications are paired so that

high-resource-usage applications are co-scheduled with low-resource-usage ones.

Adaptive parallelism has been studied also in the field of transactional memory [141, 142]. *Mohtasham and Barreto* devise an adaptive parallelism scheme that is fully decentralized. The set of processes cooperate to reach an efficient and fair configuration. This aspect is very important: adapting applications to the current system status, e.g. to the system load and the available resources, needs a strong coordination between applications, since applications that take optimization decisions independently (i.e., without being aware of what the other applications are doing) will likely cause instabilities (see also the work presented in Subsection 3.3).

HPC scenarios also benefit from the dynamic tuning of applications. Several previous works present frameworks that are application-specific: ATLAS [143] for matrix multiplication, OSKI [144] for sparse matrix kernels, SPIRAL [145] for digital signal processing, SEPYA [146] for stencil computations.

To the best of our knowledge, there are not previous works that exploit the synergy between optimal resource usage and application adaptivity. We aim at coordinating the execution of applications by making HARPA-OS act as an arbiter that assigns an optimal amount of resources to each application. Applications are isolated in well defined groups of resources, so that: a) each application is aware of the amount of resources at its disposal, b) such amount of resources is exclusively owned by the application, and c) applications are able to configure themselves in order to make an optimal usage of the resources they have at their disposal. The amount of resources that is allocated to each application is dynamically selected among a set of predefined ones by taking into account the current performance of the application as opposed to the required one.

The HARPA Operating System

The HARPA-OS is the topmost layer of the HARPA software stack. Its role is to manage resource allocation while taking into account both the status of the system resources and the requirements of applications. This is done by combining pro-active and reactive strategies.

By the application side, the performance requirements can vary not only among different applications, but also during the execution of the same application. This is a common scenario in HPC systems, where the workload is mainly made by scientific applications. For instance, monitoring systems that aim at preventing natural disasters may need to dynamically tune the accuracy (or throughput) of their computation according to the environmental conditions.

In HPC environments, the common approach to guarantee the required level performance to applications is to statically reserve them

computational resources. This is usually done by employing virtualization techniques. Since the resource demand of applications may vary over time, however, the allocated resources are likely to be over-provisioned most of the time. Scaling the problem to the whole system, the amount of under-used resources may be substantial, and this would lead to two issues: 1) the available resources may be fragmented, and this limits the space for new applications; 2) power management techniques may be less effective because without a proper consolidation of the allocated computational resources, there can be processors or single cores that are not fully exploited, but they cannot be put in deep sleet state either.

The next paragraphs provide an overview of how the HARPA-OS addresses those issues. In particular, we will focus on the BarbequeRTRM, which is the central part of HARPA-OS and supports the execution of the applications in an *adaptive* and *performance-aware* fashion.

We already introduced the BarbequeRTRM application execution flow in Subsection A.2.1; however, for the sake of clarity, we will again remark the most important aspects. The BarbequeRTRM comes with an application library (RunTime Library, *RTLib*) that is in charge of synchronizing the resource allocation with the application life-cycle. In order to benefit from the *RTLib* support, applications must be adapted to the BarbequeRTRM application execution flow, which is represented in Fig. 39. In particular, the application execution must be modeled as a state machine that features the following states:

SETUP
: the application performs a set-up, e.g., it initializes variables and spawns threads. This action is performed only once;

CONFIGURE
: the application adapts itself to the current resource allocation, e.g., by modifying its parallelism level accordingly. This action is optional (i.e., it can be left unimplemented), and it is performed once every time the resource manager changes the resource allocation of the application;

RUN
: the application processes a chunk of data. This action is repeated until the termination of the application. If an application wants to declare a throughput goal, the goal must be expressed as desired number of *run* actions per second;

MONITOR
: the application exploits the run-time library API or some custom logic in order to asses the current performance and quality. If one or both of those are not satisfactory, the application can use the run-time library API to send a feedback to the resource manager. In this

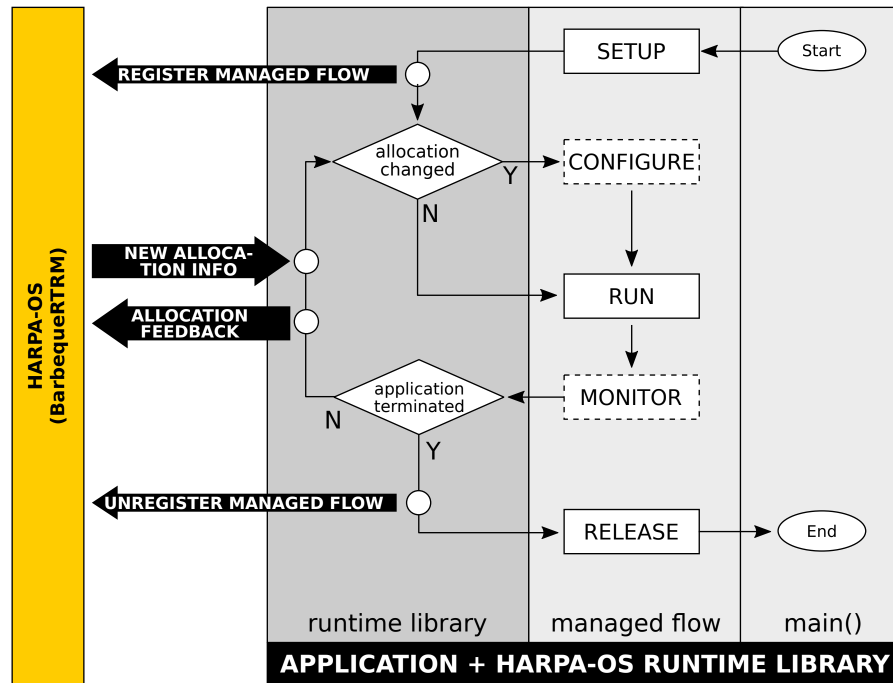


Figure 39: Overall view of HARPA-OS (applications side). From its main() function, the application instantiates the part of code that has to be managed and triggers its execution. From there on, the execution is transparently managed by the BarbequeRTRM runtime library, which drives the execution flow according to an internal state machine. Methods that are in dashed boxes are optional; that is, they can be left unimplemented.

stage, the application may also evaluate the possibility of changing its own performance/quality goals. This action is optional (i.e., it can be left unimplemented), and it is performed once after each *run* action;

RELEASE
the application terminates, e.g., it frees memory and joins threads. This action is performed only once.

When a managed application is not satisfied with its current performance or Quality of Service, the BarbequeRTRM can react in three ways:

- Migrating the application towards a different set of resources. This can be done using the already existent checkpoint/restart techniques. In the case of MPI applications, please refer to the work detailed in Section 5;
- Selecting a more performing configuration (i.e., resource allocation) from those that, according to a design-time application profiling, have been selected as the optimal ones for this application. This is exactly what we will deal with in this subsection;

- Tuning the current resource allocation by adding some resources or by moving the computation to more performing or more healthy cores. We will deal with this approach in the next subsection.

A feedback-based, performance-aware allocation policy

In the context of this work, we designed and implemented an allocation policy that performs a performance-aware resource allocation. The main idea behind this scheduling policy is that each application is known a priori—which makes sense, as applications must be adapted to the BarbequeRTRM execution flow in order to be managed—and features a set of optimal resource allocations from whom the allocation policy can choose. This set, which, as already explained in Section A.3, is called *recipe*, is built during an off-line application characterization and contains resource allocations (Application Working Modes, AWMs) that are Pareto-optimal for the application.

Indeed, information that comes from an off-line analysis is often inaccurate: at runtime, co-running applications that contend on shared resources; hardware faults; performance variability induced by aging; or simply different datasets, may cause the application to behave differently from what was observed during the off-line analysis. Our scheduling policy is therefore based on the following idea: instead of inserting in the recipe only the AWMs that are Pareto-optimal with respect to the application objectives, we insert also some AWMs that are close to the optimal ones. For instance, if an optimal AWM features 10 cores, we may also insert in the recipe AWMs that feature 11 or 12 cores. The set of sub-optimal AWMs and the optimal AWM to which they refer create therefore a cluster from which the scheduling policy can dynamically choose configurations during runtime in order to perform a fine-grained tuning according to the current system status.

This work leverages the concept of feedback but still employs a static set of allocation choices. This approach will be refined in Section 6.2, where we will present a completely application-agnostic scheduling policy.

Listing 1 illustrates what happens to running applications that send a feedback to the BarbequeRTRM. We call the feedback "performance gap", as, in order for the HARPA-OS to correctly interpret it, it must be computed as the percent distance between the current performance and the desired one. Given the throughput of the application under the current AWM (line 4) and the current performance gap (line 6), the policy estimates the desired performance level as reported (in a simplified form) in line 7. In lines 11–17, the policy looks for the Application Working Modes whose profiled performance value is closest to the expected one. Finally, the resources that are requested in the selected AWM are mapped on the hardware (line 19).

Listing 1: Pseudo-code of the performance-aware resource allocation policy.

```
1   Data: R: list of RUNNING applications
2
3   for app in R do
4       curr_awm ← GetCurrentAWM(app)
5       // Percent gap is in (-1.00, 1.00]. Closer to 0 is better.
6       gap ← GetPerformanceGap(app)
7       exp_value ← GetValue(curr_awm) / (1 + gap)
8       max_score ← 0
9       available_awms ← GetWorkingModes(app)
10
11      for awm in available_awms do
12          score = evaluateAWM(curr_awm, awm, gap)
13          if max_score < score then
14              max_score ← score
15              found_awm ← awm
16          fi
17      done
18
19      MapResources(found_awm)
20  done
```

Experimental Setup

We validated our approach using a Rainfall-Runoff (RR) model that is as a part of the Floreon+ system [147]. The model, which is one of the application use-cases of the HARPA project, predicts the water discharge levels of a geographical area by analyzing the recent precipitations information. Inasmuch as that such information is inaccurate, the model projects any inaccuracy on the output by constructing confidence intervals using the Montecarlo (MC) method. For each area, there are three flood warning levels: *Flood watch*, *Flood warning* and *Flooding*.

The quality of service (QoS) of the RR model is expressed in terms of accuracy and is therefore proportional to the number of performed Montecarlo iterations [148, 149]. Each of the three flood warning levels can be mapped to a given quality of service level:

NO WARNING
: The water level did not exceed any warning level; therefore, there is no need for the output to be highly accurate. The model must perform 1250 MC iterations every 60 minutes.

FLOOD WATCH
: The water exceeded the first warning level; there is not a strict need of increasing accuracy, but the water level information will be checked more frequently. The model must perform 1250 MC iterations every 10 minutes.

FLOOD WARNING
: The water exceeded the second warning level; in order to correctly

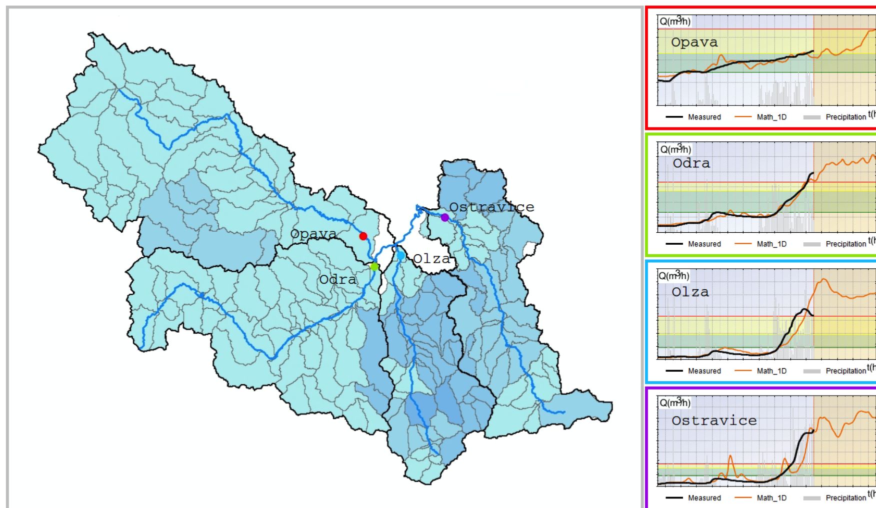


Figure 40: Main catchments (left) and outlet hydrographs (right). In the hydrographs plots, black lines show the measured discharge, orange lines show the simulated one. X-Axis: time in hours t(h), Y-Axis: Discharge, cubic meters per hour, Q(m^3/h)).

predict a potential flooding, the model must be quite accurate. The model must hence perform 2000 MC iterations every 10 minutes.

flooding
The water exceeded the third warning level; in this case, the model output must be very accurate. The model must perform 3000 MC iterations every 10 minutes.

The experiment monitors the run-time behavior of 4 concurrent instances of the uncertainty module. Each of these instances models the RR uncertainty for a different catchment of the Moravian-Silesian region: the Opava, Odra, Ostravice and Olza catchments [147] (see Figure 40). The catchments are ordered according to the impact in case of flooding (the lower the index, the higher is the importance):

- C_1: *Ostravice* - Functional urban areas with high population density and industrial areas in floodplain zones.
- C_2: *Olza* - Flood sensitive zones in urban areas.
- C_3: *Odra* - Mountains in the upper part of the catchment can cause significant runoff. Less exposed urban areas.
- C_4: *Opava* - Soils with low infiltration capacity.

Each catchment is simulated independently, and individual instances do not interact with each other.

In order to enable the HARPA-OS support, we partitioned the application source code as follows: during the *setup* phase, the application computes the initial RR model; then, in the *configure* phase, it

activates a suitable number of threads basing on the amount of allocated CPU cores. At this point, the application also chooses how to suitably divide the input data into chunks that will be independently analyzed by the active threads. During each *run* phase, the results of the RR model are incrementally refined by running a high number of Monte-Carlo iterations on a chunk of samples. After each *run* phase, the *monitor* phase monitors the quality of the execution by estimating the remaining execution time of the RR model and by comparing it with the ideal completion time (e.g. 10 minutes in Flood warning level). If the estimated execution time is too high, i.e., if the deadline is going to be violated, the application notifies it to the HARPA-OS, which will evaluate whether a more performing AWM must be allocated to the application. When the resource allocation changes, the application enters again in the *configure* stage, where it can tune its parallelism level and the input chunk size accordingly.

We executed the four application instances on an HPC system that featured twelve 6-core AMD Opteron 8425HE processors, for a total amount of 48 CPU cores. The results are shown in Figure 41. For each instance, we can observe the achieved throughput normalized to the ideal one (the closer to 100%, the better), and the number of cores that are allocated to each instance.

As can be easily observed from the charts, all the instances terminate executing close to the deadline (600s). This is exactly the goal of our approach: by minimizing resource allocation while taking into account performance, we provide applications with only the resources that are strictly needed to make them comply with their performance goal. This means that applications do indeed terminate before their deadline, but their termination will be as close to the deadline as possible.

Whereas some instances feature a throughput that is often close to the ideal one (instances 1 and 4), other instances alternate execution phases whose throughput is either lower or higher than the ideal one. The reason is straightforward: since resource allocation is discretized, the scheduling policy is not able to allocate any custom amount of resources to each application; conversely, it may be forced to alternate AWMs that are either under- or over-provisioned for the application. An application whose resource allocation is under-provisioned will soon experience an over-provisioned allocation, which is likely to temporarily steal resources from the other instances, thus creating further oscillation in the experienced throughput. This problem can be solved by switching from discrete to continuous resource allocation (we will deal with that in the next Section).

The 48-core system which we used to validate our approach does not provide means to measure power and energy consumption; therefore, we performed these tests on a 16-core NUMA machine. The machine was composed by four nodes, each node featuring a Quad-Core

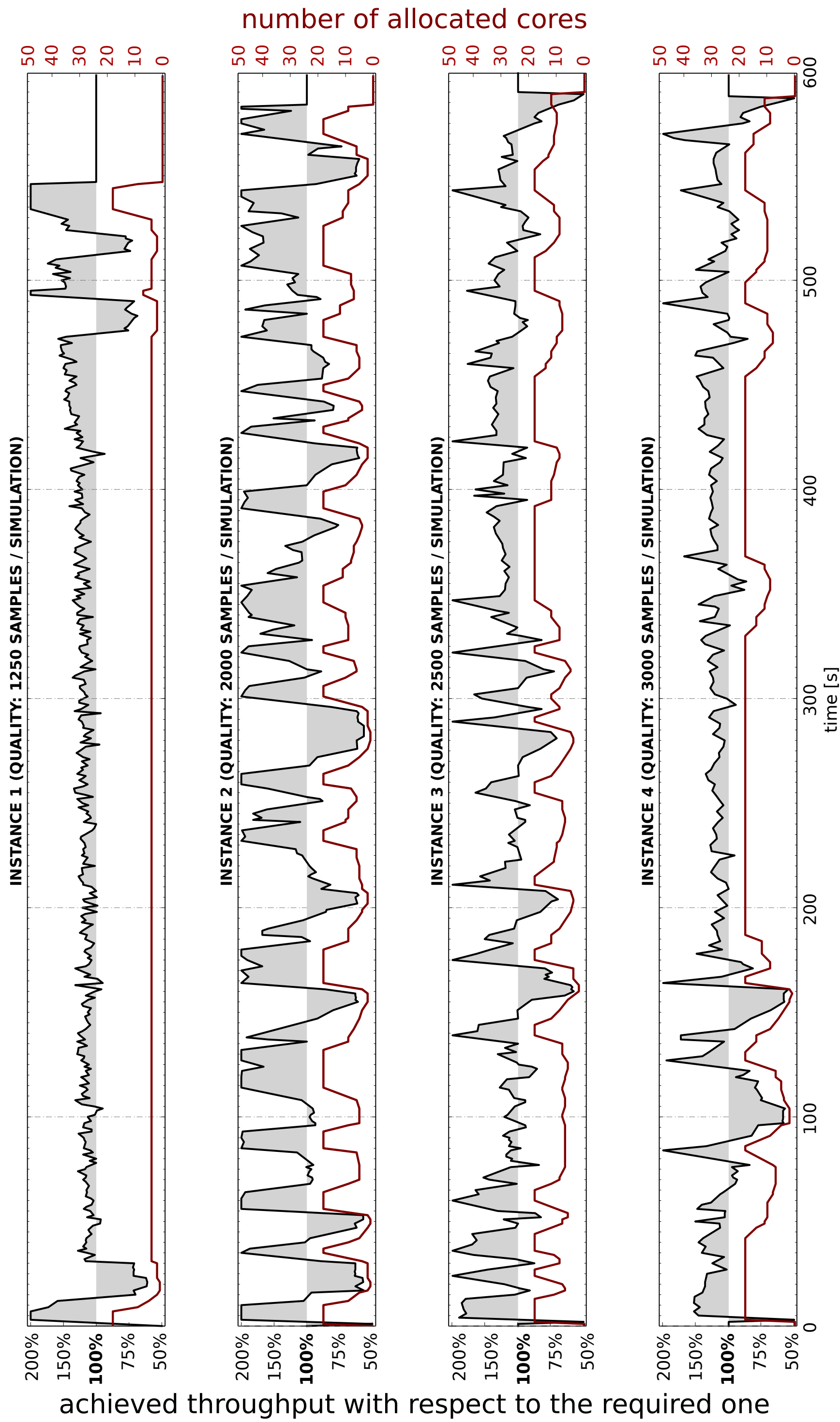


Figure 41: Four instances of the Uncertainty application running on a system that features 48 cores. Red lines represent the amount of cores that are allocated to each instance. Black lines indicate the achieved throughput met the expected one, in percentage (the close to 100%, the better).

AMD Opteron 8378 Processor and 8GB of memory. We measured the power consumption using *ipmitools* with a sampling rate of 1 second.

Figure 42 shows the power consumption of the machine during the execution of one instance of the application (1250 Monte Carlo samples) in both HARPA-OS-managed and unmanaged mode. Please note that the power consumption is represented on top of the idle power consumption of the machine, which is $300W$. The figure shows two important aspects: first of all, HARPA-OS management substantially reduces the peak power consumption of the workload, and, as already mentioned, such achievement is known to have positive effects on temperature and mean time to failure of the hardware. Second, the total energy consumption, which is the area under the lines, is roughly the same in the two scenarios. This means that the management of application operated by HARPA-OS does not induce energy inefficiency.

Conclusions

In this subsection, we have applied the adaptive performance-aware execution model introduced by the HARPA-OS in the context of a real scientific application domain on a multi-core HPC system. The experimental results shown that this approach is indeed capable of making applications comply with their runtime-variable QoS requirements

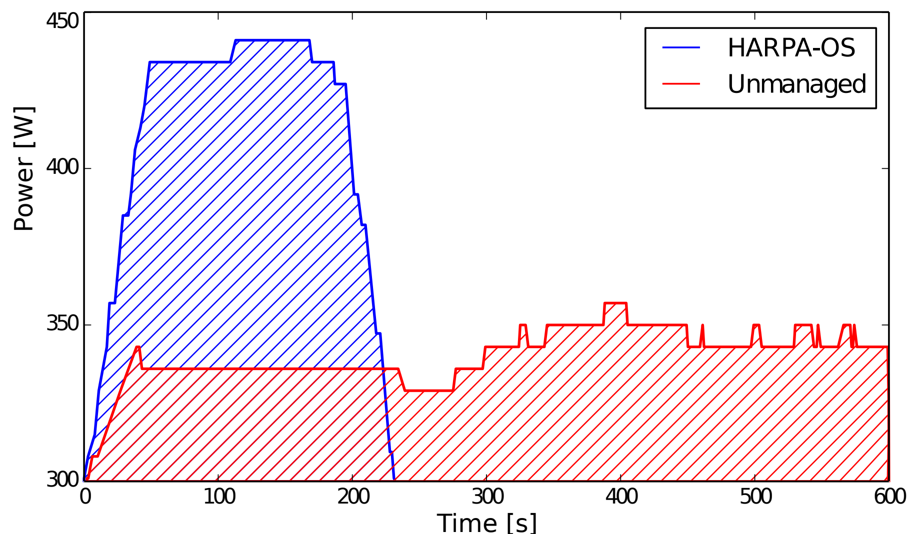


Figure 42: Power consumption of a single instance of Uncertainty, in HARPA-OS-managed and unmanaged mode, on a 16-cores NUMA machine. The power consumption is represented on top of the idle power consumption of the machine, which is 300W. The energy consumption of the execution, which is represented by the area under the lines, is roughly the same in the two scenarios; however, the managed execution shows far lower power consumption peaks with respect to the unmanaged execution.

while also minimizing resource usage and hence power consumption. Moreover, doing so does not induce energy inefficiency. From the application perspective, the minimization of resource usage does not induce unwanted effects, inasmuch as deadlines are always complied with. From the system perspective, instead, power consumption is positively affected. According to the literature, this benefit is easily translated into a reduction of temperature and aging effects, with a consequent improvement of the overall system reliability and dependability. Moreover, as a side effect, the reduction of power consumption and temperature of the processors also leads to benefits in terms of cooling costs, which are a not negligible expense item for HPC centers.

A WORKLOAD-AGNOSTIC RESOURCE USAGE OPTIMIZATION

The contents of this section are partially published in [150, 151].
You may want to consult Appendix A before venturing forth.

The ever increasing performance requirements of HPC applications call for architectures with more and more processing resources. Such architectures are typically subject to thermal and energy budgets, and running applications are allowed a wild competition for shared resources; as a consequence, making applications comply with their Quality of Service (QoS) goals is becoming increasingly difficult.

While such concerns can be addressed by knowing the characteristics and, possibly, the arrival time of applications, one does not often have the luck of knowing the workload composition a priori; moreover, especially in the case of applications whose behavior strongly depends on the input data or on unpredictable sources such as the environmental condition (e.g., the application used to validate our approach in the previous section), applications resource demand may be runtime variable.

In this Section, we address the problem of allocating resources to *uncharacterized* HPC applications with runtime variable QoS goals and software parameters. We leverage a runtime-managed execution flow that provides applications with QoS, performance and resource awareness.

Experimental results show that our approach makes applications comply with their performance goals, while allocating them only the resources that are strictly needed to do that. We also implement our approach as an extension to the Barbeque Run-Time Resource Manager (see Chapter A) in order to enhance the quality-aware HPC support introduced in the previous subsection.

Thanks to our management support, the resource manager got at its disposal a pool of unused resources that can be: put off-line to to save energy; used to run more applications; or used to temporarily store faulty and hot cores, so that cooler and healthier cores are used by active applications. We mainly focus on the latter option: in particular, we implement a resource mapping policy that avoids the usage

of faulty cores and evenly spreads heat throughout the chip, hence evenly distributing ageing effects on the hardware.

Background

Modern architectures feature a high number of shared resources that are used to concurrently run multiple tasks or to accelerate parallel ones. This is true not only for High Performance Computing (HPC), but also for middle-to-high end embedded systems, which are quickly narrowing the gap with desktop computers [152, 153].

Performance increase, however, comes at a cost: the power consumption of such architectures is becoming unbearable in terms of supplying/cooling costs [27, 28] (mostly in HPC) and chip reliability [29].

In this work, we address the problem of mitigating power consumption while meeting the performance requirements of the running applications. We state the problem in terms of dynamically reserving to applications only the processing resources that they strictly need to achieve their goals.

This topic has already been addressed in literature: resource managers allocate resources to applications while trying to comply with both application specific (throughput and Quality of Service) and system-wide (power, temperature and reliability) goals. Modern resource managers are *workload-aware*, i.e., they take into account the characteristics of applications to optimize the resource allocation [154]. Such information is well known as *application profile* and can be extracted at design-time or at runtime. We refer to the process of extracting the application profile as *application characterization*.

