Title

Permalink
https://escholarship.org/uc/item/6n93v21h

Author
Muck, Tiago Rogerio

Publication Date
2018

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA, 
IRVINE


DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Tiago Rogério Mück

Dissertation Committee:
Professor Nikil Dutt, Chair
Professor Alex Nicolau
Professor Tony Givargis

2018
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF ALGORITHMS</td>
<td>viii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>ix</td>
</tr>
<tr>
<td>CURRICULUM VITAE</td>
<td>x</td>
</tr>
<tr>
<td>ABSTRACT OF THE DISSERTATION</td>
<td>xiii</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 The problem .................................................. 3  
1.2 Proposed solution ............................................. 6  
1.2.1 Thesis contributions ...................................... 7  

## 2 Background and Related Work

2.1 Reflection in computer systems ................................. 10  
2.2 Cooperative on-chip resource management ..................... 11  
2.3 ODA and predictive resource management vs reflection ...... 13  
2.4 Self-adaptive systems ........................................ 15  

## 3 A Framework for Adaptive-Reflective Resource Management

3.1 MARS interfaces ................................................. 21  
3.2 MARS internals ................................................ 25  
3.2.1 Reflection flow summary .................................. 30  
3.3 High-level policy manager .................................... 31  

## 4 MARS implementation on Linux

4.1 Evaluation platforms .......................................... 35  
4.1.1 ODROID-XU3 .................................................. 35  
4.1.2 GEM5 ......................................................... 36  
4.2 Interface overheads ............................................ 38
# MARS offline simulator

5.1 Why an offline simulator ........................................ 39
5.2 Offline simulator implementation ................................ 41
5.3 Offline simulator limitations ..................................... 43
5.4 Validation experiments ........................................... 43
5.5 Related work on simulation of resource management policies ........................................ 45

# MARS’s Performance/Power Model

6.1 The need for performance/power modeling ....................... 47
6.2 Model definitions .................................................. 49
   6.2.1 Architecture model ......................................... 49
   6.2.2 Workload model ............................................. 50
   6.2.3 Execution model ............................................. 50
   6.2.4 Task characterization model ................................ 50
   6.2.5 Prediction model ............................................. 52
6.3 IPC and power prediction .......................................... 53
   6.3.1 Bin-based prediction ......................................... 54
   6.3.2 Bin-based predictor training ................................ 55
6.4 Predictor training .................................................. 56
   6.4.1 Microbenchmarks ............................................. 57
6.5 Task load prediction under resource sharing ..................... 60
   6.5.1 Load contribution estimation ............................... 60
   6.5.2 Load prediction under CFS .................................. 61
6.6 Accuracy evaluation ............................................... 63
6.7 Discussion and limitations ....................................... 65

# Energy-efficient Task Mapping for Heterogeneous Multiprocessors

7.1 SPARTA rationale .................................................. 68
7.2 Implementation on MARS ........................................... 70
   7.2.1 DVFS model .................................................. 71
   7.2.2 Task allocation algorithm ................................... 72
7.3 Experimental evaluation ........................................... 74
   7.3.1 Workloads .................................................... 75
   7.3.2 Simulation results ........................................... 76
   7.3.3 ODROID platform results .................................... 79
   7.3.4 Evaluation summary .......................................... 81
7.4 Summary and related work ........................................ 81
   7.4.1 Further reading .............................................. 82

# Aging-Aware Load Balancing in Mobile Platforms

8.1 Motivation and Related work .................................... 86
8.2 Aging modeling .................................................... 87
   8.2.1 System-level delay degradation model ..................... 87
8.3 Aging-aware task mapping ....................................... 90
   8.3.1 Task mapping heuristic ...................................... 91
8.4 Experimental results .................................................. 93
  8.4.1 Setup ........................................................................ 93
  8.4.2 Workloads ............................................................... 95
  8.4.3 Results and discussion ............................................. 96
8.5 Summary and conclusions ............................................ 99
8.6 Related work ............................................................. 99

9 Design Space Exploration of HMPs ................................. 102
  9.1 Asymmetric HMP Architecture and Workloads .............. 105
  9.2 Experimental Results and Discussion ............................ 108
  9.3 Summary and Conclusions ........................................... 111