Being performed off-line, design time characterization does not induce overheads; however, it suffers from non-trivial issues. First of all, the contention on shared resources, which cannot be analyzed offline if the workload and the arrival times of each application are not known a priori, causes design time profiles to be inaccurate at runtime [47]. Second, during their lifespan, applications may undergo multiple execution phases, depending on input data, external events or computing stages. Therefore, associating a single profile to one application may also lead to inaccuracy [155].

There are also applications that add yet another layer of complexity: conversely to number-crunching applications, their configuration, which we define as the union of their QoS requirements and the value of their software parameters, is often runtime-variable. For example, in the case of a video playing application under a constant resource allocation, the possibility of achieving a given frame rate depends on parameters that the user may want to configure at runtime, e.g., resolution or frame size [156]. In a surveillance system, smart camera applications may reduce frame rate or resolution, hence entering

a low-power mode, when no moving objects are detected. Medical applications may require high image processing accuracy only when needed (e.g. analysis of critical sections of a X-ray plate), otherwise focusing on the responsiveness of the user interface. These scenarios offer interesting opportunities to reduce power consumption and boost energy efficiency, and, in turn, this brings benefits in terms of thermal stress avoidance, reliability and reduction of cooling costs.

Summary of the Work

Our work extends the Barbeque Run-Time Resource Manager (BarbequeRTRM). We modified the BarbequeRTRM runtime library, which is the component in charge of synchronizing the applications execution with the resource allocation, so that it acts as a decentralized resource manager. In the new configuration, the runtime library linked to each running application negotiates resource allocation with the BarbequeRTRM. Once the application receives the resources, however, it is not able to use all of them: conversely, the runtime library directly handles resource assignment by letting the application use only the resources that are strictly needed to reach the performance goal. If a part of the resources results to be unused, the runtime library notifies the BarbequeRTRM that it can seize it back. If, on the contrary, the application needs more resources, the runtime library asks for them to the BarbequeRTRM.

The negotiation between the runtime library and the BarbequeRTRM is based on the concept of *application runtime profile*, which is a data structure that contains useful runtime statistics such as the application current resource usage, the maximum expected usage (computed using confidence intervals), the current application throughput and, optionally, a custom set of performance counters.

Finally, we developed a new BarbequeRTRM scheduling policy that is called PerDeTemp. The policy makes use of the applications *runtime profiles* and the information coming from temperature sensors and fault predictor modules in order to minimize resource usage while evenly spreading heat and power consumption though the chip and counteracting the effects of performance variability.

Please note that this approach provides a link to most of our previous works. Resource allocation is actuated by using the Linux Control Groups, whose accuracy is maximized thanks to the approach presented in Section 2.3. Scheduling choices are based on application characterization (see Subsection 3.2) but, this time, the characterization is performed during runtime. Moreover, resource allocation is handled in part by the centralized resource manager and in part by an application-specific decentralized manager, thus exploiting the synergies highlighted in Subsection 3.3 and reducing the complexity of the scheduling policy. Finally, this work enriches and extends the

Indeed, this work employs all the techniques that we presented in the previous chapthers except the ones that address big.LITTLE architectures.

"resource minimization via late termination" approach that we presented in Subsection 6.1. It does so by providing a richer information exchange between the BarbequeRTRM and the runtime library and by enabling a continuous resource allocation instead of using a discretized one.

A partially de-centralized resource management

The main idea behind this work is to move a part of the resource management complexity to the applications side. Indeed, the management logic remains transparent to applications: it is executed by the BarbequeRTRM runtime library.

In the typical BarbequeRTRM design, the resource manager allocates resources to an application, and the runtime library, which is linked by each managed application, performs all the actions that are needed to drive the application execution flow accordingly. Our approach modifies the aforementioned design as follows:

- instead of trying to allocate to applications an ideal amount of resources, the resource manager allocates them a resource budget, i.e., the biggest set of resources that, according to the current system status, can be exclusively reserved for the application;
- the resource budget is not directly exploitable by applications. Instead, it is managed by the runtime library, which acts as an application-specific resource manager and allocates resources to the managed application according the declared performance goal;
- the runtime library is in charge of understanding which is the minimum set of resources that can be exploited by the application. If the resource budget results to be over-sized, the runtime library will return the unused resources to the resource manager. On the contrary, if the budget is under-sized, the runtime library will ask the BarbequeRTRM to increase it;
- the runtime library also monitors the application execution and collects statistical information that may help the BarbequeRTRM to perform system-wide resource mapping. This information is bundled with the one that is used for resource budget negotiation purposes. The resulting data, which is an accurate descriptor of the application current behavior, is called *application runtime profile*;
- the resource manager scheduling policy is only in charge of computing the resource mapping: the amount of resources that

are needed by the application is computed by the runtime library, while the scheduling policy, according to the system status (temperature, faults. . .) chooses *which* resources will be allocated to reach that amount.

The resulting execution flow of applications is shown in Figure 43. The upper part (APP) represents the execution of application code during time. The middle part of the figure (RTLIB) represents the runtime library, which forces the application to follow the BarbequeRTRM applications execution flow (see Subsection A.2.1). Finally, the lower part (RTRM) shows the actions taken by the runtime resource manager (i.e., the BarbequeRTRM).

Once started, the application passes the control to the runtime library, which notifies the presence of a new application to the resource manager. While waiting for the resource manager to allocate a resource budget to the application, the runtime library proceeds by invoking the Setup stage to mask communication overheads. Then, the application enters the Configure stage, where it tunes its software parameters according to the the amount of resources that the runtime library allocates from the budget. At this point, the application proceeds with a bust of Run and Monitor phases that continues until the runtime library changes the allocation. In that case, the application performs the Configure phase and begins a new bust of Run and Monitor phases. While the application executes, the runtime library also sends runtime profiles to the resource manager, hence transparently tuning the resource budget size. Please note that the mapping of the resource budget (i.e., *which* resources as opposed to *how many* resources) can be modified by the resource manager over time, e.g. to perform load balancing or to avoid faulty cores. In that case, the runtime library actuates the mapping change as soon as possible by employing the Linux Control Groups *cpuset* controller (see Chapter 2).

Defining a throughput goal

After each Monitor Stage, the runtime library automatically compares the throughput of the application against the declared performance requirements. In case of unsatisfactory allocation, the runtime library updates the resource allocation accordingly. If the current resource budget results to be unfit (e.g., under or over-provisioned), the runtime library also sends the current *application runtime profile* to the resource manager. To enable the runtime library automatic performance monitoring, the application needs register a performance goal.

There are two types of performance goals: Cycles Per Second (CPS) and Jobs Per Second (JPS). CPS and JPS goals reflect two main types of Processing Stages: some applications spawn multiple threads that collaborate to perform a job; in this case, having more resources

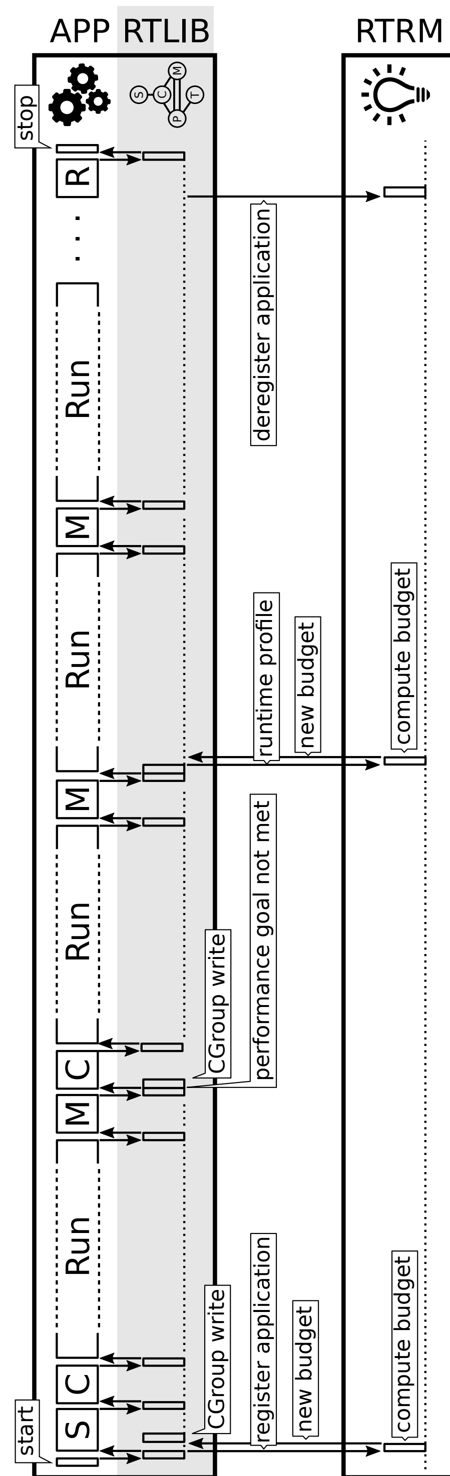


Figure 43: Example of interaction between BarbequeRTRM and the runtime library. After the application starts (top-left corner), the runtime library starts driving the applications execution flow (**S**etup, **C**onfigure, **Run**, **M**onitor, **R**elease). The runtime library locally updates the resource allocation by using CGroups, hence avoiding synchronization overheads. The runtime library can also send the application runtime profile to the resource manager, which may change the allocated resource budget accordingly.

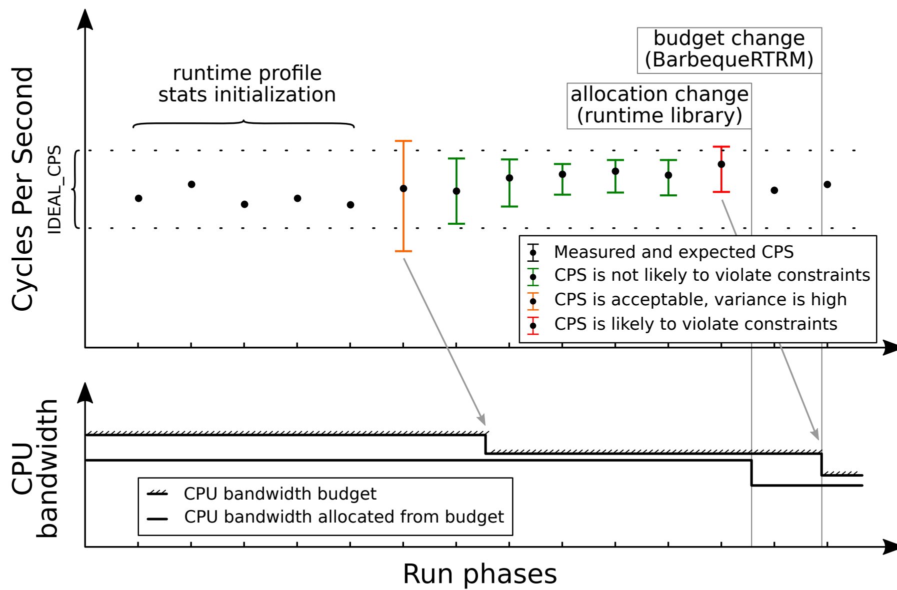


Figure 44: Runtime-library-side resource allocation. After each Run phase of the application, the runtime library computes the current CPS, and, by computing confidence intervals, it detects whether future CPS violations (i.e. Run phases at non-ideal CPS) are likely to happen. In that case, it tunes the allocation accordingly. CPU bandwidth allocation cannot exceed the resource budget allocated to the application by the BarbequeRTRM; however, if it detects a resource under or over-provisioning, the runtime library can prompt the BarbequeRTRM to modify the allocated budget.

means spawning more threads, which results in a shorter Run phase time. Conversely, other applications spawn multiple threads, and each thread performs its own job. In this case, the cycle time is more or less constant; however, having more resources at their disposal, applications are able to perform an higher number of jobs per second during the Run phase. JPS goal is best suited for this scenario, and all the CPS-related queries are also available in their JPS version.

Figure 44 shows how the runtime library checks the current throughput against the desired one. After each Run phase of the application, the runtime library computes the current throughput (either CPS or JPS), and, by computing confidence intervals, it detects whether future throughput violations are likely to happen. In that case, it tunes the allocation accordingly.

Computing the ideal resource budget

In the context of this work, we addressed the allocation of CPU time.

The concept behind CPU time allocation is straightforward: first of all, one must compute the total CPU time offered by a processor during a given period; second, one must divide that time among applications. For example, an octa-core processor offers 800 ms of total CPU

time over a period of 100 ms; allocating half of the total CPU time to an application would mean allocating it 400 ms CPU time over 100 ms, i.e. "four equivalent cores" or 400% *CPU bandwidth*. Please note that allocating 400% bandwidth to an application does not say anything about resource mapping. Using either four CPUs full-time or eight CPUs half-time results in 400% bandwidth. Whereas the allocated CPU time is computed by the runtime library, the resource mapping is selected by the scheduling policy by performing a multi-objective optimization that takes into account:

From now on, with "bandwidth" we will refer to CPU bandwidth.

- the performance of each core (e.g., the amount of unallocated CPU time and the presence of shared caches);
- the temperature of each core;
- the performance variability due to aging and faults, as computed by the fault prediction modules

The fault prediction modules are not part of our contribution; therefore, we will not detail them in this dissertation. Please refer to [150].

In order to understand the bandwidth requirement of an application, we must compute the distance between current and desired performance. When an application declares a CPS goal, the runtime library starts computing the percent distance between the real and goal CPS. We call such distance *CPS gap* and define it as:

$$cps_{gap} = \frac{CPS_{curr}}{CPS_{goal}} - 1 \quad (5)$$

where cps_{curr} and cps_{goal} are the current and goal CPS, respectively. For example, if an application requires a throughput of 20 CPS (i.e., 20 Run phases per Second) and is executing at 18 CPS, its CPS gap equals to $\frac{18}{20} - 1 = -0.1$, i.e., the application is 10% too slow.

In our examples, we will always use CPS. Please note that using JPS leads to the same equations.

To address data-induced performance variability, we generalize the formula by defining a minimum and maximum required performance value. For example, the application can require a throughput of $20 \pm$ 5% CPS, i.e., the runtime library will be satisfied as long as the CPS will stay in the range $[19.0\text{–}21.0]$.

The new equation, which is the same of Equation 5 when the minimum CPS goal is equal to the maximum one, is:

$$cps_{gap} = \begin{cases} \frac{CPS_{curr}}{CPS^{M}_{goal}} - 1 & \text{if } CPS_{curr} \geqslant CPS^{M}_{goal} \\ \frac{CPS_{curr}}{CPS^{m}_{goal}} - 1 & \text{if } CPS_{curr} \leqslant CPS^{m}_{goal} \\ 0 & \text{else} \end{cases} \quad (6)$$

where CPS^{m}_{req} and CPS^{M}_{req} are minimum and maximum required performance, respectively. Equation 6 can be reused to define a gap for CPU bandwidth:

$$b_{gap} = \begin{cases} \frac{B_{curr}}{B^{M}_{goal}} - 1 & \text{if } B_{curr} \geqslant B^{M}_{goal} \\ \frac{B_{curr}}{B^{m}_{goal}} - 1 & \text{if } B_{curr} \leqslant B^{m}_{goal} \\ 0 & \text{else} \end{cases} \tag{7}$$

where B^{m}_{goal} and B^{M}_{goal} are the bandwidths that would allow performance to stay in the interval $[\text{CPS}^{m}_{goal} - \text{CPS}^{M}_{goal}]$. As the reader can guess, the goal of the scheduling policy is to translate a performance interval $[\text{CPS}^{m}_{goal} - \text{CPS}^{M}_{goal}]$ into a bandwidth interval $[B^{m}_{goal} - B^{M}_{goal}]$. Doing this means translating CPS gaps into bandwidth gaps: if an application complains about its current performance, the policy must understand how to adjust the current (real) CPU usage to solve the issue.

By construction, each Run phase executes the same code on different data, and averaging the CPS throughout the current *fixed-configuration burst* mitigates the effects of data variability on performance. Therefore, at least in the case of CPU bound applications, we can expect that increasing the CPU bandwidth of an application, its throughput will increase by the same amount. In other words, giving an application $X\%$ more CPU time, its performance increase will be roughly $X\%$. This is not true in the case of memory bound applications, where a $X\%$ bandwidth increase usually leads to $Y\%$ performance increase, with $0 \leqslant Y < X$. However, given that in this case a $X\%$ CPU time increase cannot cause a performance gain greater than $X\%$, iterating this procedure multiple times allows the scheduling policy to reach the desired performance level even if the application is memory bound, without incurring in instabilities.

As a consequence, regardless of the type of application, we can safely assume a strict relationship between allocated resources and performance. In the best case, i.e. CPU bound applications, CPS and bandwidth gaps will be equal. This leads to:

$$b_{gap} \sim \begin{cases} \frac{P_{curr}}{P^{M}_{goal}} - 1, & \text{if } P_{curr} \geqslant P^{M}_{goal} \\ \frac{P_{curr}}{P^{m}_{goal}} - 1, & \text{if } P_{curr} \leqslant P^{m}_{goal} \\ 0, & else \end{cases} \tag{8}$$

which leads to:

$$B_{ideal}^{m} \sim \frac{B_{curr}}{1 + cps_{gap}} \qquad \text{if } cps_{goal} < 0 \tag{9a}$$

$$B_{ideal}^{M} \sim \frac{B_{curr}}{1 + cps_{gap}} \qquad \text{if } cps_{goal} \geqslant 0 \tag{9b}$$

This means that, given a CPS gap, we can estimate the CPU allocation, being it the minimum or the maximum one depending on the sign of the CPS goal. In case of memory bound applications, the reachability of the ideal allocation is nonetheless guaranteed: being applications able to send feedbacks to the resource manager, allocations will be continuously refined until an ideal allocation is found.

The PerDeTemp Scheduling Policy

We implemented a scheduling policy that computes resource mapping by merging the application runtime profiles with the data that comes from temperature sensors and fault prediction modules. The current version of the policy targets the allocation of CPU cores.

When executing a multi-threaded application on a homogeneous multi-core CPU, we expect each core to be as performing as its siblings. Unfortunately, this is not always true: each core may be subject to a different degree of performance degradation due to temperature, aging and possibly faults. Knowing which cores are actually characterized by a homogeneous performance would indeed be useful for resource management purposes: for instance, this information could be used to evenly distribute the load (and thus the aging effects) on the sane cores or to minimize the inter-threads synchronization overheads that are induced by performance variability. Given that the GNU/Linux CPU frequency governors do not handle this information, we developed a resource allocation policy that tackles the issue. The policy focuses on three objectives:

- making each application comply with its performance requirements;
- minimizing the effect of degradation on the performance of applications;
- mitigating the hardware aging process by leveling the temperature over the whole chip.

Given these objectives, we called the policy PerDeTemp (PERformance, DEgradation, TEMPerature). The main idea beneath PerDeTemp is to allocate to the managed applications the minimum amount of CPU cores that allows them to comply with their performance requirements. As to mapping, the set of cores that is allocated to each application is selected so that the cores feature a minimum or at least

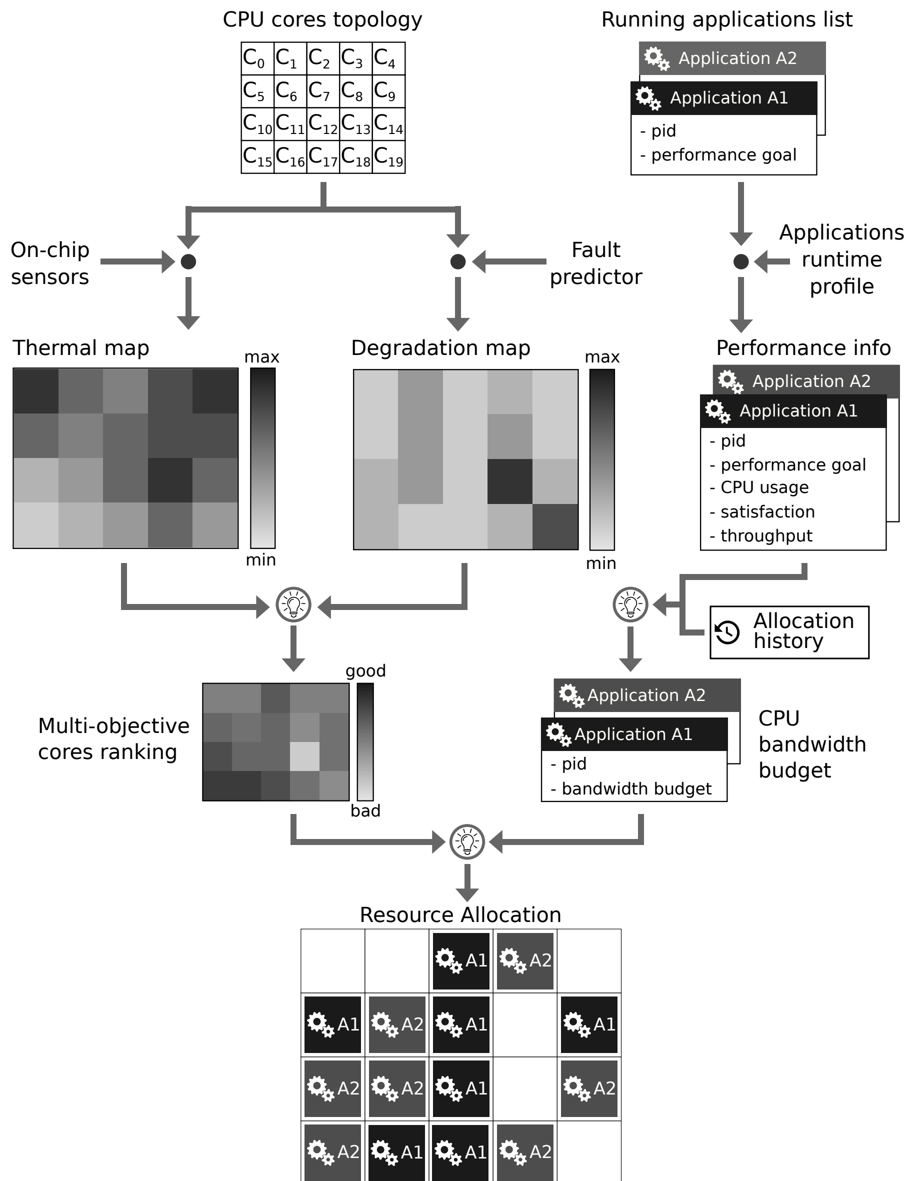


Figure 45: The PerDeTemp allocation policy. The CPU cores topology is annotated with the information coming from temperature sensors and fault prediction modules. This results in a multi-objective cores ranking that will be used to compute the resource mapping. CPU bandwidth budget is instead computed by exploiting the information that comes with the runtime profiles. In particular, runtime profiles contain a satisfaction metric that is used by the runtime library to express whether the current CPU budget is under or over-provisioned. Resource allocation is the union of bandwidth allocation (i.e., how many resources) and CPU mapping (i.e., which resources).

homogeneous degradation. During the process of selection, cool cores are preferred to hot ones, and the resource allocation is periodically re-computed to level the heat over the whole chip. To allocate resources in a performance, degradation and temperature-aware fash-

ion, PerDeTemp needs information about the status of both applications (current performance and current resource usage) and hardware (current degradation and temperature of each core). As shown in Figure 45, this is possible due to the BarbequeRTRM, which gathers this information from multiple sources:

- the Runtime Library that is linked by applications automatically notifies the BarbequeRTRM about applications statistics such as current CPU usage and satisfaction with the current CPU bandwidth budget;
- the fault prediction framework notifies the BarbequeRTRM about presence and entity of degradation for each CPU core;
- the on-chip sensors (if available) are used by the BarbequeRTRM to periodically retrieve temperature values for each core.

The resulting thermal and degradation maps are used to compute an ideal resource mapping for the running applications.

Resource allocation policy validation

To validate our resource allocation approach, we performed a set of experiments using multi-threaded applications from the PARSEC benchmark [108]. We used a NUMA system composed of two nodes, each one featuring 126 GB of RAM and an Intel Xeon E5-2640 v3 Processor. We turned off the hyper-threading, and, given that this work does not deal with frequency scaling, we set the `cpufreq` frequency governor to *performance*. Overall, the system consisted of 16 cores operating at 2.6 GHz and 252 GB of memory. The system ran the CentOS (version 6.7) Linux distribution, kernel version 3.18.29. Having disabled hyper-threading, there were a total of 16 unused hardware threads; we used them to host the Operating System and the resource manager.

Regarding applications, as already mentioned, we selected a subset of the PARSEC benchmark [71] that represents computer vision, video encoding and image processing scenarios. The applications are: *bodytrack*, which uses video streams from multiple synchronized cameras to track human bodies; *facesim*, which computes the animation of a face by simulating the underlying physics; and *x264*, a H.264 video encoder. All the applications feature a simple Configure phase, where the number of threads is changed according to the number of allocated cores.

To have an idea of their behavior, we executed each application separately, allocating it all the available cores. As shown in Figure 46, *bodytrack* features a quite constant throughput throughout its entire execution. Similarly, *facesim*, apart from some low-throughput cycles at the beginning of the execution, also features a quite constant (but

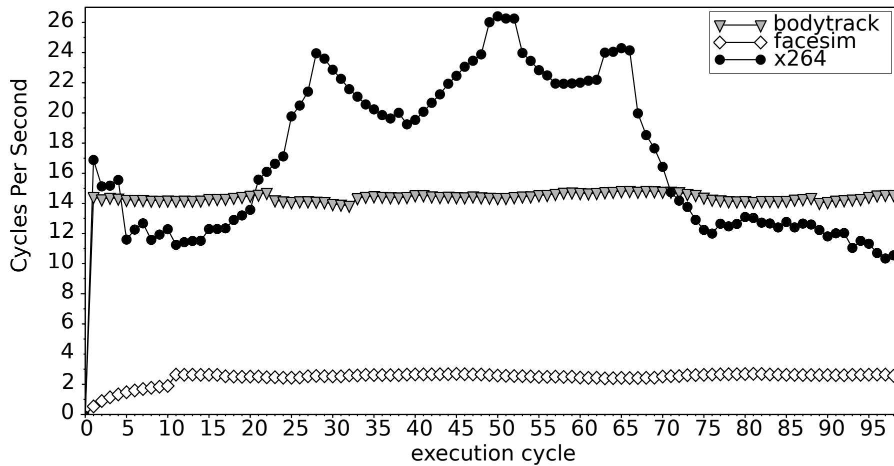


Figure 46: Maximum throughput of the applications over 100 execution cycles.

very low) throughput. On the other hand, due to the difference between the encoding of I, P and B-frames, *x264* features a fast-changing throughput.

To improve the plots readability, we defined three performance levels for each application: *high*, *medium* and *low performance*, roughly meaning 75%, 50% and 25% (±5%) of the maximum average throughput, respectively. In some test, therefore, we will express throughput in terms of performance level instead of CPS.

We set up three sets of tests: in the first set, we evaluated the overhead introduced by resource management on the execution of applications; in the second one, we executed applications one at a time to show that the resource manager allocates to applications only the resources that they need to reach the requested performance level; finally, in the third set, we concurrently executed multiple applications, thus showing how the resource manager fares in multi-application and multi-priority scenarios.

Due to the asynchronous execution of resource management-related activities (budget allocation happens while the application is running), the resource manager itself does not induce overheads on running applications. There are only two sources of overhead that hinder the execution of managed applications: writing into the Control Group configuration and locally managing the application execution at runtime library level.

Regarding the Control Groups, the overhead is more or less constant and known: on the target machine, it floats around 1 millisecond. Conversely, the overhead introduced by the runtime library is not constant: it may differ from en execution cycle to another, depending on the status of the application and on whether the runtime library communicates with the resource manager.

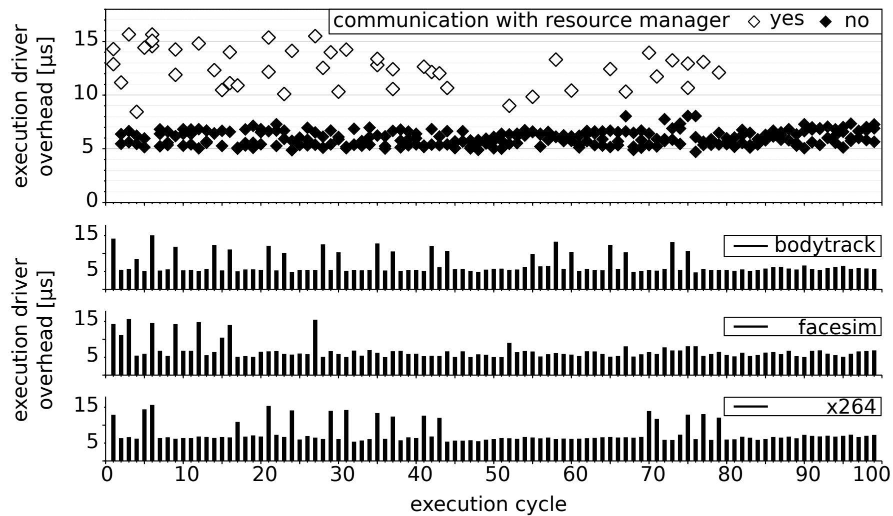


Figure 47: Runtime Library overhead [µs] over 100 execution cycles. Results are first aggregated, then shown separately for each application. In the aggregated view, white diamonds indicate that there was a communication towards the resource manager.

To evaluate the runtime library overhead, we executed each application for 100 execution cycles, i.e., each application processed 100 chunks of data. To compare the time spent processing data with the time spent executing the management logic, we evaluated the overhead separately for each cycle. Moreover, to perform a fair comparison between managed and unmanaged application, we considered the execution time of the Monitor and Configure stages as a runtime library-related overhead. Finally, we forced the applications to ask for a new resource budget multiple times over time, thus showing the additional overhead when the runtime library communicates with the resource manager.

Figure 47 reports the results of the aforementioned experiment. To facilitate the interpretation of the results, we decided to arrange the traces both in an aggregated view and separately. Unsurprisingly, the runtime library overheads are similar for all the applications; in fact, apart from the Monitor and Configure stages, the executed code is application-agnostic. When not communicating with the resource manager, the overhead of the runtime library ranges from 4 to 8 microseconds. On the other hand, when communicating with the resource manager, the overhead doubles, thus ranging from 8 to 16 microseconds.

To understand whether such overhead is reasonable, one must compare it to the duration of the Run phase. This usually depends on the application: tens of milliseconds for interactive applications, hundreds of milliseconds or more in the case of batch ones. We can state that even in the worst case scenario, i.e. an interactive application subject to the maximum overhead at each single cycle, the runtime

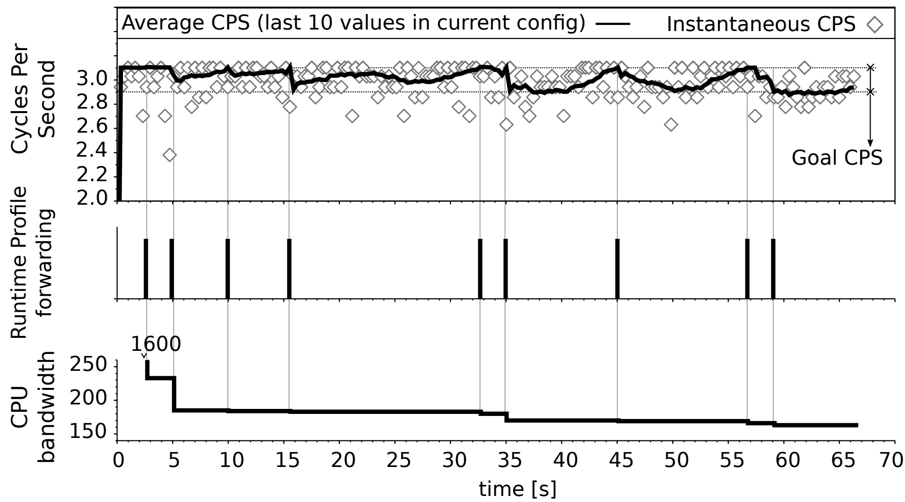


Figure 48: Managed application (*bodytrack*) forwarding runtime profiles over 200 cycles. The goal is 2.9 to 3.1 Cycles Per Second. Average CPS is computed on the last 10 values of the respective fixed-configuration loop.

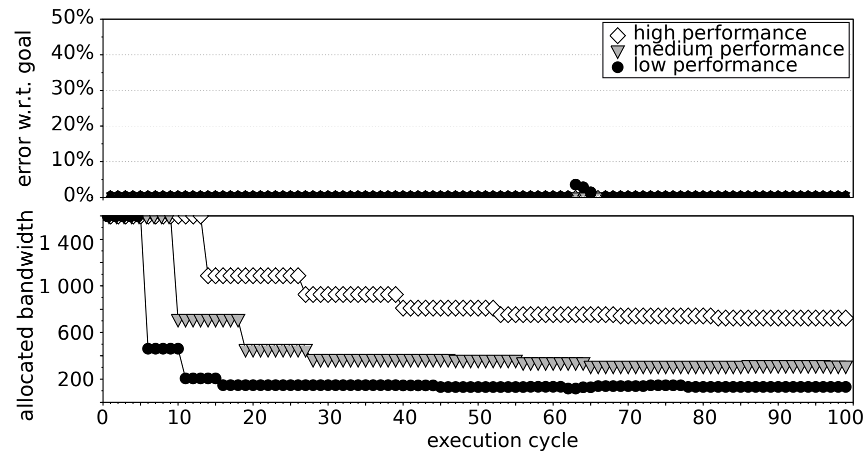
library overhead is in the order of 0.1%, hence negligible with respect to the execution time of the Run phase.

The second experiment, whose results are shown in Figure 48, evaluates the frequency at which the runtime library communicates with the resource manager. We executed *bodytrack* for 200 cycles, with a CPS goal ranging between 2.9 and 3.1 CPS. The central part of the Figure highlights the execution cycles where the runtime library forwards the application runtime profile to the resource manager, thus asking for a change in the CPU bandwidth allocation (lower part of the Figure). Please note that, even if the initial resource budget is over-provisioned, the runtime library constraints the application bandwidth; therefore, CPS is never higher of its upper limit. This minimizes interference between applications: over-assigning CPU bandwidth to an application cannot harm the performance of the rest of the workload.

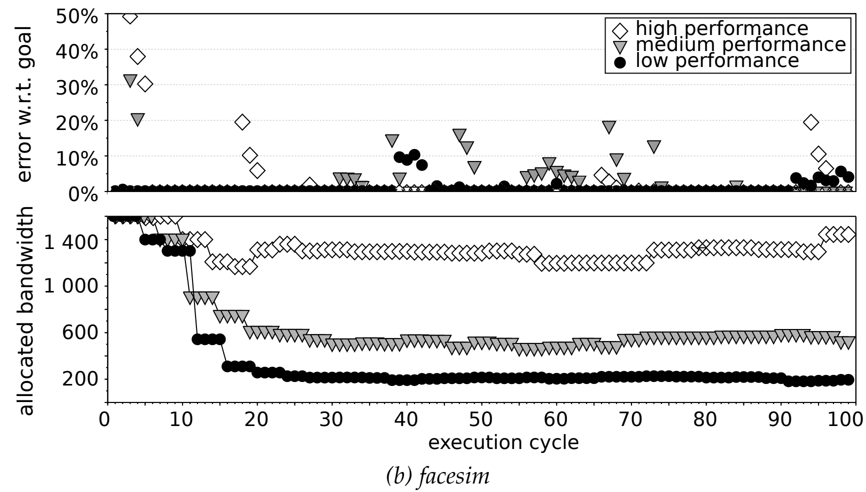
Let us now present the single-application scenarios. We executed each application for 100 cycles, in the three configurations: *high*, *medium* and *low* performance. For each execution cycle, we observed the CPS goal gap (error percentage between real and required CPS) and the amount of allocated CPU bandwidth.

In this experimental scenario, the runtime library communicated with the resource manager about 4.5% of the cycles.

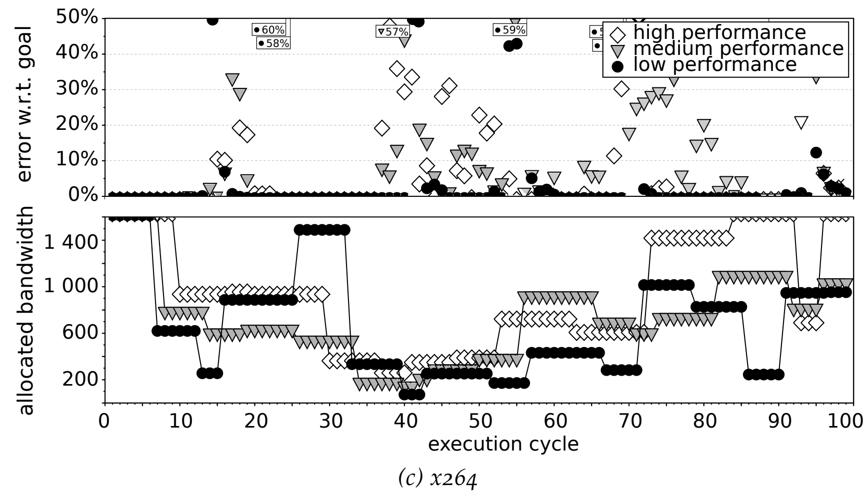
Figure 49 presents the results for the three benchmarks. As expected, *bodytrack* (a), which is the most regular application, is characterized by very small errors. That is, the CPS is almost always in the desired range, while the allocated bandwidth is gradually reduced. *facesim* (b) also shows a regular behavior, with the allocated CPU bandwidth assessing on a constant value, apart from little fluctua-



(a) bodytrack



(b) facesim



(c) x264

Figure 49: Error with respect to performance goal (upper part of each subplot) and CPU allocation (lower part) over 100 execution cycles, for the three configurations *high*, *medium* and *low* performance of each application.

tions. Please note that, due to the fact that the first cycles of *facesim* are characterized by a low throughput regardless of the allocation, the corresponding error is (as expected) always high. Finally, *x264* also confirms the expected behavior: due to the fast-changing bandwidth requirements, it forces the resource manager to change the allocation multiple times.

We then executed each application with a runtime-variable CPS goal: *high performance* during the first third of the execution, *low performance* during the second third, and *low performance* until the end of the execution. We show the results in Figure 50. Apart from the unavoidable errors due to the slow start of *facesim* and the fast-changing requirements of *x264*, the performance of the applications is always close to the required level.

Finally, we evaluated the compliance of applications with their performance goal when concurrently running in a workload. We performed two experiments. In the first one, we analyzed the performance of *bodytrack* while co-running with *facesim* and *x264*. We gave *bodytrack* an high (BarbequeRTRM-side) priority, and this caused the resource manager to make it comply with its goals despite the fact that both the other applications requested a *high* performance level. To simulate a realistic scenario and analyze the policy reaction to new applications entering the system, the arrival time of the three applications differs.

We show the results in Figure 51: the execution of *bodytrack* (second 6) forces the performance of *facesim* to drop. *x264*, which starts afterwards, is also constrained by the execution of *bodytrack*, especially when the latter requests a higher performance level (second 21). Despite the high demanding workload, the resource manager always guarantees *bodytrack* the required amount of CPU bandwidth, thus making it comply with its runtime-variable performance goal. In presence of performance fluctuations due to inherently shared resources (e.g. last level caches), the resource manager immediately refines the allocation to address the issue. This can be seen at second 15, when *x264* enters a high throughput phase, thus causing last level cache thrashing and an increase of memory bandwidth; in this case, the resource manager immediately seizes some CPU time from *x264* and lends it to *bodytrack*, which is therefore able to return to its goal-compliant CPS range. Overall, this test shows that BarbequeRTRM allows applications to be properly isolated without the need of employing virtualization environments. Moreover, the applications performance results to be stable even if resource usage is computed locally to applications by the runtime library, which is not aware of the system-wide work load.

In the second test, we analyzed the performance of the three co-running applications; this time, each application has the same priority. We selected a *medium* performance level for *facesim* and *x264*, and a

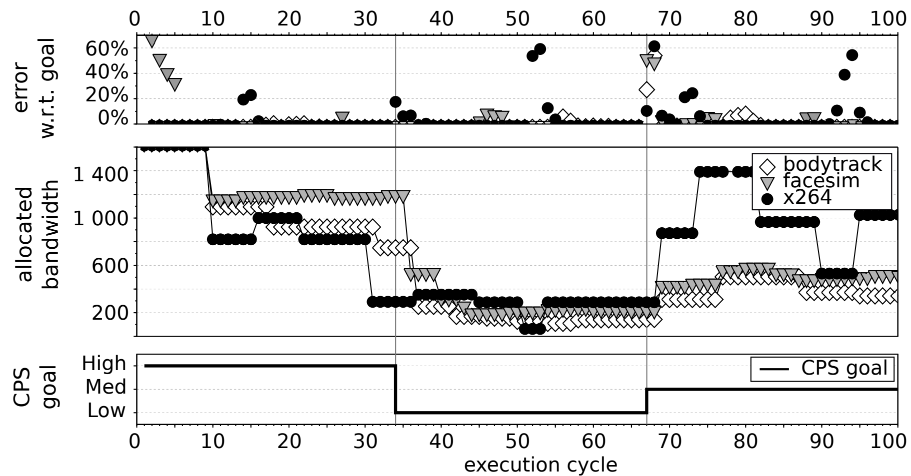


Figure 50: Error with respect to performance goal (upper part) and CPU allocation (central part) over 100 execution cycles, under runtime-variable performance requirements (lower part).

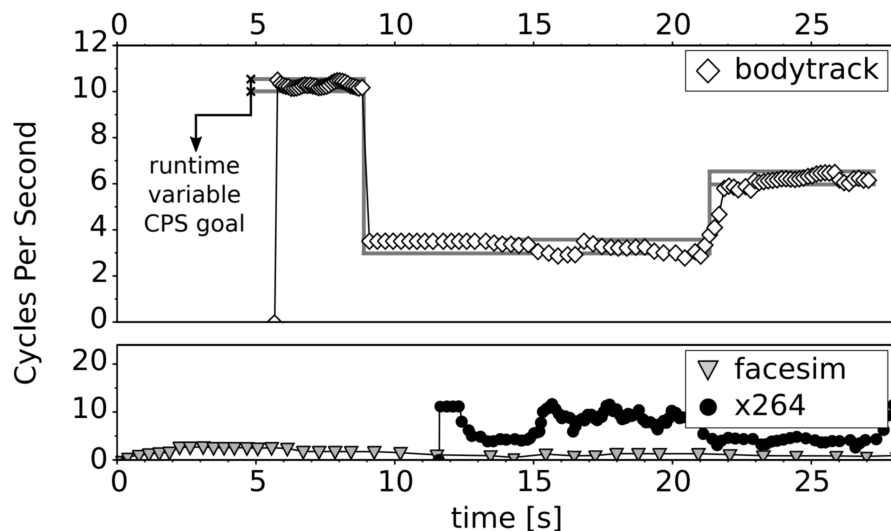


Figure 51: CPS over time for the three concurrently running applications. *bodytrack* has the higher priority and runtime-variable performance requirements, while *facesim* and *x264* have a lower priority and high performance requirements.

low-to-high performance level (5 CPS) for *bodytrack*. In this way, we obtained a performance requirement that could not be always satisfied by the available resources. This allows us to show: *1*) that the resource manager make the applications comply with their performance goal even in a multi-application scenario; and *2*) that applications with the same BarbequeRTRM-side priority equally suffer when there are not enough resources to satisfy them all.

We show the results in Figure 52: once again, apart from the slow start of *facesim* and the structural oscillations of *x264* performance, the

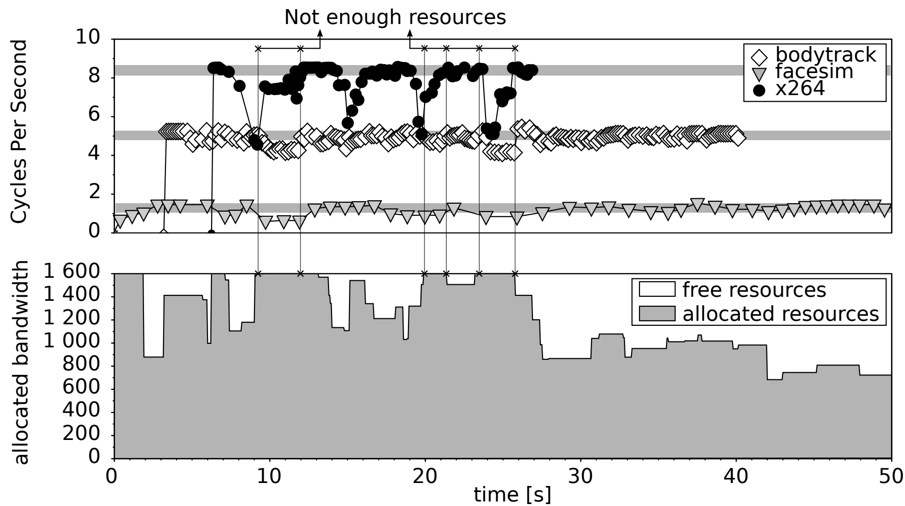


Figure 52: Cycles Per Second and allocated CPU over time for the three co-running applications. Each application have its own performance requirement, and the system resources are barely able to satisfy all the applications.

required performance levels are satisfied. In three cases, the system was not able to satisfy the demand of all the applications. In that cases, being all the applications characterized by the same priority, each application suffered in the same fashion: their distance from the performance goal was roughly the same in percentage. The lower part of the figure indicates the free resources: 100 units of free bandwidth means that one core was not allocated. Thanks to the Linux Control Groups, we are sure that that cores were really unused; therefore, it would have been possible to put them in idle or even off-line, or to use them to round-robin the allocations and mitigate thermal hot-spots, with evident system-wide benefits. Given that the applications complied with their performance goals, we are sure that the free cores were definitely not needed to satisfy the workload demands.

Resource mapping policy validation

In order to evaluate the thermal management capabilities of PerDeTemp, we employed the Rainfall-Runoff (RR) model that we already presented in Subsection 6.1. Just to recap, the model is part of the Floreon+ system [147]. The model predicts the water discharge levels of a geographical area by analyzing the recent precipitations information, whose accuracies it projects on the output by constructing confidence intervals using the Montecarlo (MC) method. For each area, there are three flood warning levels: *Flood watch*, *Flood warning* and *Flooding*.

FLOOD WATCH
The water exceeded the first warning level; there is not a strict

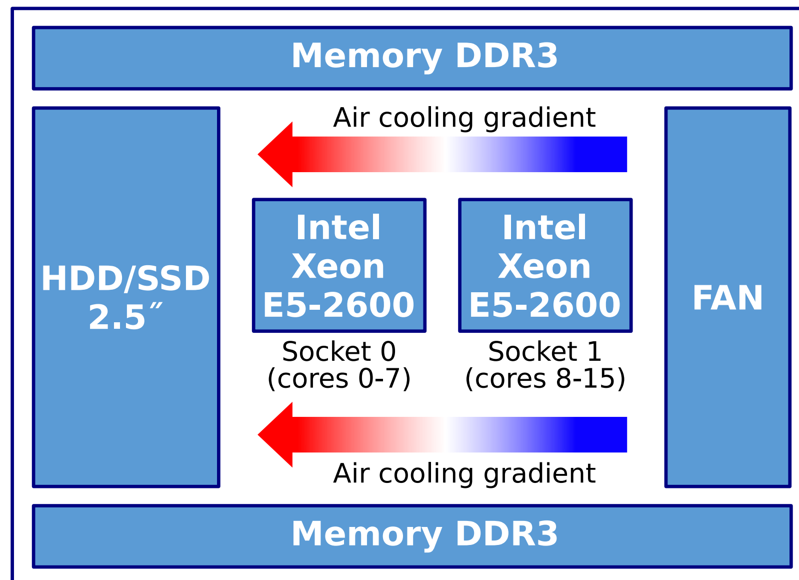


Figure 53: Schema of the blade. Since the fan is on the right of the blade, there is an air cooling gradient between socket 1 and socket 0.

need of increasing accuracy, but the water level information will be checked more frequently. The model must perform 1250 MC iterations every 10 minutes.

FLOOD WARNING
The water exceeded the second warning level; in order to correctly predict a potential flooding, the model must be quite accurate. The model must hence perform 2000 MC iterations every 10 minutes.

FLOODING
The water exceeded the third warning level; in this case, the model output must be very accurate. The model must perform 3000 MC iterations every 10 minutes.

We were not able to use the same computing system of the previous work. We instead used 16 nodes of an HPC cluster. The nodes, which are connected through InfiniBand, are part of the Anselm system [157]. Each node is a powerful x86-64 computer equipped with 16 cores (two eight-core Intel Sandy Bridge processors), with 64GB RAM and a local hard drive. Figure 53 shows a simplified representation of the blades air cooling system. Given that the fan is not equally distant from the two sockets, one socket is better cooled than the other one. When the system is idle, the temperature difference between the two sockets is approximately 10 Celsius degrees. The system has several monitors tools installed like power meters, *ganglia* [158] and *likwid* [70].

We modified the RR model so that, by using a hybrid OpenMP and MPI approach [159], it could distribute the computation among

Table 10: For each experiment, number of Monte Carlo samples to be performed by the instance that models each catchment.

Experiment	Thousand of MC samples to perform			
	C_1	C_2	C_3	C_4
α_1	1.5	1.5	1.5	1.5
α_2	1.5	3.5	5.0	7.0
α_3	7.0	7.0	7.0	7.0
β_1	3.5	3.5	3.5	3.5
β_2	7.0	7.0	7.0	7.0
β_3	3.5	7.0	12.0	15.0
γ	80.0 between all catchments			

multiple nodes. Given that the performance requirements of the application are time-variable (e.g., they are low when sunny, intermediate / critical when rainy), in a real scenario, the HPC center may allocate to the application only some computing nodes and use the remaining ones to execute other applications. Therefore, we performed our experiments in three different configurations: in the first one, we used only one node (i.e., Flood watch); in the second, we used two nodes (i.e., Flood warning); in the third, we used all the cluster (i.e., Flooding).

Table 10 presents the set of experiments performed in the cluster. The experiments tagged α refer to the single node scenario, while those tagged with β and γ respectively refer to the dual node and entire cluster scenarios.

Figures 54.a.1, 54.b.1, 54.c.1 and 54.d.1 show the number of cores allocated during the 10 minutes execution for each catchment, while Figures 54.a.2, 54.b.2, 54.c.2 and 54.d.2 show the perceived satisfaction of applications (the closer to 100% is the satisfaction, the better).

As shown by the experimental results, the number of allocated resources gets higher as the number of samples of MC samples to be performed increases. The allocation of resources is satisfactory for α_1 and α_2 scenarios, but not for α_3. As shown by Figure 54.a.2, the performance is below 100%, meaning that the resources required for this experiment are not enough for the application to reach the desired quality level. In this situation, a second node should be allocated.

Similarly, Figure 55 shows the results for the dual-node configurations. Please note that β_2 has the same computational workload of α_3; however, since in this case we have two computing nodes at our disposal, the computation is performed without issues. Similarly to

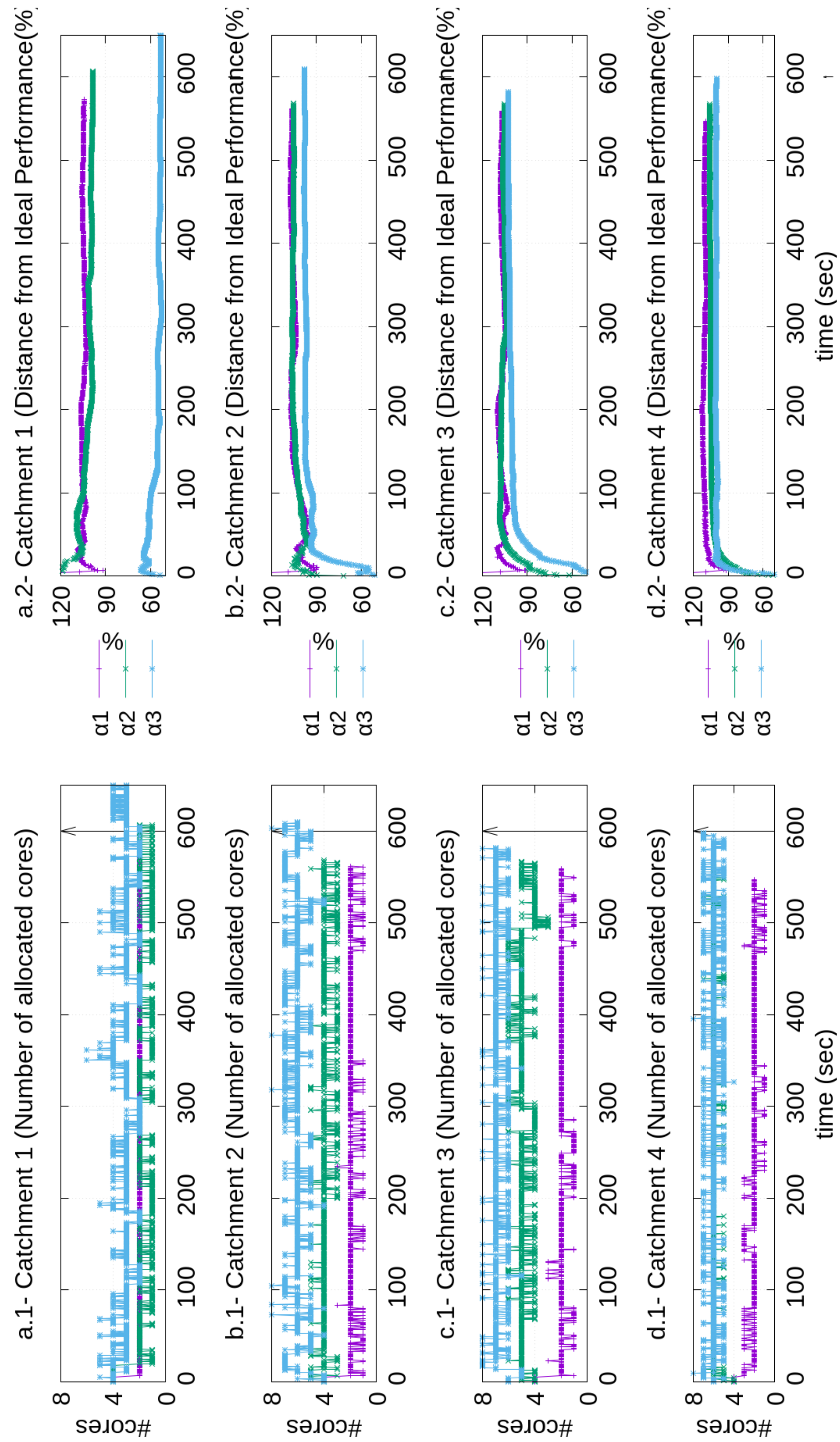


Figure 54: Results for α scenarios, listing number of allocated cores and level of satisfaction (the closer to 100%, the better).

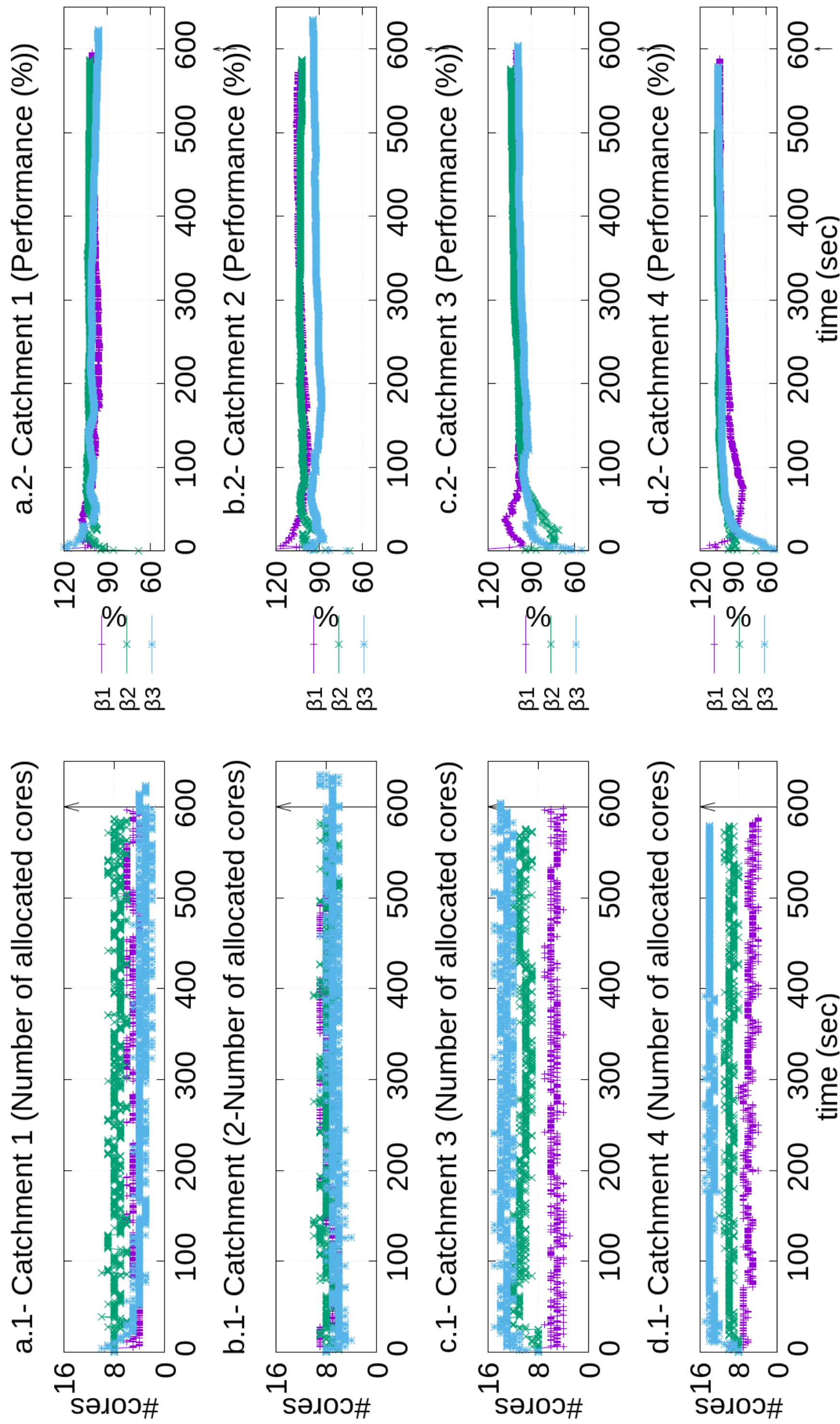


Figure 55: Results for β scenarios, listing number of allocated cores and level of satisfaction (the closer to 100%, the better).

the single-node experiments, we can also observe that the available resources are not always enough to serve the resource demand: the performance of β_3 (Figure 55.b.2) is below 100%.

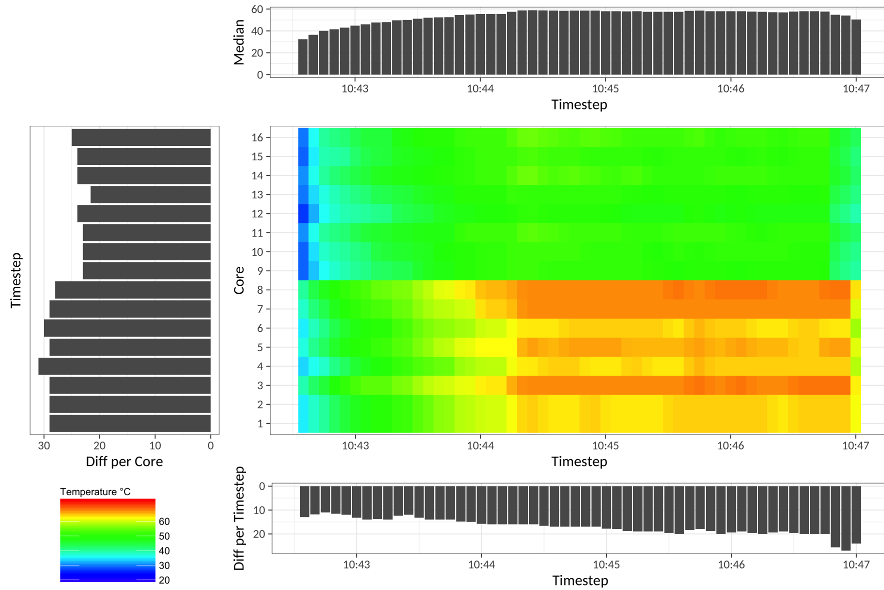
In the last experiment of this set, we executed the γ scenario on 16 nodes with a requirement 80K MC samples. Also this time, all the instances achieved an optimal satisfaction. We repeated the experiment without the HARPA-OS support, and we monitored the system power consumption by using the *likwid* power monitor. Whereas the HARPA-OS scenario led to a maximum power consumption of 100W, the unmanaged scenario led to a peak of 160W per node. Therefore, we saved around 44% in maximum power consumption while nonetheless complying with the deadline.

Figures 56a and 56b present the heat-map of a single node when running an uncertainty module instance (12K MC samples) with and without HARPA-OS (with PerDeTemp) scenarios. We obtained the heat-maps by using the ganglia monitoring tool. In both figures, the X and Y axes represent time and the IDs of the processing cores, respectively. The *Diff per Core* metric (see left part of the figures) is the difference in temperature per each core c_i ($0 < i < 16$). The *Diff per Timestep* (lower part of the figures) is the difference in temperature among all cores given a timestep. The *Median* (upper part) provides the median temperature per timestep. As can be seen in Figure 56a, there is a hotspot in socket 0, which, as already shown in Figure 53, is the socket that is farther from the fan. Conversely, Figure 56b shows the execution with HARPA-OS-PerDeTemp. In this case, there are not hotspots and, thanks to the temperature-aware tasks migration performed by PerDeTemp, the temperature is perfectly distributed throughout all the available processing elements.

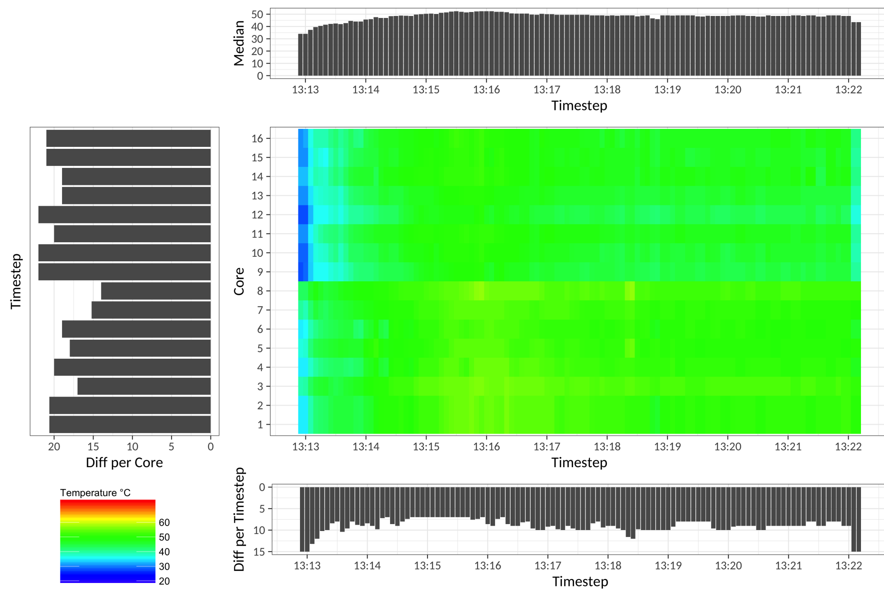
The presence of hotspots is known to have negative effects on the Mean Time Between Failures of the system (MTBF), and the aging acceleration factor depends on the difference (Δ) of temperature. According to the MTBF estimation presented in [160], running the application in a HARPA-OS PerDeTemp configuration improves the reliability of the system from 17% to 43% in case of bad cooling (socket 0). With a better cooling (socket 1), the MTBF for the best-effort relatively grows, but it is still 11% to 30% better if we manage the system with *PerDeTemp* instead of using an unmanaged approach.

Conclusion

In this Section, we presented a resource management approach that is composed of a quality-aware allocation policy and a resource-aware mapping policy. The allocation policy relies on a partially decentralized resource management and leverages the concept of application runtime profile to minimize the resource usage of applications while allowing them to comply with their Quality of Service requirements.



(a) Plain Linux.



(b) HARPA-OS-managed Linux.

Figure 56: Heatmap of the computing node in managed and unmanaged configuration. In both experiments, we used the cpufreq performance governor.

Minimizing the amount of allocated resources allows the resource manager to have a set of unallocated resources at its disposal. In this regard, the resource mapping policy uses the unallocated resources pool in order to isolate faulty and hot cores, so that: *a*) the effects of performance variability on applications are minimized; and *b*) heat is evenly spread among the available processing cores even in case of a non-homogeneous cooling system.

7 TOWARDS A SUITABLE RESOURCE MANAGEMENT SUPPORT FOR DEEPLY-HETEROGENEOUS HPC

The performance/power efficiency wall is one of the major challenges that must be faced by modern HPC systems. This issue can be tackled by leveraging heterogeneity: the closer a computing system matches the structure of an application, the most efficiently the available computing power will be exploited. It follows, then, that enabling a deeper customization of architectures is the main pathway towards computation power efficiency.

The contents of this chapter are partially submitted in a Transaction. They are currently under review.
You may want to consult Appendix A before venturing forth.

In this section, we present our novel contributions in the context of the MANGO European project [161], which will terminate in late 2018.

The MANGO project aims at developing heterogeneous accelerators for HPC systems with requirements targeting performance, power and predictability. The research investigates the architectural implications of the emerging requirements of HPC applications, aiming at the definition of new-generation high-performance, power-efficient, deeply heterogeneous architectures with native mechanisms for isolation and quality-of-service.

If you are interested in the MANGO project, please visit our website: `www.mango-project.eu`

OVERVIEW

High-Performance Computing is quickly evolving at the hardware, software and application level.

From the hardware point of view, heterogeneity is emerging as a dominant trend to boost performance and, most importantly, the performance per watt ratio, as shown by the dominance of such heterogeneous architectures in the Green 500 [162] and Top 500 [163] lists.

From the application point of view, new classes of applications are emerging, as HPC is now regarded as a valuable tool beyond the traditional application domains of oil & gas, finance, meteorology and scientific computation.

Finally, from the software point of view, the push towards cloud HPC [164] follows the hardware and application trends, aiming at providing computational resources to classes of users that could not afford them in the past. In this context, applications that are time-critical—such as financial analytics, online video transcoding, and medical imaging—require a predictable performance. Unfortunately, this is at odds with the need to maximize resource usage while minimizing power consumption.

As of today, regardless of the architecture type and scale, users and system needs seems to be consistently orthogonal.

Extending the traditional optimization space, the MANGO project aims at addressing what we call the PPP space: *power*, *performance*, and *predictability*.

In this scenario, the objective of MANGO is to achieve extreme resource efficiency in future QoS-sensitive HPC through an ambitious cross-boundary architecture exploration. The MANGO project investigates the architectural implications of the emerging requirements of HPC applications, aiming at the definition of new generation high-performance, power-efficient, deeply heterogeneous architectures with native mechanisms for isolation and QoS compliance.

MANGO follows a disruptive approach that challenges several basic assumptions, and it explores new many-core architectures that specifically target High Performance Computing. In particular, it focuses on deeply heterogeneous architectures, where multiple accelerators coexist and can serve either multiple concurrent applications or a single application composed by multiple kernels. The former scenario is the most critical, inasmuch as it requires to allocate resources to multiple applications in a way that maximizes resource usage but also preserves the predictable execution time of critical applications.

In a MANGO computing node, heterogeneous accelerators share the same memory. This makes the MANGO architecture a heterogeneous NUMA architecture.

We now highlight the *technical and scientific challenges* that will need to be tackled at runtime software level as heterogeneity increases its presence in the HPC field. Indeed, we will also provide an outline of the proposed solutions.

Resource Management

The biggest challenge for heterogeneous resource management is to optimize resource allocation while taking into account that:

- each application may be composed by multiple tasks, each of them possibly having data and timing dependencies with the other ones;
- executing a task on different computing units of an heterogeneous architecture does lead to different throughput, QoS, and power/energy consumption;
- especially in case of data dependencies, the performance of an application depends not only on where its tasks are executed, but also on where the data of the tasks is located in the system;
- requirements coming from each application (performance/QoS) must be complied with while also addressing the system-wide (power/thermal/energy) requirements.

To address this very complex problem, we move the hardware-aware logic from the application source code to the resource management layer. By doing so, we will provide the managed applica-

tions with a resource-agnostic view of the available resources. Application developers will focus on what must be done—optimizing the implementation of their algorithms and describing the inter-task dependencies—and how it should be done—defining throughput and QoS requirements but also providing the resource manager with some meta-data about the tasks that will be executed. On the other hand, the resource manager, which has a system-wide view of the available resources and the current workload, will optimally allocate the tasks and their data while making both applications and hardware comply with their requirements.

Heterogeneous computing is seldom resource-agnostic, as application developers tend to manually select a device and to configure the kernels accordingly.

Memory Management

Taking into account inter-task data dependencies is of paramount importance even at memory management level: tasks executing on different parts of an heterogeneous architecture want to be close to their data in order to minimize communication overheads; hence, memory and resource management are indeed tightly coupled. The proposed approach consists in interfacing resource and memory managers, so that the memory manager will serve memory requests in a resource allocation-aware fashion.

In the context of the MANGO project, the BarbequeRTRM is acting as a middleware between applications and memory manager. This enables a resource-aware memory allocation.

Programming Model Support

At programming model level, the challenge is to support programming models across a wide range of different accelerators. The proposed approach consists in adopting an intermediate runtime support that exposes basic features which, while per se not sufficient for the application programmer needs, easily map on the hardware features which are common to all the accelerators (i.e., those provided by the communication architecture). The intermediate runtime support must expose basic tools for communication, synchronization and task spawning. Higher level models will then be built over the intermediate model, and each accelerator will also be able to expose its own specific primitives, thus providing the programmer with all the features that are needed to achieve performance.

This kind of approach is motivated by the need of exposing to the application developer an effective and efficient management support for the deeply heterogeneous accelerator resources developed within the MANGO project [165]. An example of the MANGO computing system is shown in Figure 57. Such architecture, which consists in a mix of general purpose nodes (GN) and homogeneous nodes (HN), allows us to explore a wide number of scenarios: from single to multi-node, from homogeneous to heterogeneous.

To reach exascale parallelism, the programming model needs to be hierarchical much like the runtime management system. Traditionally,

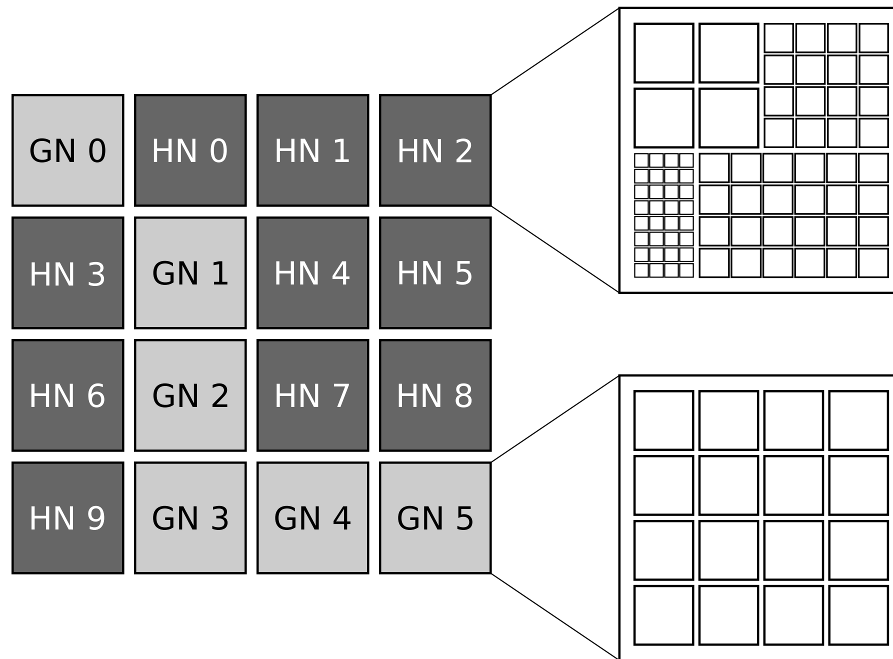


Figure 57: The MANGO architecture is composed by a mix of general purpose nodes (light gray), which feature homogeneous computing resources; and heterogeneous nodes (dark gray), which feature heterogeneous computing resources.

the programming model for homogeneous HPC systems is based on a combination of MPI and OpenMP. When heterogeneity comes into the game, however, the programming model needs to be extended to allow the exploitation of hardware resources. This can be done using OpenCL [166], which is an open standard for the development of parallel applications on a variety of heterogeneous multi-core architectures.

In OpenCL, heterogeneity is managed explicitly: the applications developer is in charge of choosing a device among the available ones and of managing the memory hierarchy. Given that even small variations in the architecture parameters may dramatically change the resulting performance, the code must be carefully optimized for the target device. Moreover, to explicitly handle heterogeneity, the programmer is forced to write high amounts of boilerplate code, which reduces the readability of the application [167, 168].

Figure 58 shows the MANGO programming model stack and its interaction with the underlying architecture and runtime software components. The programming model employs MPI to enable internode computation. At node-level, we will provide a set of language options for expressing different types of accelerators and application features. In particular, OpenCL can be effectively used to manage heterogeneity.

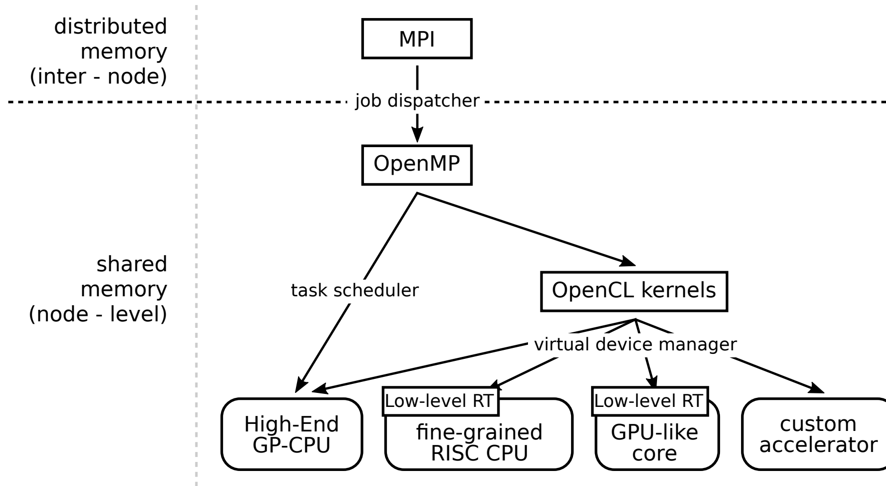


Figure 58: The MANGO Software Stack: MPI is employed to express inter-node computation, while OpenMP and OpenCL are instead used at node level to exploit the heterogeneous resources.

As already stated, one of the main goals of MANGO is to automate the selection of the best set of resources instead of leaving the choice to the application developer. The experience of the 2PARMA project [169, 170] will help in this regard. The programming model will be integrated with the runtime management facilities, thus allowing job dispatcher, task scheduler, and virtual device manager to interact with the corresponding levels of the programming model. Promising examples of the effectiveness of this approach have been already shown in the literature [171].

Since the MANGO nodes can concurrently serve multiple applications with different levels of priority, the project explores innovative techniques for analyzing the interference between multiple applications that concurrently run under QoS constraints [172, 173] (see also Chapter 3).

Regarding the resource allocation policies, we will exploit a mix of pro-active and reactive strategies. The policies will base their allocation decisions on information coming from both software and hardware side. From the application side, the off-line characterization of kernels and inter-kernel interference will be used to compute optimal resource allocation choices; moreover, the feedback coming from running applications will allow the allocation policies to dynamically adjust the resource allocation according to the real behavior of the workload (see also Chapter 6). From the hardware side, similarly, MANGO will exploit a set of custom monitoring interfaces to both investigate and implement novel resource management policies targeting heterogeneous hardware platforms and to allow the policies to perform resource-aware decisions at runtime.

Another feature that is worth to remark about the MANGO software stack is the combination between the aforementioned fine-grained

configuration and the task isolation mechanisms that are already provided by operating systems (e.g., Linux Control Groups, see Chapter 2) [34]. This will allow the MANGO runtime resource manager (RTRM) to enforce resource allocation constraints at multiple levels of the heterogeneous hardware architecture, from the general-purpose CPUs to the FPGA based computing devices, passing through the control of the interconnection infrastructure bandwidth. These mechanisms will also enable safe mixed workload executions: the MANGO run-time resource manager will be indeed capable of guaranteeing the required QoS to critical tasks while also providing space to best-effort applications.

In the following sections, we will further discuss the resource management, memory management and programming model support. For the sake of clarity, we will start from the applications side—that is, from the programming model support. Then, we will introduce the resource and memory management layers.

PROGRAMMING MODEL SUPPORT

Parallel programming models expose to the application developers a non-functional aspect of the architecture—i.e., parallelism—as a way to enable effective target-dependent optimizations starting at the algorithmic level.

There is no unified programming model, since the difference between architectures (especially those of different architectural families) is often substantial, and different models address different hardware or memory hierarchy features. However, all parallel programming models deal with the following problems:

- how to decompose a program into a set of parallel tasks;
- how to map tasks to processing elements;
- how to manage communication between tasks;
- how to manage synchronization between tasks.

Typically, the decomposition is performed by exploiting some kind of regularity in the behavior of the program: for instance, a loop which has no dependencies among its iterations can be parallelized if the index evolves with a regular pattern, such as a fixed-step increment. In these cases, we can adopt *data-parallel* programming models. When no regularities can be found, we can instead adopt a *task-parallel* model, that is, the programmer identifies chunks of code as units of parallelism, then spawns several parallel tasks to execute them.

When considering communication and synchronization, we can identify the following main classes of programming models:

MESSAGE PASSING

The programming model exposes the distributed memory structure to the programmer. Only local memory is directly accessible, and communication is achieved through the exchange of messages. MPI [174] is the most popular example of message passing library.

SHARED MEMORY

The programming model provides developers with the powerful abstraction of a global address space. This simplifies the programming effort at the expense of scalability, since maintaining the illusion of a global address space quickly grows infeasible in large scale systems. OpenMP [175] is an example of library for parallelism that supports the shared memory model.

DATA FLOW

The programming model, as in the case of Message Passing, exposes the distributed memory structure to the programmer. In this case, however, communication among executors is performed via asynchronous exchange of data through explicit channels. Languages like StreamIt [176] fall in this category.

Message Passing Model

The Message Passing programming model is typically implemented by exposing explicit *send* and *receive* primitives. In this model, the programmer needs to explicitly take into account the topology of the network and the distribution of data among processes.

Most modern Message Passing languages are influenced by the Actor model [177], where a set of reactive entities (*actors*) are addressable via unique names and are endowed with mailboxes to store incoming messages from other actors. In response to messages received from other entities, actors can send messages, spawn new actors, or modify their reactions to future messages, according to their current internal state.

ERLANG AND SCALA

Erlang and Scala [178, 179] are examples of languages that implement actor-based concurrency. In Erlang, actors are identified with (lightweight) processes that are addressed by pid. They employ a `spawn` primitive to create new processes. Messages can be sent asynchronously from a process to another through a `send` primitive, which takes the message and the destination address as parameters. A synchronous `receive` primitive can be used to retrieve messages from the mailbox, optionally using pattern matching to wait for a specific type of message. Scala also implements the actor model; it maps actors to Java threads, but extensions have been proposed to support synchronous communication [180].

MESSAGE PASSING INTERFACE

MPI is a portable, efficient and flexible standard for Message Passing programming. The MPI standard [174] was first released in 1994, and, in the next decade, it underwent a significant evolution that culminated in the creation of three separate versions that support different sets of features.

Conversely to the Actor model, MPI supports both synchronous and asynchronous communication; moreover, it supports broadcast and multicast communication using *Collective Operations*. Collective operations allow communication within a predefined group of processes, called *communicator object*, which defines a virtual topology and allows complex communication patterns such as reduction and scattering/gathering.

Shared Memory Model

The shared memory programming model is usually employed to program symmetric multiprocessors. Since data resides in the same global address space, communication between executors is achieved by means of memory read and write operations; therefore, the executors typically assume the properties of threads rather than processes. Moreover, given the lack data ownership, the correct ordering of memory operations must be preserved by using synchronization mechanisms such as *locks* and *semaphores*.

OPENMP

OpenMP [181] is a well-known example of Shared Memory programming model. It supports C, C++ and Fortran, and it consists of a mix of runtime library, directives, and API. Directives are used to identify parallel code regions, which can also be nested, and fork/join points.

Similarly to the weak ordering described in [182], OpenMP implements a relaxed-consistency shared memory model. Starting with OpenMP 4.0, it is also possible to offload parallel regions to accelerators such as GPGPUs.

CILK/CILK++

Cilk/Cilk++ [183] extends the C/C++ language with keywords that support the definition and execution of parallel tasks. The keyword `cilk` is used to identify functions that can be executed either serially or in parallel; when such a function is preceded by the keyword `spawn`, its execution is delegated to a new thread that is spawned when the function is called. The `sync` keyword forces a thread to wait for the termination of spawned functions.

Similarly to OpenMP, Cilk implements a relaxed consistency shared memory model. To ensure that all memory operations must be committed at a given point of the code, Cilk provides a primitive called `Cilk_fence()`. An `abort` primitive can be used to force the abrupt termination of threads, thus supporting speculative execution. Similarly to the dynamic loop scheduler featured by OpenMP, the task scheduler in Cilk uses the *work-stealing* policy to balance the workload across the executors.

The term "work-stealing" refers to the practice of adopting a pool of worker entities, each of whom is endorsed with a job queue. A worker whose queue is empty is allowed to "steal" works from the other queues.

INTEL TBB

Intel TBB is a C++ template library intended to support task parallelism on multi-core processors. Tasks are scheduled using the work stealing policy as in Cilk. The library features several specialized parallel constructs like `parallel_for` or `parallel_reduce`, concurrent containers, scalable memory allocators, mutual exclusion primitives, and a task scheduler.

PARTITIONED GLOBAL ADDRESS SPACE

PGAS is a parallel programming model that implements task parallelism with a shared memory model but assumes a logical partition of the memory. Each portion of the memory is considered local to each process, so that application developers are able to optimize the distribution of code and data by exploiting memory locality.

Fortress [184], X10 [185] and High Performance Fortran [186] all share a similar set of primitives for data distribution and task-data locality. Fortress makes use of a tree hierarchy of regions to describe the underlying architecture, whereas X10 only provides a flat view of the processor space.

Data Flow Model

Data Flow models are data-centric: they allow the programmer to define a data-flow graph where nodes represent operations (simple or complex as needed) and arcs represent communication channels. Labels on the arcs define the data rate, i.e., the computation in the target node is triggered only when the specified number of data items is collected in a channel.

STREAMIT

StreamIt is the most popular example of streaming language [176]. A StreamIt application is composed by filters that can be connected in cascade to generate a *pipeline*. Special nodes called *splitter* and *joiner* allow data streams to be parallelized.

Hybrid Models

It is also possible to combine multiple programming models. This is often done with MPI and OpenMP [187] in order to mitigate load imbalance and scalability problems in the case of architectures that expose both shared and distributed memory (e.g. distributed systems).

MPI is usually employed for inter-node communication purposes. At node level, each MPI process employs OpenMP, which is indeed more suited for non-distributed computations.

STARSS

StarSs is a programming framework that combines the task parallel and data flow models. It is developed at Barcelona Supercomputing Center [188, 189]. StarSs targets multi-core and heterogeneous accelerators, and, as OpenMP, it is directive based; however, in this case, directives are used to express data dependencies between tasks. Tasks are organized in a graph according to their data dependencies. The runtime system manages the resolution of dependencies, then it moves each ready task to a worker thread. Data dependencies are expressed as memory regions (via base address and size).

Heterogeneous Platforms

Especially in the case of heterogeneous platforms, programming models are essential to abstract from the architecture complexity.

OPENCL

OpenCL is a framework for programming parallel heterogeneous platforms [190]. The architecture of the typical OpenCL platform is composed by a *host* and one or more *devices*. The idea is that the host-side code orchestrates the execution of parallel jobs on the OpenCL devices.

OpenCL supports data parallelism and, to a lesser extent, task parallelism. Programmers can exploit data parallelism by invoking an explicitly parallel function (*kernel*), which is performed by an user-specified number of executors. Contrary to proprietary models such as CUDA [191, 192], OpenCL does not impose predefined limits on the number of executors; instead, it relies on a platform introspection API that allows such limits to be retrieved at runtime. This allows OpenCL to support compute devices from multiple vendors and to have multiple compute devices attached to the same host.

Every OpenCL kernel is explicitly invoked by the host-side code and executed on one of the devices. While the kernel executes, the host-side code continues ecuting asynchronously; therefore, programmers can exploit task parallelism by executing multiple serial kernels (i.e. only one executor per kernel) in parallel. OpenCL also

Table 11: Allocation and access capabilities of both host and devices on the four OpenCL memory address spaces [193].

	Global	Constant	Local	Private
Host allocation	Dynamic	Dynamic	Dynamic	None
Device allocation	None	Static	Static	Static
Host access	R/W	R/W	None	None
Device access	R/W	RO	R/W	R/W

features a specific synchronizing function, which stalls the execution of the host-side code while waiting for the kernels to terminate.

In order to abstract from the actual parallelism implemented by the hardware architecture, OpenCL features the concepts of *work-item* and *work-group*. As mentioned above, once a kernel is launched from the host-side code, multiple parallel executors are spawned on the target OpenCL device. The single executor is called work-item, and all the work-items compose a work-group. Work-items belonging to the same group always execute the same kernel; moreover, they share local memory and work-group barriers. Work-items belonging to different work-groups must instead communicate through *global memory*, which is generally mapped to an off-chip memory. In addition to the local and global memories, each work-item may access a *constant memory*, which shared by all work-items in the kernel, and a *private memory*, which is private to each work-item. Table 11 summarizes the allocation and access capabilities of both host and devices on the four OpenCL memory address spaces.

OpenCL 2.0 introduced some new features, like the support to *shared virtual memory* between host and devices, a generic address space, and the support to *dynamic parallelism*, i.e., the possibility of spawning new kernels from a device with no needs for host interaction.

OPENACC, SYCL AND C++ AMP

OpenACC, SYCL and C++ AMP [194, 195, 196] are frameworks that provide similar features, in some cases aven emitting OpenCL or CUDA code as a back-end. They attempt to provide easier-to-use interfaces by leveraging either directives or C++ features.

SKEPU

SkePU features an approach similar to that of Intel TBB: it provides a range of pre-defined, generic components implementing specific patterns of parallel programming. These “skeletons” provide a high degree of abstraction and portability, since all low level details are hidden within their implementation. Contrary to TBB,

SkePU supports multi-cores and multi-GPUs systems, leveraging native support such as OpenCL and CUDA.

The MANGO project aims at allowing developers to easily develop applications that target different types of accelerator architectures. In particular, the MANGO architecture will employ three types of accelerators: symmetric multiprocessors, which are characterized by good capabilities in terms of OS support and execution flexibility (i.e., they are able to run a POSIX-compliant runtime); GPGPU-like accelerators, which are programmable but are not able to run a fully compliant POSIX runtime; and hardware accelerators, which do not need or support any kind of software runtime. Applications, on the other hand, may be developed either by domain experts with limited knowledge of parallel computing and programming models, or by more experienced programmers. Thus, the following requirements arise:

- supporting the use of industry-standard programming models for heterogeneous systems, such as OpenCL, while guaranteeing functional portability across different programmable accelerators, as well as host-side compatibility for all accelerators;
- supporting a simple fork-join model, on which application developers not willing to use OpenCL can map their applications;
- supporting future extensions of the MANGO software stack to support skeleton-based programming.

The low-level MANGO runtime system, therefore, needs to operate in a way that is akin to an intermediate language in a compiler: it must allow the software stack developers to easily map high-level programming models on the range of supported accelerators, while providing at least functional compatibility. Depending on the individual capabilities of each accelerator, the low-level runtime system should also introduce optimizations or additional features; this would indeed cause compatibility issues, but it would also allow developers to implement specialized versions of their applications application for any given accelerator.

Host-side low-level runtime

The *host-side low-level runtime* (HLR) provides the general purpose nodes with an interface to access the functionalities of accelerators.

KERNEL LOADING AND LAUNCHING.

The HLR exposes functionalities and data structures that can be used to represent and manipulate kernels. Kernels are stored either in memory or in external files and are processed in a unit-specific way – source code for a GPGPU-like accelerator would be

dynamically compiled, whereas a hardware accelerator may execute pre-compiled kernels. The HLR also provides developers with an interface to set the arguments of kernels and to trigger their execution.

TASK GRAPH MANAGEMENT.
The HLR API allows developers to indicate to the runtime which components (kernels, memory objects, and synchronization events) need to be shared within the heterogeneous node. These components are then connected into a task graph, thus providing the resource manager with the information needed to generate the best feasible resource allocation for the requested QoS.

COMMUNICATION AND SYNCHRONIZATION.
Finally, the HLR provides developers with functions to synchronize with the executing kernels (wait for completion) and for communication purposes. In particular, it supports two forms of asynchronous communication: a simple copy of memory objects, and a burst copy through a fifo memory object.

The code shown in Listing 2 demonstrates the use of the HLR. In this case, a FIFO communication is set up between the host and a single kernel, which is loaded as a binary from an external file. For the sake of simplicity, the kernel does not return data. The execution is not unlike that of a native kernel in OpenCL, but the resource assignment is controlled by the resource manager, which is therefore able to optimize the use of resources in a multi-application scenario.

Device-side low-level runtime

The *device-side low-level runtime* support (DLR) serves as a baseline to implement more complex programming models. It only implements the minimal functionalities that are needed to work with the heterogeneous node, which does not feature a full operating system layer.

The DLR allows developers to spawn tasks from the host-side and to wait for their completion. It also provides synchronization mechanisms at accelerator level, between different accelerators, and between the host and the accelerators. Finally, it allows the device-side to perform memory mapping of buffers that are allocated in the shared memory, generating virtual addresses for them.

The code shown in Listing 3 demonstrates the use of the DLR. In this case, which matches the HLR code shown in Listing 2, the DLR API is used to access the shared memory region, which is then read by the parallel tasks that are spawned from the main executor. Once more, for the sake of simplicity, the actual operation of the kernel and the generation of results are omitted.

Listing 2: Example of HLR use: a FIFO communication is set up between the host and the single kernel, which is loaded from an external file.

```
1  /* Initialization of MANGO library*/
2  mango_init();
3
4  /* Loading a single kernel from file*/
5  kernelfunction *k = mango_kernelfunction_init();
6  mango_load_kernel("./test_kernel_fifo", k, GN, BINARY);
7
8  /* Registration of the task graph */
9  mango_kernel_t *k1 = mango_register_kernel(KID, k);
10 mango_buffer_t *b1 = mango_register_memory(4, FIFO, 1, 1, k1, k1);
11 mango_task_graph_t *tg = mango_task_graph_create(1,1,0,k1,b1);
12
13 /* Resource Allocation */
14 mango_resource_allocation(tg);
15
16 /* Setting the kernel arguments */
17 mango_arg_t arg1 = {
18     (void *)b1->phy_addr,
19     sizeof(uint64_t),
20     FIFO
21 };
22 mango_arg_t arg2 = {
23     (void *)((uint64_t)b1->event->event_id),
24     sizeof(uint32_t),
25     SCALAR
26 };
27 mango_args_t *args = mango_set_args(k1, 2, &arg1, &arg2);
28
29 /* Data transfer and kernel execution */
30 mango_write(argv[1], b1, BURST,strlen(argv[1]));
31 mango_event_t *e2 = mango_start_kernel(k1, args, NULL);
32
33 mango_wait(e2);
34
35 /* Deallocation and teardown */
36 mango_resource_deallocation(tg);
37 mango_task_graph_destroy_all(tg);
38 mango_release();
```

Discussion

To better explain the choices that drove the selection of the primitives in the MANGO runtime support, we compare them to possible alternatives, namely POSIX and OpenCL.

With respect to the POSIX API, the set of primitives presented above is much more restricted. The reason behind this choice is that communication and synchronization primitives must be homogeneous among the different accelerators, as they all access the same memory, communication and synchronization resources. Moreover, some of the accelerators lack the ability to efficiently support more complex activities, such as context switching or control divergence. Thus, the selected primitives focus on a set of functionalities that can easily be supported in all the accelerators.

Listing 3: Example of DLR use: the DLR API is used to access the shared memory region, which is then read by the parallel tasks that are spawned from the main executor.

```
1  char *shared_memory;
2  mango_event_t *fifo;
3
4  void *task(task_args *a) {
5      int i;
6      char d;
7
8      for(i = 0; i < N; i ++) {
9          mango_barrier(a, fifo);
10         d = shared_memory[tid(a)];
11         /* do something with the data */
12     }
13
14     return mango_exit(a);
15 }
16
17 int main(int argc, char **argv){
18     int i;
19     uint64_t phy_addr;
20     uint32_t fifo_id;
21
22     mango_init();
23     mango_get_arg(argv[1], sizeof(uint64_t), FIFO, (void **)&phy_addr);
24     mango_get_arg(argv[2], sizeof(uint32_t), SCALAR, &fifo_id);
25
26     shared_memory = (char*) mango_map(phy_addr,SIZE);
27     fifo = mango_get_event(fifo_id);
28
29     mango_event_t *e = mango_spawn(&task, SIZE);
30     mango_join(e);
31
32     mango_unmap(shared_memory);
33     mango_close(15);
34 }
```

With respect to the OpenCL API, the set of primitives presented above lacks introspection capabilities; however, it can rely on the resource management support. This is a key goal of MANGO, as there is a widespread reluctance of users towards the manual selection of computational resources.

As future development, we plan to map OpenCL on our low-level runtime, thus allowing the portability of existing applications and the access to an industry standard programming model. Specifically, the runtime will report the availability of a special OpenCL device type, which will be mapped to an actual unit at runtime.

RUNTIME MANAGEMENT

The MANGO hardware architecture is based on the idea of building very energy efficient HPC systems. In this regard, several studies have remarked the opportunities offered by heterogeneous comput-

ing platforms [197]. In the case of heterogeneous architectures, resource management must address the complex problem of distributing and scheduling tasks over processing resources that are characterized by different instruction sets, different architectures, and different support for the software layer.

Addressing such a problem requires the cooperation of several actors, and, among those, the resource manager plays a key role. The resource manager can be defined as a software framework that is in charge of applying decisions in terms of task scheduling and resource allocation (also "mapping"), according to one or more objectives.

In this section, we describe our resource management approach. Subsection 7.3.1 briefly describes the most valuable state-of-the-art examples of run-time resource mangers, including the one we selected for the MANGO project. Subsection 7.3.2 details the resource management from a high level point of view, describing how the computing nodes cooperate to execute applications. Subsection 7.3.3 describes how applications must be designed and implemented to be runtime-manageable.

Run-time resource manager

The first resource manager worth considering is StarPU [198]. This framework has been specifically designed to address the problem of scheduling tasks on systems that feature heterogeneous processing devices. The proposed approach is based on the introduction of additional C-language constructs, with the consequent extensions of the compiler (GCC) in order to generate code for different architectures. The allocation policy that is used at runtime can be selected among a set of pre-defined ones. StarPU features a greedy policy that aims at balancing the workload over all the available processing devices while providing performance guarantees to applications (whenever the target of the selected policy is a real-time workload). StarPU also features an energy-based scheduler that requires each application to provide an energy consumption model. The energy-based scheduler tries to optimize energy at application level by scheduling each task on the most energy-conservative processing unit. However, even in this case, maximizing the energy efficiency of a workload while complying with the QoS requirements of each application is not explicitly considered as an objective.

SLURM, alias "Simple Linux Utility for Resource Management", is another widespread resource manager [199]. It is modular and scalable, and, Similarly to StarPU, it includes several greedy scheduling policies; however, from the hardware perspective, it focuses on Linux cluster systems. Power management can be performed by switching-off idle computing resources or explicitly setting the CPU frequencies according to the job input. Overall, this resource manager enforces

coarse-grained decisions. Moreover, to the best of our knowledge, it does not take into account the possibility of controlling the bandwidth allocation of a Network-on-Chip (NoC) in a many-core processor; hence, its capability of isolating applications in the architecture proposed by the MANGO project would be limited.

Another resource manager worth considering is Nanos++, a runtime library developed by the Barcelona Supercomputing Center (BSC) to support parallel programming environments [200]. The functionalities provided by the library range from task scheduling to throttling policies, plus synchronization mechanisms and a support for profiling. To effectively exploit the run-time library, the application must be written in OmpSs (an extension of OpenMP) and compiled with the Mercurium compiler provided by BSC. From our perspective, the main limitation of Nanos++ is that the tool aims at optimizing the execution of a single parallel application without considering the resource contention originated by the concurrent execution of multiple applications in the system.

A further and more recent proposal is SoPHy [201]. The authors describe the framework as a "hybrid resource management" software platform, in the sense that the user can specify static or dynamic task scheduling and mapping policies, or a combination of both. Concerning the target hardware, the resource manager seems to be specifically designed to control the execution of tasks on a many-core accelerator. Again, similarly to Nanos++, SoPHy seems to aim at optimizing the execution of a single parallel application.

Finally, we analyze the BarbequeRTRM (see Appendix A), a runtime resource management framework developed by Politecnico di Milano [202]. It is based on a centralized resource manager and targets run-time adaptive applications, that is, resource-aware applications that are able to dynamically tune their software parameters to better exploit the available resources. The BarbequeRTRM includes several resource allocation policies that drive task scheduling and power management decisions at the same time. Moreover, it is modular: new platforms and features can be supported without making invasive changes to the resource manager itself.

We selected the BarbequeRTRM as the reference resource manager for the MANGO project. The main idea is to use the BarbequeRTRM to manage the single computing node, while a Global Resource Manager will be in charge of distributing the applications among the nodes. MANGO will drive the development of the BarbequeRTRM in order to support highly heterogeneous multi-node HPC platforms. In this regard, we plan to extend the resource manager in order to work in a distributed fashion (i.e., making each node collaborate with the others), to design and implement novel policies for heterogeneous systems, and to enable low-level control capabilities like per-application interconnect (NoC) bandwidth reservation.

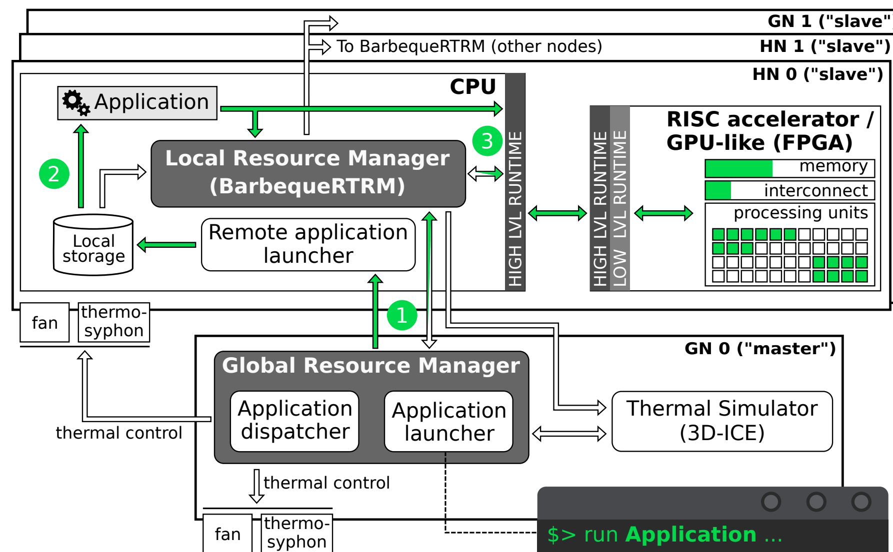


Figure 59: Hierarchical and distributed run-time resource management strategy. The Global Resource Manager runs on a "master" general-purpose node (GN), while several instances of BarbequeRTRM run on as many other nodes, general-purpose or heterogeneous (HN). The managers exchange control information and statistics about the hardware status and the application performance/QoS. Colored arrows represent the typical application launch flow.

Distributed Management

The overall resource management strategy in the MANGO project relies on the hierarchical and distributed approach shown in Figure 59.

The Global Resource Manager is at the top of the hierarchy, and it runs on a general purpose node, also referred to as the "master node". The BarbequeRTRM runs on the managed nodes ("slave nodes"), which can be either general purpose or heterogeneous. Each instance of the BarbequeRTRM is in charge of managing the resources of one slave node, thus acting as a local manager.

From a hardware perspective, the master node is equal to the general purpose slave nodes. However, it differs in the software setup: first of all, it hosts the global resource manager instead of the local one; second, it is the system entry point, i.e., it exposes a shell-like user interface. This interface, whom we call *Application Launcher*, allows the user to launch applications and to set their performance requirements. Such requirements are taken as input by the *Application Dispatcher*, which computes how many (and which type of) nodes should be assigned to the application.

To actually launch the application on multiple nodes (step 1 of Figure 59), the Global Resource Manager performs two actions for each assigned node:

1. it notifies to the local manager the IDs of *all* the assigned nodes;

2. it triggers the *Remote Application Launcher*, which is in charge of loading the architecture-specific binary and starting the execution on the node.

The first action is needed to implement a lightweight virtualization mechanism: as long as the application is running, the instances of the BarbequeRTRM from all the allocated nodes will cooperate in order to optimize its execution, thus creating a virtual "super-node" for the application. To allow that, in the context of the MANGO project, we will extend the BarbequeRTRM framework to enable distributed runtime resource management strategies.

The second action, instead, matches the runtime support provided by already known distributed parallel programming paradigms, e.g., Message Passing Interface (MPI): an instance of the application is started on each assigned node (step 2), where the local BarbequeRTRM instance will manage the resource assignment (step 3).

At slave node level, resource allocation will take into account both static information, which will be collected during an off-line analysis of hardware and applications and will be stored locally on each node; and runtime information, which will be collected from applications and on-chip sensors and sent as a feedback to the local manager. All the runtime information that is relevant to understand the status of the node will also be forwarded to the Global Manager, which, merging the information coming from all the nodes, will be able to have a system-wide view of the MANGO architecture. This view will be used by the Global Resource Manager, in cooperation with the 3D-ICE framework [203], to adapt the application dispatching and the thermal management actions to the real system response.

Developing runtime-manageable applications

From the BarbequeRTRM perspective, a managed application is a collection of independent tasks (those that have data dependencies are merged), each of whom will be allocated a private share of the available computing resources (e.g. the MANGO units).

Managed application must link to the BarbequeRTRM *MANGO Synchronization Library*, which automatizes the process of run-time management and reconfiguration. More in detail, the launch of a MANGO application creates an instance of the *Execution Synchronizer*, a C++ object in charge of:

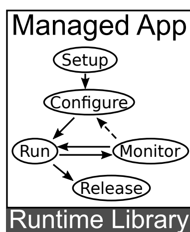
1. synchronizing the application tasks execution with the runtime-variable resource allocation;

2. profiling the task execution through user-defined metric counters;

3. transparently collecting the task performance/resource usage statistics and forwarding them back to the BarbequeRTRM, which will tune the resource allocation accordingly.

The *Execution Synchronizer* is allocated when the `mango_init()` function is called. A suitable control thread is in charge of calling the member functions of the *Execution Synchronizer* according to both the application-side events (e.g., MANGO API function calls) and the resource manager actions (e.g., resources re-allocations). The control thread is started when the `mango_resource_allocation()` function is invoked, and it drives the application execution by calling the following *Execution Synchronizer* member functions:

The execution syncronizer is itself a BarbequeRTRM application!



The BarbequeRTRM execution phases (Setup, Configure, Run...) are indeed enriched to fully exploit the capabilities of the MANGO hardware.

SETUP()
: called when a task-graph has been built. It checks the consistency of the provided task-graph, then it forwards it to the resource manager;

CONFIGURE()
: called when the resource allocation changes. The function transparently allocates the buffers on the HN memory by interacting with the *Memory Manager* (see Section 7.4) and offloads the tasks on the assigned units. Once buffers and kernels are ready, the synchronizer must simply wait for the sequence of `mango_start_kernel()` function calls. A *kernel execution control thread* is spawned to monitor the execution of each kernel;

RUN()
: this function is called once all the tasks are running. It waits for the current "step" to complete;

MONITOR()
: this function is called after each `Run()` execution. It forwards application profiling information to the resource manager;

RELEASE()
: by means of this function, the application explicitly releases the resources. This function also notifies the application termination to the resource manager. It is invoked when the `mango_resource_deallocation()` function is called.

Figure 60 provides a simplified view of the synchronization mechanisms underlying the application execution flow. The `Setup()` method is triggered when the managed application creates the task graph. When the BarbequeRTRM allocates resources to the application, the *Execution Synchronizer* can proceed by invoking the `Configure()` method. As long as the BarbequeRTRM does not change the resource allocation, the *Execution Synchronizer* will stay in the main execution loop, which is composed of a `Run()` and a `Monitor()` stage. Once the task

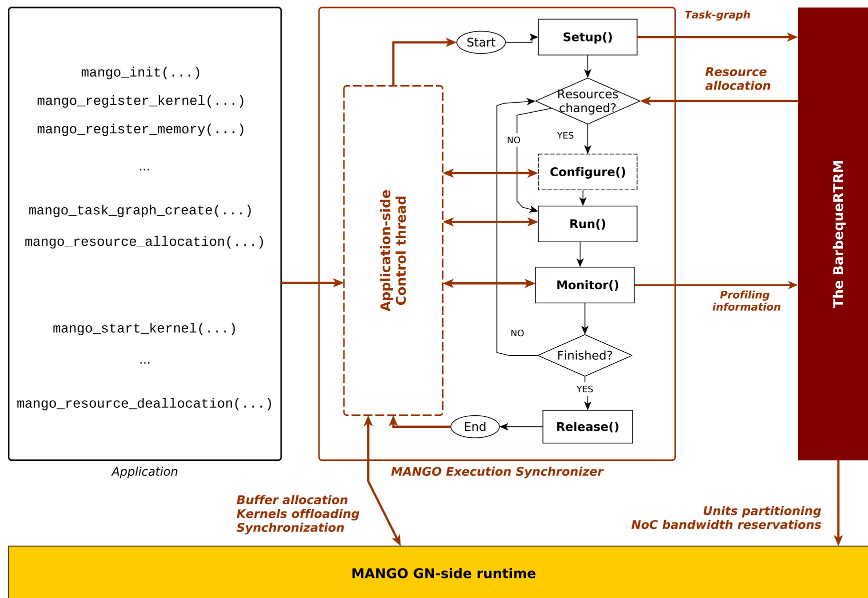


Figure 60: The synchronization mechanisms between the application execution and the resource manager (BarbequeRTRM) control actions. The MANGO Execution Synchronizer is a special BarbequeRTRM application that wraps the original application and makes it runtime-manageable.

graph is completed, the *Execution Synchronizer* invokes the `Release()` method and returns.

Profiling runtime-manageable applications

The BarbequeRTRM allows application developers to create a list of resource assignment options for their applications. This can be done by compiling a list of *Application Working Modes* (AWM) for the application (see Section A.3). Depending on the BarbequeRTRM configuration, the resource allocation of an application may be:

1. selected from the list of AWMs;
2. selected from the list of AWMs and dynamically tuned according to the runtime feedback of the RTLib (see Section 6.1);
3. Computed from scratch according to the runtime feedback of the RTLib without using Design-time Profiles altogether (see Section 6.2).

The Design-time profile of a task, i.e., the list of AWMs, is contained in an XML file called *recipe*. Listing 4 shows an example of recipe. Each task got a priority level that is used by the resource allocation policy to prioritize it with respect to the other ones. Please note that a task is referred to by using the term “application”. The

Listing 4: Example of MANGO application Recipe.

```xml
<?xml version="1.0"?>
<BarbequeRTRM recipe_version=" 0.8 ">
    <application priority="4">
        <platform id="org.linux.cgroup" hw="mango">
            <awms>
                <awm id="0" value="1" config-time="150">
                    <resources>
                        <cpu>
                            <pe qty="100"/>
                            <mem qty="2" units="MB"/>
                        </cpu>
                        <net qty="50" units="Kbps"/>
                    </resources>
                </awm>
                <awm id="2" value="4" config-time="150">
                    <resources>
                        <cpu>
                            <pe qty="200"/>
                            <mem qty="10" units="MB"/>
                        </cpu>
                        <acc>
                            <pe qty="4"/>
                        </acc>
                        <net qty=" 100 " units="Kbps"/>
                    </resources>
                </awm>
            </awms>
            <tasks>
                <task name="..." id="0" ctime="10" hw_prefs="peak,cpu,nup"/>
                <task name="..." id="1" ctime="50" hw_prefs="cpu"/>
                <task name="..." id="2" ctime="30" hw_prefs="nup,peak"/>
            </tasks>
        </platform>
    </application>
</BarbequeRTRM>
```

reason is straightforward: given that tasks do not have data dependencies by construction, the BarbequeRTRM considers each task as a distinct application. The `platform` section identifies the target system, which, in this case, is a general purpose node of the MANGO hardware that runs a Linux operating system. The set of Application Working Modes is platform-specific; therefore, the recipe format allows the developer to include several platform sections into a single recipe file. The proper section will be parsed at the application start-time depending on the actual system. The platform in the recipe example features two AWMs. Each of those defines several attributes:

1. a progressive numeric identifier (`id`);
2. a descriptive name, which is used only for logging purposes;
3. the aforementioned preference score (`value`);
4. the profiled configuration time, expressed in milliseconds (`config-time`), which keeps track of the time overhead experienced by the task to adapt to the AWM.

The <resources> section contains the resource requirements of each AWM. In the example, such requirements are expressed in terms of CPU time (cpu→pe, expressed in percent), amount of memory (mem), number of accelerator cores (acc→pe) and network bandwidth (net).

Concerning the specific case of the CPU time, the values reported must be read as percentages. Therefore, values greater than 100 simply expresses the usage requirement of more than a CPU core. Generally, a good practice would be to write an application Recipe through a suitable profiling of the application execution under different resource assignment configurations (see Section 3.3).

As already mentioned in the previous chapters, CPU time allocation is expressed in terms of CPU "bandwidth".

Especially in the case of multi-tasking applications, it is paramount to specify the tasks performance requirements to the resource manager. The <tasks> section can be used to define the expected completion time of each task. Moreover, the developer can drive the allocation choices by ordering the computing units types in order of preference. This is done by defining the hw_prefs attribute. Indeed, such performance requirements can be modified during runtime by the application.

MEMORY MANAGEMENT

Recent advances in memory allocation for homogeneous multi-core architectures aimed at removing the need for application-specific allocators and at improving scalability and allocation speed [204]; nevertheless, memory management for heterogeneous systems is still a challenge, as those systems often feature separate physical spaces for the general purpose part of the system and for the heterogeneous accelerators. Moreover, in many kinds of heterogeneous or accelerator-based systems, there is no dynamic memory allocation at all, with each accelerator endowed with its own private memory.

The work presented in [205] addresses the adoption of dynamic memory management for the design of many-accelerator FPGA-based systems, thus allowing each accelerator to dynamically adapt its allocated memory according to the runtime memory requirements. They support fully-parallel memory access paths by grouping BRAM modules into unique memory banks, named heaps, each managed by an allocator. The approach proves effective in increasing FPGA density.

Similarly, the Accelerator Store [206] framework supports dynamic provisioning of memory resources in many-accelerator architectures. Once more, the main advantage is that, since not all accelerators are in use at all times, a large amount of memory can be saved by sharing. Moreover, the ratio between shared and private memories can be fine-tuned to limit the subsequent performance penalty. In [207], global shared address spaces for heterogeneous chip multiprocessors are supported by means of specialized hardware—essentially modifications to the Translation lookaside buffer (TLB) to support dynam-

ical migration of memory pages from on-chip to off-chip memory. This work focuses on transparently supporting legacy applications on architectures where scratchpad memories are available.

Conversely to the above mentioned approaches, the MANGO architecture memories are shared at node level among all the accelerators, which makes the MANGO architecture a heterogeneous Non-Uniform Memory Access (NUMA) architecture. Such kind of architecture indeed poses several challenges, as the placement of data on different memory regions leads to substantial performance variations. In this cases, memory should be allocated as close as possible to the target execution units, while also taking into account resource allocation and data-dependencies between tasks; therefore, to efficiently allocate resources, resource and memory management decisions must be tightly coupled. Techniques based on prefetching and thread pinning have been developed to this end for use on general purpose multiprocessors [208, 209]. In the case of heterogeneous architectures, however, the literature is limited due to the above mentioned total or partial lack of shared memory. As introduced in Section 7.2, we address the problem by defining task graphs at programming model level, which allow the memory allocator to take into account both the resource allocation choices and the NUMA characteristics of the underlying MANGO architecture when allocating memory.

Another class of heterogeneous systems that is worth mentioning is that of *embedded many-cores*, such as STMicroelectronics STHORM [210] (originally known as Platform 2012). In those architectures, the available memory is shared among the processing elements, but it is usually limited, and this makes custom memory allocators the most efficient choice [211]. In these cases, simplicity and efficiency of the allocator are paramount, and the characteristics of the specific platform can be exploited by developing a parameterized dynamic memory allocator that can be fine-tuned by means of profiling or design space exploration techniques. We already explored the idea of integrating memory management in the overall runtime resource management in the context of the 2PARMA project, on the STHORM platform [212]. However, this kind of solution does not fit well with tasks that vary widely in latency and resource requests, which is not usually the case in the embedded domain, but is a common occurrence in high performance computing.

The MANGO approach

In the MANGO architecture, all the memory modules in a given Heterogeneous Node (HN) share a single physical address space that can be accessed by all the computational units (ARM-based nodes, GPU-like accelerators, and hardware accelerators).

For the proper management of memory resources, we will define a segmentation process scheme that supports the definition of private or shared segments for applications. The resource manager will manage HN memory by setting segments of variable sizes and assigning them to incoming applications. Those segments will be set at strategic locations, that is, they will be mapped on memory modules that are close to the processing elements where the application will be running. The segments will be defined as either shared or private, thus enabling the concept of shared memory between applications (if needed) and between different kernels of the application. Moreover, some of the accelerators will enforce coherent memory access of the different threads of the same kernel; this process will be steered by the QoS requirements of applications, thus enforcing partitions at the network and memory subsystem levels.

The memory management system runs on a General purpose Node (GN) and controls memory allocation on the HN. It deals with the following issues:

- choosing the most suitable memory according to the allocated processing elements;
- enabling concurrent, thread-safe memory allocation and deallocation while avoiding fragmentation;
- performing translation from virtual to physical addresses and vice versa;
- performing runtime optimization.

Choosing the most suitable memory.

In order to select the best memory modules, the memory manger will take into account the following search criteria:

- bandwidth between the allocated processing elements and the memory module;
- latency of the memory module;
- direction of data transfer (in/out);
- available space on the module;
- load on routing and ports.

More criteria could be added depending on the application requirements, which are provided by the BarbequeRTRM. Furthermore, depending on the target architecture, the bandwidth between memory and processing elements may change.

Another challenge is understanding which is the best memory module for a given kernel or group of kernels; in the absence of a single

performance criterion, we plan to employ a fuzzy multi-criteria analysis. Finally, the choice between equivalent memory banks will be based on a statistical prediction of the future memory usage.

Concurrent, thread-safe memory allocation and deallocation.

There are many implementations of the malloc function, each one with its own strengths and weaknesses. The criteria which must be taken into account when implementing an allocation algorithm include:

- speed of allocation and deallocation;
- the presence of popular-sized requests;
- utility of memory usage;
- degree of segmentation;
- thread safety.

In general, the first step of allocation is searching for free space. The most popular algorithms for memory space management are:

BEST FIT: the allocator selects the smallest block of unallocated memory that is large enough to satisfy the request;

WORST FIT: the allocator selects the largest block of unallocated memory;

FIRST FIT/NEXT FIT: the allocator selects the first block of unallocated memory that is large enough to satisfy the request, starting either from the begin of the memory or from the last selected block;

SEGREGATED LISTS: if the application has one (or a few) popular-sized request, the allocator keeps a separate list of free segments to manage objects of that size; all other requests are processed by another algorithm.

BINARY BUDDY ALLOCATOR: the allocator recursively divides the unallocated memory by two until a block that is big enough to accommodate the request is found.

In order to implement the memory management system, these algorithms need to be evaluated basing on their compliance with the aforementioned criteria. Moreover, we will evaluate the possibility of implementing flexible memory allocation algorithms that tune their configuration according to the specifics of running applications.

Runtime optimization.

When free memory becomes available, the memory manager could evaluate the possibility to migrate the data of running kernels to more suitable memory modules. This would allow the memory allocator to optimize the performance of already running kernels and to free memory for higher priority applications; however, the benefits of migration should be carefully evaluated, because the kernel that uses the to-be-migrated data must be stopped while the data is moved.

CONCLUSIONS AND FUTURE DEVELOPMENTS

In this section we have discussed the goals, requirements and solutions for an HPC software stack that targets deeply heterogeneous architectures composed of general purpose nodes and a variety of accelerators. In particular, we proposed to employ a combination of resource management techniques to control the allocation of computing units and memory resources to different applications under QoS requirements; and a low-level runtime support to provide a minimum common base among different accelerators, thus allowing functional portability of applications and an easier porting of high-level programming models on the different accelerators.

During the context of this project, we will heavily rely on the methodologies that we introduced in the previous chapters of this dissertation. This proves that our work succeeded in providing a set of tools, best practices and techniques to tackle the resource management problem for a quite wide range of architectures.

8 CONCLUSIONS

The end of Dennard's scaling has been one of the most disruptive events occurring in the evolution of the computing platforms. In order to cope with the subsequent increase in chips power density, hardware designers have progressively moved towards solutions that leverage the concepts of *parallelism* and *heterogeneity*. As a consequence, modern architectures feature an increasing number of shared computational resources that are power-hungry, can possibly be different in nature, and can be concurrently used by multiple applications.

This increasing hardware complexity has in turn affected the software stack: task scheduling and mapping have become challenging problems due to the need of maximizing the performance of applications while minimizing power, temperature, and contention on the shared resources.

This dissertation directly tackles the above mentioned problem. In particular, we address resource management from a horizontal perspective, trying to identify the challenges and solutions that pertain the increasingly blurred area between high-end embedded and High Performance Computing systems.

SUBJECTS COVERED BY THE DISSERTATION

Most and foremost, we focused on achieving power and energy efficiency by developing formalisms, methodologies and tools to deal with the "desired Quality of Service" of applications: by assigning to applications only the computational resources that they need to comply with their Quality of Service goals, it is possible to reduce overspecification. In order to address the above mentioned challenges, we strove to master the parallelism and heterogeneity opportunities that are offered by modern systems.

From the resource management perspective, we mostly focused on scheduling policies. That is, we tried to understand, depending from the target architecture and the optimization goals, how the resource manager can compute the most suitable resource allocation for each application. We also analyzed how the Linux operating system supports resource management across different architectures, and we implemented new mechanisms to support resource allocation.

We exploited resource management techniques to maximize performance and to minimize power (mostly HPC) and energy consumption (mostly battery-based embedded), but also to minimize thermal hot-spots and to mitigate the effects of memory contention and per-

formance variability. We did this by trying to understand which is the minimum amount of resources that applications need to comply with their Quality of Service goals and which is the best way to map resource demand on the hardware.

Regarding architectures, we evenly focused on embedded, desktop and HPC systems, but we also tried to find synergies between the two areas. For both these kinds of architectures, we addressed both homogeneous and heterogeneous scenarios.

During our work, we extensively used, modified and extended the Barbeque Run-Time Resource Manager, which is an Open-Source manager developed at Politecnico di Milano. You can find our contributions to this framework at `https://bitbucket.org/bosp/barbeque`.

CONTRIBUTIONS OF THE DISSERTATION

In this section, we summarize our novel contributions, and we draw the red line that links all the works presented in this dissertation.

Single-Computing-Node Systems

At operating system level, we studied how the Linux Control Groups perform CPU time allocation in linux-based systems. Indeed, this is a subject that is hard to find in literature: instead of focusing on how many and which resources should be allocated to applications, we analyzed what happens after resource allocation choices have been made. That is, we studied how CPU time allocation is actually enforced on the system. In particular, we analyzed how the cgroups *cpu* controller limits bandwidth. We discovered that, in certain scenarios, the CPU bandwidth that is effectively exploited by the applications could potentially be not only lower than the expected one, but also notably higher, which may cause troubles in multi-application scenarios.

Then, we dealt with application characterization, i.e., on how to extract application features that can be effectively exploited by resource managers to perform clever allocation choices. We employed Design Space Exploration techniques to characterize resource usage and energy consumption of applications. In this context, we extended YaMs, which is a multi-objective scheduling policy of the BarbequeRTRM, in order to provide it with memory-contention and energy consumption-aware mapping capabilities. Experimental results shown that, by exploiting the characterization information to suitably map applications, the resource manager is indeed able to optimize scheduling decisions. We also stressed the fact that the relationship between performance counters and energy consumption primarily depends on the target architecture and on the workload that we must execute. Hence, using performance counters as a proxy to energy is possible, but it also

requires a significant effort on the characterization side. Finally, we shown that, in order to maximize the Quality of Service that can be squeezed out from the available resources, applications need to constantly update their software parameters in order to continuously adapt them to the system status.

As a direct follow-up, we therefore decided to study the benefits that come from allowing a system-wide resource manager and an application-specific auto-tuner to work in a synergistic way. The main idea behind that study is that, whereas resource managers allocate resources to applications according to precise and known system-wide optimization goals, they are often unaware of what applications (and users) really need. That is, each application computes its Quality of Service in its own way. We tackled this problem by moving part of the management complexity to the applications side: each application relies on an application-specific auto-tuner, i.e., a component that is specifically configured for the target application and is able to tune the application software parameters at runtime in order to make it comply with its quality of service goal despite a runtime-variable resource availability.

In order to better exploit the capabilities of multi-core processors, we then added another degree of complexity: *heterogeneity*. We specifically addressed big.LITTLE architectures, i.e., processors that feature two clusters of cores—a performing and power-hungry one, which is called "big", and a slower and power-efficient one, which is called "little"—that share the same Instruction Set Architecture. In those systems, threads are allowed to freely migrate between the two clusters during runtime. By featuring different types of cores in the same chip, big.LITTLE processors allow operative systems to exploit the trade-off between performance and power consumption.

We studied how to dynamically migrate threads among the big and little clusters in order to maximize the usage of the big cluster while minimizing the performance losses that are induced by resource contention. We did so by introducing the concept of *stakes function*, which represents the trade-off between *exclusive allocation* and *sharing of resources* in multi-core processors. We introduced a co-scheduling policy that exploits *stakes functions* as a metric to take co-scheduling decisions on heterogeneous processors.

Then, we used the two clusters of cores as a heterogeneous OpenCL device. In this context, we presented a mechanism that forces the OpenCL runtime to view the big.LITTLE processor as a custom set of heterogeneous devices instead of viewing it as a single device.

Multiple-Computing-Nodes Systems

Finally, we dealt with distributed systems, mainly focusing on HPC. In those scenarios, the objectives of resource management are typ-

ically different from those of embedded systems. In particular, resource management techniques for HPC mostly focus on minimizing power consumption and thermal hot-spots, detecting and counteracting faults and aging-induced performance variability, and exploiting heterogeneous accelerators.

First of all, we performed and interesting study of how the freeze/restore-based process migration of MPI applications, which is usually performed at node granularity to address faults, can be made fine-grained in order to migrate only parts of the application on a different computing node. This allows resource managers to perform optimizations such as load balancing, resource consolidation, or also to counteract the effects induced on the hardware by aging. The outcome of this study was the development of the `mig` framework, an OpenMPI module that allows MPI applications—or even just a subset of their processes—to be migrated from a HPC node to another one without requiring developers to change their applications' code nor performing intrusive changes to the OpenMPI framework.

Once having dealt with system-wide mapping, we focused on the node-level optimization of distributed computation. In particular, we presented a resource management approach that exploits the trade-off between power consumption and performance when executing HPC applications that must comply with runtime-variable Quality of Service requirements. We applied an adaptive performance-aware execution model in the context of a real scientific application domain on a multi-core HPC system. The approach is based on the concept of "resource minimization via late termination". That is, we minimize the amount of resources that the applications can use so that they are barely able to comply with their Quality of Service requirements. The unused resources can be therefore used to perform system-wide optimizations such as minimizing power consumption or isolating faulty parts of the system. The experimental results shown that this approach is indeed capable of making applications comply with their runtime-variable QoS requirements while also minimizing resource usage and hence power consumption. Moreover, doing so did not induce energy inefficiency.

That of PerDeTemp was indeed a successiful story. The scheduling policy was used to cover the HPC use-case and, for the thermal standpoint, one of the embedded use-cases of the HARPA European project.

Our final step towards a performance, quality and power aware (but yet homogeneous) HPC resource management consisted in designing a feedback-based and partially decentralized resource management approach that allowed applications to comply with their Quality of Service Goals while minimizing resource usage; minimizing the negative effects of faults-induced performance variability; and leveling the temperature throughout the available computing cores, so that hot-spots are avoided and the effects of temperature on MTTF are equally balanced on the cores. This approach made use of most of the aforementioned contributions: it employed Control Groups-based CPU time allocation (configured according to our accuracy op-

timization approach), which, up to now, was used only in embedded systems; scheduling choices relied on our novel application characterization and auto-tuning approaches; resource allocation was partially handled by the centralized resource manager and in part by an application-specific decentralized manager, thus exploiting the aforementioned synergies between system-wide resource managers and application-specific auto-tuners; finally, the work enriched and extended our "resource minimization via late termination" approach. It did so by providing a richer information exchange between the BarbequeRTRM and the runtime library and by enabling a continuous resource allocation instead of using a discretized one.

To conclude this dissertation, we presented our "work in progress" in the context of the MANGO European Project, which aims at performing the first steps towards an unified runtime management support for deeply heterogeneous HPC systems. We discussed the goals, requirements and solutions for an HPC software stack that targets deeply heterogeneous architectures composed of both general purpose nodes and a variety of accelerators. In particular, we proposed to employ a combination of resource management techniques to control the allocation of computing units and memory resources to different applications under QoS requirements; and a low-level runtime support to provide a minimum common base among different accelerators, thus allowing functional portability of applications and an easier porting of high-level programming models on the different accelerators. We also shown how the methodologies and tools presented in this dissertation will be of great help during the MANGO project. In turn, this proves that our work succeeded in providing a set of tools, best practices and techniques to tackle the resource management problem for a quite wide range of architectures and application domains.

Part III

APPENDIX

A
THE BARBEQUE RUN-TIME RESOURCE MANAGER

This chapter introduces the Barbeque Run-Time Resource Manager (BarbequeRTRM), an open-source resource manager that may be employed on any Linux-based system. The modular structure of the BarbequeRTRM—new scheduling policies are developed as plug-ins and do not require the existing code to be changed—makes it a very interesting tool to carry out research on resource management.

If you are interested in the BOSP project, please visit our website: `https://bosp.dei.polimi.it`

USER-SPACE RESOURCE MANAGEMENT

As already discussed in Chapter 1, a resource manager is a software layer that orchestrates configuration and allocation of computational resources while taking into account system-wide and user-specific goals. Relying on a resource manager instead of distributing the management logic throughout the entire software stack is a very convenient approach for both operating systems and applications developers, since it allows them to move most of the management complexity in a black box that is easily portable and maintainable. That is why, during the last years, there was a strong push towards migrating the resource management logic into user-space processes [154].

Figure 61 shows a typical example of user-space-managed environment. Unmanaged applications run, as usual, on top of the operating system, and they are allowed to use a predefined set of general purpose resources. Managed applications run instead on top of the resource manager, which, according to some optimization policy, computes which is the best set of resources that must be allocated to each application. In order to monitor the system and to enforce the alloca-

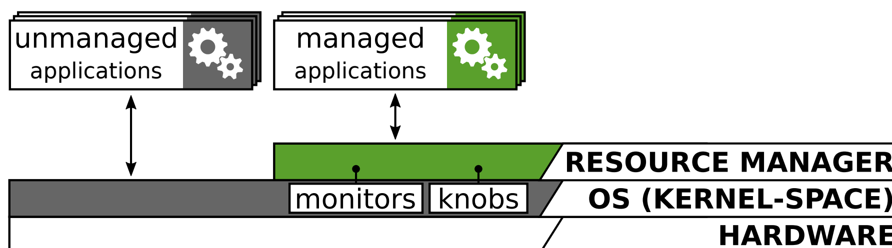


Figure 61: An user-space-managed environment. Unmanaged applications run on top of the operating system, while managed applications run on top of the resource manager, which exploits the monitors and knobs exposed by the operating system for management purposes.

tion choices, the resource manager relies respectively on the monitors and the knobs that are exposed by the operating system.

It is worth noticing that some resource managers move the enforcement complexity to the applications side. That is, the resource manager monitors the resources and computes allocations, while applications are in charge of enforcing the allocation, e.g. by using thread affinity to pin their threads on the allocated cores.

THE BARBEQUERTRM APPROACH

The BarbequeRTRM is an user-space deamon that dynamically allocates computing resources to managed applications (from now on, only "applications"). With "dynamic allocation" we refer to the ability of changing the resource allocation during runtime in order to address changes in system status, resource availability or applications QoS requirements. One of the most interesting features of the BarbequeRTRM is that, conversely to the typical job schedulers, it endows applications with an inherently resource-aware and quality-aware execution flow: the processing of applications is divided into bursts whose quality and resource demand are individually monitored. The allocation may change only between processing bursts; in that case, applications are notified about the new allocation so that they are able to reconfigure their own software parameters (e.g. number of threads) accordingly before the upcoming burst.

Figure 62 shows a typical example of BarbequeRTRM-managed environment. Applications do not directly communicate with the resource manager: they instead rely on the BarbequeRTRM Runtime Library (*RTLib*), which transparently negotiates resource allocation with the BarbequeRTRM. In order to monitor the system and to enforce resource allocation, the BarbequeRTRM in turn relies on the available monitors and knobs exposed by the operating system, e.g. sysfs, cpufreq and the Linux Control Groups.

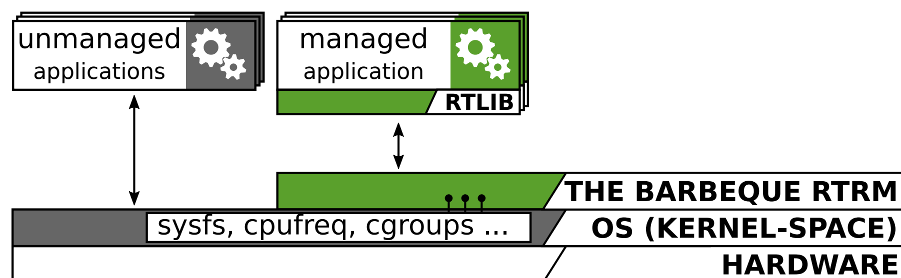


Figure 62: A BarbequeRTRM-managed environment. Unmanaged applications run on top of the operating system, while managed applications rely on the BarbequeRTRM Runtime Library, which transparently negotiates resource allocation with the BarbequeRTRM.

Managed Applications Execution Flow

To synchronize their processing with the dynamical resource allocation, the BarbequeRTRM requires applications to relinquish the control of their execution flow to the *RTLib*. That is, the *RTLib* drives the execution of running applications by choosing when to execute processing bursts and reconfiguration routines. To benefit from such support, the applications code must be moved into a C++ class that exposes the following methods:

SETUP Setting up the processing, e.g., spawning threads and performing mallocs;

CONFIGURE (re) Configuring the application software parameters according to the current allocation;

RUN Process the next chunk of data;

MONITOR Monitor the current Quality of Service. If needed, ask for a higher or lower one;

RELEASE Terminate the application, e.g., join threads and free mallocs.

During runtime, the aforementioned methods will be invoked by the *RTLib* according to the BarbequeRTRM resource and quality-aware execution flow shown in Figure 63. When the application starts, the *RTLib* asks for resources to the BarbequeRTRM (yellow arrow) and, while waiting for the allocation to be computed, it invokes the application *setup* method, which will take place in a pre-defined set of shared resources. While the application performs the setup, the BarbequeRTRM allocates it some resources and notifies the decision to the *RTLib* (green arrow). After the *setup*, the *RTLib* can therefore invoke the *configure* method, where the application, being aware of the current allocation, is able to configure itself accordingly. After that, the *RTLib* lets the application fall into the common execution flow (bold pattern), which consists bursts of *run* and *monitor* methods that are executed under a constant resource allocation. An allocation change can be triggered either by external events (e.g. some resources becoming unavailable) or in case of an unsatisfactory Quality of Service. In the latter case, the *RTLib* notifies the problem to the *BarbequeRTRM* (orange arrow), which will correct the allocation as soon as possible while letting the application continue its execution bursts.

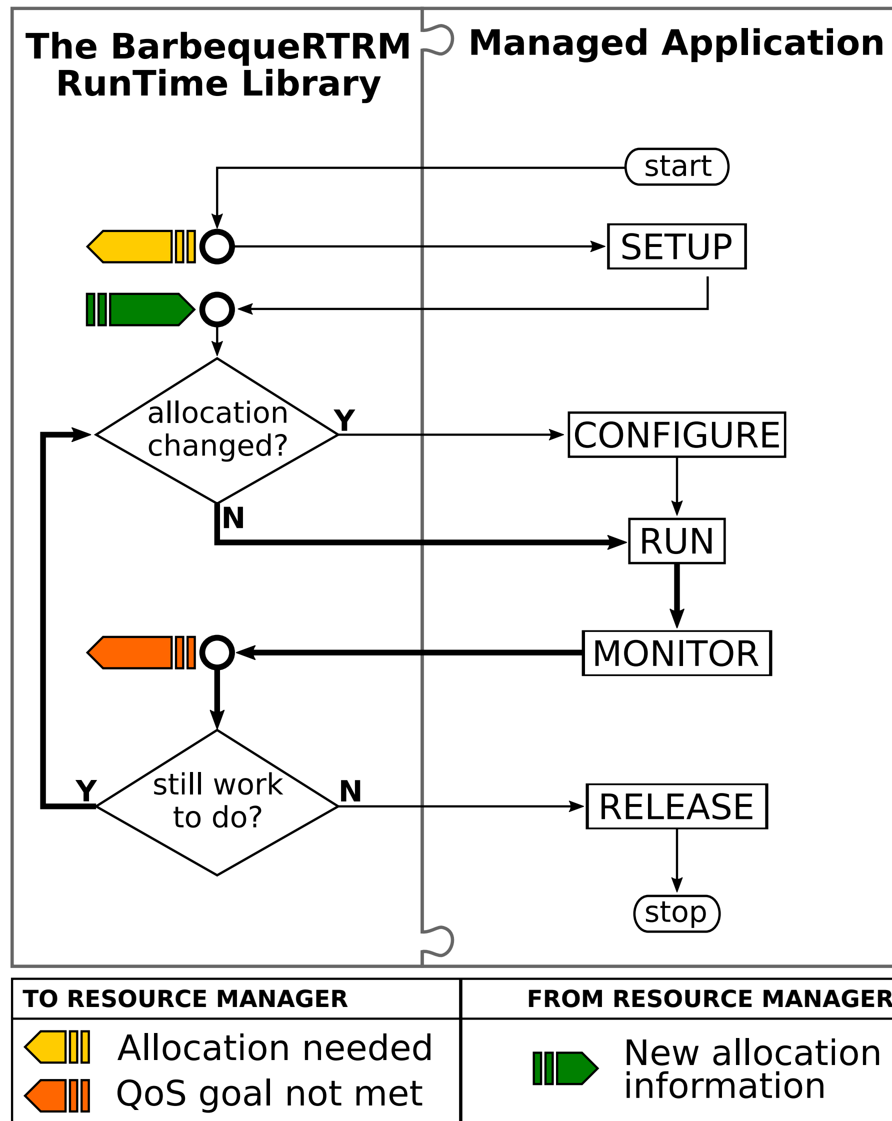


Figure 63: The BarbequeRTRM: execution flow of a managed application. The Runtime Library drives the execution and transparently synchronizes it with the runtime-variable resource allocation. Bold arrows indicate the most common execution flow, i.e, a continuous burst of run and monitor invocations under a constant resource allocation.

Integrating applications

In this subsection, we provide an example of application integration. Listing 5 shows the simplified code of a multimedia unmanaged application that processes a stream of video frames. Once started, the application performs an initialization and selects a parallelism level equal to the number of online cores (lines 3 – 8). Then, it processes the frames in bursts of one frame per active thread (lines 10 – 19). Finally, it checks the results correctness and terminates (lines 21 – 22).

Listing 5: Simplified example of a frame processing application in its non-integrated version. The application parses the arguments, initializes some data structure and selects a parallelism level. Then, it processes the frames in "threads_number"-sized bursts. Finally, it checks the results, joins the threads and terminates.

```
1  void main(int argc, char *argv[])
2  {
3      ParseCommandLine(argc, argv);
4      // Load data, initialize structures, ...
5      Initialize();
6      int result = 0;
7      // Using one thread per core (constant!)
8      int threads_number = sysconf(_SC_NPROCESSORS_ONLN);
9
10     while(HaveWorkToDo()) {
11         // Process 1 frame per thread
12         result = ProcessFrames(threads_number);
13
14         // Handle processing errors, if any
15         if (result) {
16             HandleError(result);
17             break;
18         }
19     }
20
21     CheckResult();
22     JoinThreads();
23 }
```

Listing 6: Main file of the integrated version of the application from Listing 5. The application initializes the RTLib and uses it to instantiate the class whose methods encapsulate the processing code. Then, it launches the processing and waits for it to terminate.

```
1  void main(int argc, char *argv[])
2  {
3      // Initialize the RTLib
4      rtlib rtlib_services;
5      ManagedTask t(rtlib_services, argc, argv);
6
7      // Start the application
8      t->StartExecution();
9      t->WaitTermination();
10 }
```

Listing 6 shows the main function of the corresponding integrated application. The original code is moved to a class, i.e. `ManagedTask`, that links with the RTLib. As a consequence, the main function is simplified: it just initializes the RTLib and the `ManagedTask` class (lines 4 – 5), then it starts the processing and waits for it to terminate (lines 8 – 9). Listing 8, instead, shows the implementation of the C++ class that encapsulates the application code by defining the Setup, Configure, Run, Monitor and Release methods. Apart from the logic that was added to perform runtime reconfiguration and to check if the

Listing 7: Simplified code of the integrated version of the application from Listing 5 (header).

```
class ManagedTask: public ManagedApplication {
public:

ManagedTask(rtlib rtlib_services, int argc, char *argv[]);

int Setup() override;
int Configure() override;
int Run() override;
int Monitor() override;
int Release() override;

int result = 0;
int threads_number = 0;
};
```

current QoS is satisfactory, the resulting code is the same of that of the original application.

DEFINING RESOURCE ALLOCATIONS

Given that the execution flow of managed applications is based on the concepts of Quality of Service compliance and software parameters reconfiguration, applications are supposed to be profiled at design time in order to compute a set of static allocations, each of whom enables a specific Quality of Service level and needs a known set of software parameters to be appropriately exploited. Accordingly, the BarbequeRTRM scheduling policies base their allocation choices on a static set of applications configurations, which are called Application Working Modes (AWMs).

Relying on a static set of configurations is a reasonable choice, since, in order to be managed by the BarbequeRTRM, applications must be re-factored. That is, the typical managed workload is entirely composed of applications that are known and that can therefore be profiled at design time. However, relying on static configurations also leads to complex issues:

- Profiles that are obtained at design time may be inaccurate at runtime, e.g., due to different CPU cores frequencies, temperature variation, aging-induced performance variability, data variability or presence of concurrently running applications;
- An exhaustive application profiling (i.e. profiling the application for each software and parameter combination and for each possible resource allocation) is complex and time wasting. Moreover, it must be carried out once for each different architecture;
- Discrete allocations may lead to instability. For instance, if the current resource demand of an application is three cores but the

Listing 8: Simplified code of the integrated version of the application from Listing 5 (implementation). Underlined functions are calls to the RTLib APIs.

```
1
2 ManagedTask::ManagedTask(rtlib rtlib_services, int argc, char *argv[]) {
3     init_interaction(rtlib);
4     ParseCommandLine(argc, argv);
5 }
6
7 int ManagedTask::Setup() {
8     Initialize();
9     return RTLIB_OK;
10 }
11
12 int ManagedTask::Configure() {
13     // Using one thread per allocated core (runtime-variable!)
14     threads_number = get_alloc_cores();
15     return RTLIB_OK;
16 }
17
18 int ManagedTask::Run() {
19     if (! HaveWorkToDo()) return RTLIB_WORKLOAD_NONE;
20
21     // Process one frame per thread and update frame count
22     result = ProcessFrames(threads_number);
23
24     if (result) {
25         HandleError(result);
26         return RTLIB_ERROR;
27     }
28
29     return RTLIB_OK;
30 }
31
32 int ManagedTask::Monitor() {
33     if (DontLike(get_throughput()))
34         complain();
35     return RTLIB_OK;
36 }
37
38 int ManagedTask::Release() {
39     CheckResult();
40     JoinThreads();
41     return RTLIB_OK;
42 }
```

profiled configurations feature two or four cores, the allocation will continuously switch between the two available allocations.

It is worth noticing that the current version of the BarbequeRTRM also supports scheduling policies that compute allocations in the continuous space. That is, resource allocations are computed on-the-fly with disregard of the profiled configurations. We deal with continuous allocation in the dissertation's proper, while describing our own contributions.

Listing 9 shows an example of application recipe, which is an XML file that contains the static list of AWMs. When computing how many

Listing 9: Example of application recipe. A recipe is an XML file that contains a static list of Application Working Modes (AWMs). When computing the resource allocation for an application, the scheduling policy chooses an AWM between those that are listed in its recipe.

```
<?xml version="1.0"?>
<BarbequeRTRM recipe_version="0.8">
<!-- Priority wrt other managed applications -->
<application priority="1">
<!-- Allocations for any Linux/CGroups-based multi-core -->
    <platform id="org.linux.cgroup">
        <awms>
            <!-- AWM 0: low quality (got lowest value) -->
            <awm id="0" name="only_one_cpu" value="10">
                <resources>
                <!-- Allocation: a whole core -->
                    <cpu id="0">
                        <pe qty="100"/>
                    </cpu>
                </resources>
            </awm>
            <!-- AWM 1: high quality (x10 wrt AWM 0) -->
            <awm id="1" name="cpu_and_gpu" value="100">
                <resources>
                <!-- Allocation: 20% of a core -->
                    <cpu id="0">
                        <pe qty="20"/>
                    </cpu>
                    <!-- Allocation: a GPU -->
                        <gpu id="0">
                        <pe qty="100"/>
                    </gpu>
                </resources>
            </awm>
        </awms>
    </platform>
    </application>
    </BarbequeRTRM>
```

and which resources will be allocated to an application, the scheduling policy chooses an AWM between those that are listed in the corresponding recipe and tries to wisely mapping it on the available resources. The recipe can contain multiple platform sections, each of whom lists the AWMs for a particular architecture. Each AWM is identified by an ID, a human-readable description, a Quality of Service level (expressed as an integer value) and the list of resources that must be allocated to the application in order to make it reach (ideally) that QoS level.

BIBLIOGRAPHY

[1] Gordon E Moore et al. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[2] Gordon E Moore. Progress in digital integrated electronics [technical literaiture, copyright 1975 ieee. reprinted, with permission. technical digest. international electron devices meeting, ieee, 1975, pp. 11-13.]. *IEEE Solid-State Circuits Society Newsletter*, 20(3), 2006.

[3] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[4] Tom Simonite. Intel puts the brakes on moore s law, 2016.

[5] Qing Cao, Shu-Jen Han, Jerry Tersoff, Aaron D Franklin, Yu Zhu, Zhen Zhang, George S Tulevski, Jianshi Tang, and Wilfried Haensch. End-bonded contacts for carbon nanotube transistors with low, size-independent resistance. *Science*, 350(6256):68–72, 2015.

[6] Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[7] Kirk M Bresniker, Sharad Singhal, and R Stanley Williams. Adapting to thrive in a new economy of memory abundance. *Computer*, 48(12):44–53, 2015.

[8] D. Hillman. Integrated power management, leakage control and process compensation technology for advanced processes. https://www.design-reuse.com/articles/20296/power-management-leakage-control-process-compensation.html. Accessed: April 11, 2017.

[9] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[10] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30 (atlantic city, nj, apr. 18–20), afips press, reston, va., 1967, pp. 483–485, when dr. amdahl was at international business machines corporation, sunnyvale, california. *IEEE Solid-State Circuits Society Newsletter*, 12(3):19–20, 2007.

[11] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7), 2008.

[12] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 365–376. ACM, 2011.

[13] James Donald and Margaret Martonosi. Techniques for multicore thermal management: Classification and new exploration. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 78–88. IEEE Computer Society, 2006.

[14] Shekhar Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Ieee Micro*, 25(6):10–16, 2005.

[15] Ehsan Pakbaznia and Massoud Pedram. Minimizing data center cooling and server power costs. In *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*, pages 145–150. ACM, 2009.

[16] M Duranton, KD Bosschere, A Cohen, J Maebe, and H Munk. Hipeac vision 2015. high performance and embedded architecture and compilation. *http://www. hipeac. org/assets/public/publications/vision/hipeac-vision-2015_DqoboL8. pdf*, 2015.

[17] M Duranton, KD Bosschere, C Gamrat, J Maebe, H Munk, and O Zendra. Hipeac vision 2017. high performance and embedded architecture and compilation. *http://www. hipeac. org/assets/public/publications/vision/hipeac-vision-2015_DqoboL8. pdf*, 2017.

[18] Bill Cheswick. An evening with berferd in which a cracker is lured, endured, and studied. In *Proc. Winter USENIX Conference, San Francisco*, pages 20–24, 1992.

[19] Cyrus Peikari and Anton Chuvakin. *Security Warrior: Know Your Enemy*. " O'Reilly Media, Inc.", 2004.

[20] M. Riondato. Freebsd handbook chapter 15 jails. `https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/jails.html`. Accessed: May 2, 2017.

[21] Benoit des Ligneris. Virtualization of linux based computers: the linux-vserver project. In *High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on*, pages 340–346. IEEE, 2005.

[22] Paul B Menage. Adding generic process containers to the linux kernel. In *Proceedings of the Linux Symposium*, volume 2, pages 45–57. Citeseer, 2007.

[23] J. Corbet. Notes from a container. https://lwn.net/Articles/256389/. Accessed: May 2, 2017.

[24] http://man7.org/linux/man-pages/man7/cgroups.7.html. Accessed: 2017-02-06.

[25] Jens Axboe. Linux block io—present and future. In *Ottawa Linux Symp*, pages 51–61, 2004.

[26] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

[27] Fulya Kaplan, Jie Meng, and Ayse K Coskun. Optimizing communication and cooling costs in hpc data centers via intelligent job allocation. In *Green Computing Conference (IGCC), 2013 International*, pages 1–10. IEEE, 2013.

[28] Z. Zhou, Z. Lan, W. Tang, and N. Desai. Reducing energy costs for ibm blue gene/p via power-aware job scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 96–115. Springer, 2013.

[29] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31:6–15, 2011.

[30] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM Sigplan Notices*, volume 45, pages 129–142. ACM, 2010.

[31] Y. Georgiou, T. Cadeau, D. Glesser, D. Auble, M. Jette, and M. Hautreux. Energy accounting and control with slurm resource and job management system. In *International Conference on Distributed Computing and Networking*, pages 96–118. Springer, 2014.

[32] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[33] C. Bolchini, S. Cherubin, G. C Durelli, S. Libutti, A. Miele, and M.D. Santambrogio. A runtime controller for opencl applications on heterogeneous system architectures. 2016.

[34] P. Bellasi, G. Massari, and W. Fornaciari. Effective runtime resource management using linux control groups with the barbequertrm framework. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(2):39, 2015.

[35] S. Libutti, G. Massari, and W. Fornaciari. Co-scheduling tasks on multi-core heterogeneous systems: An energy-aware perspective. *IET Computers & Digital Techniques*, 10(2):77–84, 2016.

[36] A. Portero, R. Vavrık, S. Kuchár, M. Golasowski, V. Vondrák, S. Libutti, G. Massari, and W. Fornaciari. Flood prediction model simulation with heterogeneous trade-offs in high performance computing framework. In *29th EUROPEAN Conference on Modelling and Simulation ECMS*, 2015.

[37] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[38] Dakshina Dasari and Vincent Nelis. An analysis of the impact of bus contention on the wcet in multicores. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 1450–1457. IEEE, 2012.

[39] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Bandwidth bandit: Quantitative characterization of memory contention. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2013.

[40] Heechul Yun. Parallelism-aware memory interference delay analysis for cots multicore systems. *arXiv preprint arXiv:1407.7448*, 2014.

[41] Josh Aas. Understanding the linux 2.6. 8.1 cpu scheduler. *Retrieved Oct*, 16:1–38, 2005.

[42] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 220–229. ACM, 2008.

[43] Kai Tian, Yunlian Jiang, and Xipeng Shen. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In *Proceedings of the 6th ACM conference on Computing frontiers*, pages 41–50. ACM, 2009.

[44] Yunlian Jiang, Kai Tian, Xipeng Shen, Jinghe Zhang, Jie Chen, and Rahul Tripathi. The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions. *Parallel and Distributed Systems, IEEE Transactions on*, 22(7):1192–1205, 2011.

[45] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems (TOCS)*, 28(4):8, 2010.

[46] Tanima Dey, Wei Wang, Jack W Davidson, and Mary Lou Soffa. Characterizing multi-threaded applications based on shared-resource contention. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 76–86. IEEE, 2011.

[47] M. Tillenius, E. Larsson, R.M. Badia, and X. Martorell. Resource-aware task scheduling. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(1):5, 2015.

[48] Changjiu Xian, Yung-Hsiang Lu, and Zhiyuan Li. A programming environment with runtime energy characterization for energy-aware applications. In *Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on*, pages 141–146. IEEE, 2007.

[49] Seong Jo Kim, Seung Woo Son, Wei-keng Liao, Mahmut Kandemir, Rajeev Thakur, and Alok Choudhary. Iopin: Runtime profiling of parallel i/o in hpc systems. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 18–23. IEEE, 2012.

[50] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N Bhuyan. Adapt: A framework for coscheduling multithreaded programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):45, 2013.

[51] Alexandros-Herodotos Haritatos, Georgios Goumas, Nikos Anastopoulos, Konstantinos Nikas, Kornilios Kourtis, and Nectarios Koziris. Lca: a memory link and cache-aware co-scheduling approach for cmps. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 469–470. ACM, 2014.

[52] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*, pages 153–166. ACM, 2010.

[53] Radim Vavřík, Antoni Portero, Štěpán Kuchař, Martin Golasowski, Simone Libutti, Giuseppe Massari, William Fornacciari, and Vít Vondrák. Precision-aware application execution for energy-optimization in hpc node system. *arXiv preprint arXiv:1501.04557*, 2015.

[54] Josué Feliu, Julio Sahuquillo, Salvador Petit, and José Duato. L1-bandwidth aware thread allocation in multicore smt processors. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pages 123–132. IEEE, 2013.

[55] Josué Feliu, Julio Sahuquillo, Salvador Petit, and José Duato. Addressing bandwidth contention in smt multicores through scheduling. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 167–167. ACM, 2014.

[56] Sébastien Hily and André Seznec. *Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading*. PhD thesis, INRIA, 1997.

[57] Wei Wang, Tanima Dey, Jason Mars, Lingjia Tang, Jack W Davidson, and Mary Lou Soffa. Performance analysis of thread mappings with a holistic view of the hardware resources. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 156–167. IEEE, 2012.

[58] Hossam El Din, Wael Amr, Hany Mohamed ElSayed, and Ihab ElSayed Talkhan. Reducing shared cache misses via dynamic grouping and scheduling on multicores. *International Journal of Advanced Computer Science & Applications*, 5(9), 2014.

[59] Deukhyeon An, Jeehong Kim, JungHyun Han, and Young Ik Eom. Reducing last level cache pollution in numa multicore systems for improving cache performance. In *Computational Science and Its Applications–ICCSA 2012*, pages 272–282. Springer, 2012.

[60] Xin Xu and Manman Peng. Management for shared cmp caches. *Information Technology Journal*, 12(7):1366–1372, 2013.

[61] Josue Feliu, Salvador Petit, Julio Sahuquillo, and Jose Duato. Cache-hierarchy contention-aware scheduling in cmps. *Parallel and Distributed Systems, IEEE Transactions on*, 25(3):581–590, 2014.

[62] Zoltan Majo and Thomas R Gross. Memory management in numa multicore systems: trapped between cache contention and interconnect overhead. In *ACM SIGPLAN Notices*, volume 46, pages 11–20. ACM, 2011.

[63] Di Xu, Chenggang Wu, and Pen-Chung Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 237–248. ACM, 2010.

[64] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern

warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259. ACM, 2011.

[65] Sandro Penolazzi, Ingo Sander, and Ahmed Hemani. Predicting bus contention effects on energy and performance in multiprocessor socs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–4. IEEE, 2011.

[66] Andreas De Blanche and Thomas Lundqvist. A methodology for estimating co-scheduling slowdowns due to memory bus contention on multicore nodes. In *International conference on parallel and distributed computing and networks*, 2014.

[67] Simone Libutti, Giuseppe Massari, Patrick Bellasi, and William Fornaciari. Exploiting performance counters for energy efficient co-scheduling of mixed workloads on multi-core platforms. In *Proceedings of Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, page 27. ACM, 2014.

[68] Robert L McGregor, Christos D Antonopoulos, and Dimitrios S Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 28a–28a. IEEE, 2005.

[69] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. Contention aware execution: online contention detection and response. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 257–265. ACM, 2010.

[70] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 207–216. IEEE, 2010.

[71] C. Bienia and K. Li. *Benchmarking modern multiprocessors*. Princeton University USA, 2011.

[72] Andreas Merkel and Frank Bellosa. Memory-aware scheduling for energy efficiency on multicore processors. *HotPower*, 8:123–130, 2008.

[73] GIUSEPPE MASSARI. *Run-time resource management of multi/many-core computing systems*. PhD thesis, Politecnico di Milano, 2015.

[74] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Introduction to NP-Completeness of knapsack problems*. Springer, 2004.

[75] Davide Gadioli, Simone Libutti, Giuseppe Massari, Edoardo Paone, Michele Scandale, Patrick Bellasi, Gianluca Palermo, Vittorio Zaccaria, Giovanni Agosta, William Fornaciari, et al. Opencl application auto-tuning and run-time resource management for multi-core platforms. In *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on*, pages 127–133. IEEE, 2014.

[76] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ACM SIGPLAN Notices*, volume 46, pages 199–212. ACM, 2011.

[77] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 13–24. ACM, 2013.

[78] Intel. Intel xeon processor v4 family. https://ark.intel.com/products/family/93797. Accessed: October 2, 2017.

[79] Nvidia. Nvidia pascal. https://www.nvidia.com/en-us/geforce/products/10series/architecture/#forceLocation=US. Accessed: October 2, 2017.

[80] Nvidia. Nvidia tegra x1. http://www.nvidia.com/object/tegra-x1-processor.html. Accessed: October 2, 2017.

[81] Adapteva. The parallela computer. http://www.adapteva.com/parallella/. Accessed: October 2, 2017.

[82] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[83] E. Paone, D. Gadioli, G. Palermo, V. Zaccaria, and C. Silvano. Evaluating orthogonality between application auto-tuning and run-time resource management for adaptive opencl applications. In *2014 IEEE 25th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 161–168. IEEE, 2014.

[84] Ke Zhang, Jiangbo Lu, and Gauthier Lafruit. Cross-based local stereo matching using orthogonal integral images. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(7):1073–1079, 2009.

[85] Holger Endt and Kay Weckemann. Remote utilization of opencl for flexible computation offloading using embedded ecus, ce devices and cloud servers. In *PARCO*, pages 133–140, 2011.

[86] Rance Rodrigues, Arunachalam Annamalai, Israel Koren, and Sandip Kundu. Improving performance per watt of asymmetric multi-core processors via online program phase classification and adaptive core morphing. *ACM Transactions on Design Automation of Electronic Systems*, 18(1):5, 2013.

[87] Rance Rodrigues, Israel Koren, and Sandip Kundu. Performance and power benefits of sharing execution units between a high performance core and a low power core. In *VLSI Design and 2014 13th International Conference on Embedded Systems, 27th International Conference on*, pages 204–209. IEEE, 2014.

[88] Peter Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 2011.

[89] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, Nickolai Zeldovich, et al. An analysis of linux scalability to many cores. In *OSDI*, volume 10, pages 86–93, 2010.

[90] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N Bhuyan. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 116–125. IEEE, 2011.

[91] Rajiv Nishtala, Daniel Mossé, and Vinicius Petrucci. Energy-aware thread co-location in heterogeneous multicore processors. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–9. IEEE, 2013.

[92] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N Bhuyan. Shuffling: a framework for lock contention aware thread scheduling for multicore multiprocessor systems. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 289–300. ACM, 2014.

[93] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 220–229. ACM, 2008.

[94] R. Knauerhase, P. Brett, B. Hohlt, Tong Li, and S. Hahn. Using os observations to improve performance in multicore systems. *Micro, IEEE*, 28(3):54–66, May 2008.

[95] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems*, pages 153–166. ACM, 2010.

[96] Josué Feliu, Julio Sahuquillo, Salvador Petit, and José Duato. L1-bandwidth aware thread allocation in multicore smt processors. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 123–132, Piscataway, NJ, USA, 2013. IEEE Press.

[97] Yingxin Wang, Yan Cui, Pin Tao, et al. Reducing shared cache contention by scheduling order adjustment on commodity multi-cores. In *IPDPS Workshops*, pages 984–992. IEEE, 2011.

[98] Shin gyu Kim, Hyeonsang Eom, and Heon Y. Yeom. Virtual machine scheduling for multicores considering effects of shared on-chip last level cache interference. In *Proceedings of the 2012 International Green Computing Conference (IGCC)*, pages 1–6, Washington, DC, USA. IEEE.

[99] Jun Wei Lam, I. Tan, Boon Leong Ong, and Chang Kian Tan. Effective operating system scheduling domain hierarchy for core-cache awareness. In *TENCON - 2009 IEEE Region 10 Conference*, pages 1–7.

[100] Yingxin Wang, Yan Cui, Pin Tao, et al. Reducing shared cache contention by scheduling order adjustment on commodity multi-cores. In *Parallel and Distributed Processing Workshops and Phd Forum, IEEE International Symposium on*, pages 984–992, 2011.

[101] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers*, pages 29–40. ACM, 2006.

[102] Luca Lugini, Vinicius Petrucci, and Daniel Mosse. Online thread assignment for heterogeneous multicore systems. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 538–544. IEEE, 2012.

[103] Kenzo Van Craeynest, Safia Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. Fairness-aware scheduling on single-isa heterogeneous multi-cores. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pages 177–187. IEEE, 2013.

[104] Quan Chen and Minyi Guo. Adaptive workload-aware task scheduling for single-isa asymmetric multicore architectures.

ACM Transactions on Architecture and Code Optimization, 11(1):8, 2014.

[105] Alexandra Fedorova, Margo Seltzer, and Michael D Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38. IEEE, 2007.

[106] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *SIGPLAN*, 45(3):129–142, 2010.

[107] Huanzhou Zhu and Ligang He. A Graph based approach for Co-scheduling jobs on Multi-core computers. In *Imperial College Computing Student Workshop*, volume 35, pages 144–151, 2013.

[108] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[109] P. Menage, R. Seth, P. Jackson, and C. Lameter. Linux control groups, 2007.

[110] G Stoner. Hsa foundation overview. *HSA Foundation*, 2012.

[111] SAMSUNG. Exynos 5 octa (5420). http://www.samsung.com/semiconductor/minisite/Exynos/w/solution/mobile_ap/5420. Accessed: May 30, 2017.

[112] Xilinx. Zynq-7000 all programmable soc. https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. Accessed: May 30, 2017.

[113] Intel. Opencl™device fission for cpu performance. https://software.intel.com/en-us/articles/opencl-device-fission-for-cpu-performance. Accessed: May 31, 2017.

[114] Amazon. Amazon ec2. https://aws.amazon.com/ec2/. Accessed: June 5, 2017.

[115] Shuangcheng Niu, Jidong Zhai, Xiaosong Ma, Xiongchao Tang, Wenguang Chen, and Weimin Zheng. Building semi-elastic virtual clusters for cost-effective hpc cloud resource provisioning. *IEEE Transactions on Parallel and Distributed Systems*, 27(7):1915–1928, 2016.

[116] Bartosz Balis, Kamil Figiela, Konrad Jopek, Maciej Malawski, and Maciej Pawlik. Porting hpc applications to the cloud: A multi-frontal solver case study. *Journal of Computational Science*, 2016.

[117] Wu-chun Feng and Kirk Cameron. The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12), 2007.

[118] Federico Reghenzani, Gianmario Pozzi, Giuseppe Massari, Simone Libutti, and William Fornaciari. The mig framework: Enabling transparent process migration in open mpi. In *Proceedings of the 23rd European MPI Users' Group Meeting*, pages 64–73. ACM, 2016.

[119] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L Scott. Proactive process-level live migration in hpc environments. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 43. IEEE Press, 2008.

[120] Ifeanyi P Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[121] Ian Philp. Software failures and the road to a petaflop machine. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, 2005.

[122] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[123] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[124] Joshua Hursey, Timothy I Mattox, and Andrew Lumsdaine. Interconnect agnostic checkpoint/restart in open mpi. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 49–58. ACM, 2009.

[125] Bryan Mills, Ryan E Grant, Kurt B Ferreira, and Rolf Riesen. Evaluating energy savings for checkpoint/restart. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*, page 6. ACM, 2013.

[126] Miguel G Xavier, Marcelo Veiga Neves, Fabio D Rossi, Tiago C Ferreto, Tobias Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240. IEEE, 2013.

[127] Simon Pickartz, Jens Breitbart, and Stefan Lankes. Impacts of virtualization on intra-host communication. 2016.

[128] Criu - checkpoint/restore in userspace. https://criu.org/. Accessed: 2016-04-11.

[129] W. Li, A. Kanso, and A. Gherbi. Leveraging linux containers to achieve high availability for cloud services. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 76–83, March 2015.

[130] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[131] Rami Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 2013.

[132] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[133] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. Open mpi: A flexible high performance mpi. In *Parallel Processing and Applied Mathematics*, pages 228–239. Springer, 2005.

[134] L Peter Deutsch. Gzip file format specification version 4.3. 1996.

[135] G. Massari, S. Libutti, A. Portero, R. Vavrik, S. Kuchar, V. Vondrak, L. Borghese, and W. Fornaciari. Harnessing performance variability: A hpc-oriented application scenario. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 111–116. IEEE, 2015.

[136] Rajkumar Buyya, James Broberg, and Andrzej M Goscinski. *Cloud computing: Principles and paradigms*, volume 87. John Wiley & Sons, 2010.

[137] SAEID Abolfazli, Z Sanaei, MH Sanaei, M Shojafar, and A Gani. Mobile cloud computing: The-state-of-the-art, challenges, and future research. 2015.

[138] Fábio Itturiet, Gabriel Nazar, Ronaldo Ferreira, Álvaro Moreira, and Luigi Carro. Adaptive parallelism exploitation under physical and real-time constraints for resilient systems. *ACM Trans. Reconfigurable Technol. Syst.*, 7(3):25:1–25:17, September 2014.

[139] Myeongjae Jeon, Yuxiong He, Sameh Elnikety, Alan L Cox, and Scott Rixner. Adaptive parallelism for web search. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 155–168. ACM, 2013.

[140] Major Bhadauria and Sally A McKee. An approach to resource-aware co-scheduling for cmps. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 189–199. ACM, 2010.

[141] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Robust adaptation to available parallelism in transactional memory applications. In *Transactions on high-performance embedded architectures and compilers III*, pages 236–255. Springer, 2011.

[142] Diego Didona, Pascal Felber, Derin Harmanci, Paolo Romano, and Joerg Schenker. Identifying the optimal level of parallelism in transactional memory applications. In *Networked Systems*, pages 233–247. Springer, 2013.

[143] R Clint Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998.

[144] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[145] Markus Püschel, José MF Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *International Journal of High Performance Computing Applications*, 18(1):21–45, 2004.

[146] Shoaib A Kamil. Productive high performance parallel programming with auto-tuned domain-specific embedded languages. Technical report, DTIC Document, 2013.

[147] M. Golasowski, M. Litschmannova, S. Kuchar, M. Podhoranyi, and J. Martinovic. Uncertainty modelling in rainfall-runoff simulations based on parallel monte carlo method. In *International Journal on non-standard computing and artificial intelligence NNW . Accepted for publication, 2015. Proceedings*, 2015.

[148] Antoni Portero, Stepán Kuchár, Radim Vavrík, Martin Golasowski, Simone Libutti, Giuseppe Massari, William Fornaciari,

and Vít Vondrák. Flood prediction model simulation with heterogeneous trade-offs in high performance computing framework. In *29th EUROPEAN Conference on Modelling and Simulation ECMS 2015, Albena (Varna), Bulgaria. May 26th - 29th, 2015. Proceedings*, 2015.

[149] Antoni Portero, Stepán Kuchár, Radim Vavrík, Martin Golasowski, and Vít Vondrák. System and application scenarios for disaster management processes, the rainfall-runoff model case study. In *Computer Information Systems and Industrial Management - 13th IFIP TC8 International Conference, CISIM 2014, Ho Chi Minh City, Vietnam, November 5-7, 2014. Proceedings*, pages 315–326, 2014.

[150] Nikolaos Zompakis, Michail Noltsis, Lorena Ndreu, Zacharias Hadjilambrou, Panagiotis Englezakis, Panagiota Nikolaou, Antoni Portero, Simone Libutti, Giuseppe Massari, Federico Sassi, et al. Harpa: Tackling physically induced performance variability. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 97–102. IEEE, 2017.

[151] A. Portero, M. Podhoranyi, S. Libutti, G. Massari, and W. Fornaciari. Just-in-time execution to adapt on demand resource allocation in hpc systems. In *Algorithms, Computing and Systems, 2017. ICACS 2005. International Conference on*. ACM, 2017.

[152] M. Wolf. *High-Performance Embedded Computing: Applications in Cyber-Physical Systems and Mobile Computing*. Newnes, 2014.

[153] H. et al. Mair. A highly integrated smartphone soc featuring a 2.5 ghz octa-core cpu with advanced high-performance and low-power techniques. *IEEE ISSCC Dig. Tech. Papers*, 2015.

[154] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys (CSUR)*, 45(1):4, 2012.

[155] P. Gupta, S. G Koolagudi, R. Khanna, M. Ganguli, and A.N. Sankaranarayanan. Analytic technique for optimal workload scheduling in data-center using phase detection. In *Energy Aware Computing Systems & Applications (ICEAC), 2015 International Conference on*, pages 1–4. IEEE, 2015.

[156] H. E Egilmez, S. Civanlar, and A.M. Tekalp. An optimization framework for qos-enabled adaptive video streaming over openflow networks. *Multimedia, IEEE Transactions on*, 15(3):710–715, 2013.

[157] Roman Sliva and Filip Stanek. Best practice guide anselm. http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Anselm.pdf. Accessed: July 7, 2017.

[158] Federico D Sacerdoti, Mason J Katz, Matthew L Massie, and David E Culler. Wide area cluster monitoring with ganglia. In *null*, page 289. IEEE, 2003.

[159] Antoni Portero, Jiri Sevcik, Martin Golasowski, Radim Vavrík, Simone Libutti, Giuseppe Massari, Francky Catthoor, William Fornaciari, and Vít Vondrák. Using an adaptive and time predictable runtime system for power-aware hpc-oriented applications. In *Green and Sustainable Computing Conference (IGSCo< 2016 Seventh International*, pages 1–6. IEEE, 2016.

[160] Paul Ellerman. Calculating reliability using fit & mttf: Arrhenius htol model. *microsemi, Tech. Rep.*, 2012.

[161] Jose Flich, Giovanni Agosta, Philipp Ampletzer, David Atienza Alonso, Alessandro Cilardo, William Fornaciari, Mario Kovac, Fabrice Roudet, and Davide Zoni. The mango fet-hpc project: An overview. In *Computational Science and Engineering (CSE), 2015 IEEE 18th International Conference on*, pages 351–354. IEEE, 2015.

[162] top500.org. Top green 500, november 2016. https://www.top500.org/green500/lists/2016/11/. Accessed: July 18, 2017.

[163] top500.org. Top 500, november 2016. https://www.top500.org/lists/2016/11/. Accessed: July 18, 2017.

[164] Bastian Koller, Nico Struckmann, Jochen Buchholz, and Michael Gienger. *Towards an Environment to Deliver High Performance Computing to Small and Medium Enterprises*, pages 41–50. Springer International Publishing, Cham, 2015.

[165] Jose Flich, Giovanni Agosta, Philipp Ampletzer, David Atienza, Carlo Brandolese, Alessandro Cilardo, William Fornaciari, Ynse Hoornenborg, Mario Kovač, Igor Piljić, et al. Mango: exploring manycore architectures for next generation hpc system. In *Design, Automation, and Test in Europe (DATE2016)*, 2016.

[166] Khronos Group. The Open Standard for Parallel Programming of Heterogeneous Systems. https://www.khronos.org/opencl/, (retr. Jul 2015).

[167] Giovanni Agosta, Alessandro Barenghi, Alessandro Di Federico, and Gerardo Pelosi. OpenCL Performance Portability for General-purpose Computation on Graphics Processor Units: an

Exploration on Cryptographic Primitives. *Concurrency and Computation: Practice and Experience*, 2014.

[168] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. Towards Transparently Tackling Functionality and Performance Issues across Different OpenCL Platforms. In *2nd Int'l Symp. on Computing and Networking (CANDAR)*, pages 130–136, Dec 2014.

[169] 2PARMA Project. Parallel paradigms and run-time management techniques for many-core architectures. http://www.2parma.eu.

[170] C. Silvano, W. Fornaciari, S.C. Reghizzi, G. Agosta, G. Palermo, V. Zaccaria, P. Bellasi, F. Castro, S. Corbetta, E. Speziale, D. Melpignano, J.M. Zins, D. Siorpaes, H. Hubert, B. Stabernack, J. Brandenburg, M. Palkovic, P. Raghavan, C. Ykman-Couvreur, A. Bartzas, D. Soudris, T. Kempf, G. Ascheid, H. Meyr, J. Ansari, P. Mahonen, and B. Vanthournout. Parallel paradigms and run-time management techniques for many-core architectures: The 2parma approach. In *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pages 835–840, July 2011.

[171] Giuseppe Massari, Chiara Caffarri, Patrick Bellasi, and William Fornaciari. Extending a run-time resource management framework to support opencl and heterogeneous systems. In *Proceedings of Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM '14, pages 21:21–21:26, New York, NY, USA, 2014. ACM.

[172] D. Zoni, S. Corbetta, and W. Fornaciari. Thermal/performance trade-off in network-on-chip architectures. In *System on Chip (SoC), 2012 International Symposium on*, pages 1–8, Oct 2012.

[173] Simone Libutti, Giuseppe Massari, and William Fornaciari. Co-scheduling tasks on multi-core heterogeneous systems: An energy-aware perspective. *IET Computers & Digital Techniques*, 2015.

[174] Message Passing Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[175] David Clark. Openmp: A parallel standard for the masses. *IEEE Concurrency*, 6:10–12, 1998.

[176] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceed-*

ings of the 11th International Conference on Compiler Construction, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.

[177] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[178] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[179] Philipp Haller and Martin Odersky. Actors that unify threads and events. In *COORDINATION'07: Proceedings of the 9th international conference on Coordination models and languages*, pages 171–190, Berlin, Heidelberg, 2007. Springer-Verlag.

[180] Bernard Sufrin. Communicating scala objects. In Peter H. Welch, Susan Stepney, Fiona Polack, Fred R. M. Barnes, Alistair A. McEwan, Gardiner S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *CPA*, volume 66 of *Concurrent Systems Engineering Series*, pages 35–54. IOS Press, 2008.

[181] ARB. *OpenMP Application Program Interface, version 3.0*, 2008.

[182] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[183] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*, pages 212–223, 1998.

[184] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.

[185] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[186] CORPORATE Rice University. High performance fortran language specification. *SIGPLAN Fortran Forum*, 12(4):1–86, 1993.

[187] Lorna Smith and Mark Bull. Development of mixed mode mpi / openmp applications. *Sci. Program.*, 9(2,3):83–98, 2001.

[188] Pieter Bellens, Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. Memory – CellSs: a Programming Model for the Cell BE Architecture. In *SC*, page 86, 2006.

[189] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Productive Cluster Programming with OmpSs. In *Euro-Par (1)*, pages 555–566, 2011.

[190] Khronos OpenCL Working Group. The OpenCL Specification, Version 1.2. https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, October 2014. Aaftab Munshi eds.

[191] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[192] nVidia Corp. CUDA Technology. http://www.nvidia.com/CUDA, September 2008.

[193] Khronos WG. OpenCL–The Open Standard for Parallel Programming of Heterogeneous Systems. http://www.khronos.org/opencl/, Nov 2011.

[194] OpenACC.org. The OpenACC™ Application Programming Interface, Version 2.0. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf, August 2013.

[195] Khronos OpenCL Working Group – SYCL subgroup. SYCL™ Specification, Version 1.2. https://www.khronos.org/registry/sycl/specs/sycl-1.2.pdf, September 2014. Lee Howes and Maria Rovatsou eds.

[196] Microsoft Corporation. C++ AMP: C++ Accelerated Massive Parallelism, Version 1.2. http://download.microsoft.com/download/4/0/E/40EA02D8-23A7-4BD2-AD3A-0BFFFB640F28/CppAMPLanguageAndProgrammingModel.pdf, December 2013.

[197] Ehsan Totoni, Babak Behzad, Swapnil Ghike, and Josep Torrellas. Comparing the power and performance of intel's scc to state-of-the-art cpus and gpus. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*, ISPASS '12, pages 78–87, Washington, DC, USA, 2012. IEEE Computer Society.

[198] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, February 2011.

[199] Morris Jette and Mark Grondona. Slurm: Simple linux utility for resource management. In *ClusterWorld Conference and Expo*, 2003.

[200] Judit Planas Carbonell. *Programming Models and Scheduling Techniques for Heterogeneous Architectures*. PhD thesis, Universitat Politècnica de Catalunya - UPC, 2015.

[201] Taeyoung Kim, Jintaek Kang, Sungchan Kim, and Soonhoi Ha. Sophy: A software platform for hybrid resource management of homogeneous many-core accelerators. In *Proceedings of the 3rd International Workshop on Many-core Embedded Systems*, MES '15, pages 17–24, New York, NY, USA, 2015. ACM.

[202] Patrick Bellasi, Giuseppe Massari, and William Fornaciari. Effective runtime resource management using linux control groups with the barbequertrm framework. *ACM Trans. Embed. Comput. Syst.*, 14(2):39:1–39:17, March 2015.

[203] Arvind Sridhar, Alessandro Vincenzi, David Atienza, and Thomas Brunschwiler. 3d-ice: A compact thermal model for early-stage design of liquid-cooled ics. *IEEE Transactions on Computers*, 63(10):2576–2589, 2014.

[204] Martin Aigner, Christoph M. Kirsch, Michael Lippautz, and Ana Sokolova. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 451–469, New York, NY, USA, 2015. ACM.

[205] Dionysios Diamantopoulos, S. Xydis, K. Siozios, and D. Soudris. *Dynamic Memory Management in Vivado-HLS for Scalable Many-Accelerator Architectures*, pages 117–128. Springer International Publishing, Cham, 2015.

[206] Michael J. Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. The accelerator store: A shared memory framework for accelerator-based systems. *ACM Trans. Archit. Code Optim.*, 8(4):48:1–48:22, January 2012.

[207] Carlos Villavieja, Yoav Etsion, Alex Ramirez, and Nacho Navarro. *FELI: HW/SW Support for On-Chip Distributed Shared Memory in Multicores*, pages 282–294. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[208] Andrea Di Biagio, Ettore Speziale, and Giovanni Agosta. *Exploiting Thread-Data Affinity in OpenMP with Data Access Patterns*, pages 230–241. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[209] Houjun Tang, Xiaocheng Zou, John Jenkins, A David, II Boyuka, Stephen Ranshous, Dries Kimpe, Scott Klasky, and Nagiza F

Samatova. Improving read performance with online access pattern analysis and prefetching. In *Euro-Par*, pages 246–257, 2014.

[210] Julien Mottin, Mickael Cartron, and Giulio Urlini. *The STHORM Platform*, pages 35–43. Springer New York, New York, NY, 2014.

[211] I. Koutras, A. Bartzas, and D. Soudris. Efficient memory allocations on a many-core accelerator. In *ARCS 2012*, pages 1–6, Feb 2012.

[212] C. Silvano, W. Fornaciari, S. C. Reghizzi, G. Agosta, G. Palermo, V. Zaccaria, P. Bellasi, F. Castro, S. Corbetta, E. Speziale, D. Melpignano, J. M. Zins, D. Siorpaes, H. Hübert, B. Stabernack, J. Brandenburg, M. Palkovic, P. Raghavan, C. Ykman-Couvreur, A. Bartzas, D. Soudris, T. Kempf, G. Ascheid, H. Meyr, J. Ansari, P. Mähönen, and B. Vanthournout. Parallel paradigms and run-time management techniques for many-core architectures: The 2parma approach. In *2011 9th IEEE International Conference on Industrial Informatics*, pages 835–840, July 2011.