10 Conclusions ............................................................... 113

Bibliography .................................................................. 116
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Top 100 systems on the GREEN500 supercomputer ranking</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>ARM’s big.LITTLE HMP architecture</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Trends of Smartphone Applications</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>Linux’s CPU main resource allocation and power management subsystems</td>
<td>4</td>
</tr>
<tr>
<td>1.5</td>
<td>Motivational example</td>
<td>5</td>
</tr>
<tr>
<td>1.6</td>
<td>Task mapping interacting with other policies’ models</td>
<td>7</td>
</tr>
<tr>
<td>1.7</td>
<td>MARS Overview</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>ODA-based approach for implementing tasks mapping and DVFS vs MARS</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>CPSOC architecture</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Policies make decisions at different granularities</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>MARS framework overview</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Task mapping policy implemented on MARS</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>MARS interfaces UML diagram</td>
<td>21</td>
</tr>
<tr>
<td>3.5</td>
<td>Reflective DVFS management policy</td>
<td>23</td>
</tr>
<tr>
<td>3.6</td>
<td>System info example for an Exynos5422 HMP</td>
<td>25</td>
</tr>
<tr>
<td>3.7</td>
<td>Implementation of <em>sense</em> and <em>actuate</em></td>
<td>26</td>
</tr>
<tr>
<td>3.8</td>
<td>Specialization of <em>SensingTypeInfo</em></td>
<td>26</td>
</tr>
<tr>
<td>3.9</td>
<td>MARS internal components UML diagram</td>
<td>27</td>
</tr>
<tr>
<td>3.10</td>
<td>Task mapping policy example</td>
<td>28</td>
</tr>
<tr>
<td>3.11</td>
<td>UML sequence policy diagram for one iteration of <em>Simple_Map_Policy</em></td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Linux toolchain overview</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>Implementation of virtual interface functions in Linux</td>
<td>35</td>
</tr>
<tr>
<td>4.3</td>
<td>Odroid-XU3 block diagram</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>Offline simulation toolchain</td>
<td>40</td>
</tr>
<tr>
<td>5.2</td>
<td>Trace-based simulation flow</td>
<td>41</td>
</tr>
<tr>
<td>6.1</td>
<td>MARS overview</td>
<td>46</td>
</tr>
<tr>
<td>6.2</td>
<td>Performance of bodytrack and x264 PARSEC benchmarks</td>
<td>47</td>
</tr>
<tr>
<td>6.3</td>
<td>Correlation between various performance metrics and IPS.</td>
<td>54</td>
</tr>
<tr>
<td>6.4</td>
<td>Example of workload classification for performance and power prediction</td>
<td>55</td>
</tr>
<tr>
<td>6.5</td>
<td>Model training toolchain</td>
<td>57</td>
</tr>
<tr>
<td>6.6</td>
<td>Average IPC and load of PARSEC benchmarks and microbenchmarks</td>
<td>58</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Examples of sensors and actuators</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Exynos 5422 core parameters</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>Core parameters for gem5 simulations</td>
<td>37</td>
</tr>
<tr>
<td>4.3</td>
<td>Linux interface overheads</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Offline simulation accuracy</td>
<td>44</td>
</tr>
<tr>
<td>7.1</td>
<td>Benchmark Mix composition</td>
<td>75</td>
</tr>
<tr>
<td>8.1</td>
<td>HMP core parameters</td>
<td>94</td>
</tr>
<tr>
<td>8.2</td>
<td>Chip area and thermal specification</td>
<td>94</td>
</tr>
<tr>
<td>8.3</td>
<td>Aging model parameters</td>
<td>95</td>
</tr>
<tr>
<td>8.4</td>
<td>Workload patterns</td>
<td>96</td>
</tr>
<tr>
<td>9.1</td>
<td>Heterogeneous cpu parameters</td>
<td>106</td>
</tr>
<tr>
<td>9.2</td>
<td>Heterogeneous cache parameters</td>
<td>106</td>
</tr>
<tr>
<td>9.3</td>
<td>Benchmark Mix composition</td>
<td>107</td>
</tr>
<tr>
<td>9.4</td>
<td>Most energy-efficient configurations for each workload</td>
<td>109</td>
</tr>
<tr>
<td>No.</td>
<td>Algorithm</td>
<td>Page</td>
</tr>
<tr>
<td>-----</td>
<td>----------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>Classification bins generation</td>
<td>56</td>
</tr>
<tr>
<td>2</td>
<td>Task load contribution estimation</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>Load share prediction</td>
<td>62</td>
</tr>
<tr>
<td>4</td>
<td>SPARTA Heuristic</td>
<td>73</td>
</tr>
<tr>
<td>5</td>
<td>ADAMANT task mapping algorithm</td>
<td>92</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I would like to thank everyone who has supported me during all these years of work that culminated in this dissertation. I would like to give special thanks to my advisor, Prof. Nikil Dutt, for giving the opportunity to work under his guidance and for all the help in the pursuit of my goals. I would like to thank for the support of all my colleagues and friends at UCI, in particular, I thank Abbas Banaiyan, Prof. Antonio Augusto Frohlich, André Luís del Mestre Martins, Prof. Amir Rahmani, Prof. Alex Nicolau, Biswadip Maity, Bryan Donyanavard, Prof. Bruno Zatt, Delaram Amiri, Emad Kasaeyan Naeini, Prof. Eli Bozorgzadeh, Hamid Nejatollahi, Prof. Hyoungh Hwang, Hirak Kashyap, Hossein Tajik, Jurngyu Park, Kasra Moazzemi, Majid Namaki Shoushtari, Roger Chen-Ying Hsieh, Sina Labbaf, Santanu Sarma, Prof. Sung-Soo Lim, Sajjad Taheri, Prof. Tony Givargis, Zana Ghaderi, and Zhi Chen, for their contributions, interesting discussions, and all great moments during these years.

I would like to acknowledge the funding assistance that made this work possible: the Brazilian government and CAPES though the Science Without Borders program for 4 years of financial support, and the Computer Science Department at the Donald Bren School of Information and Computer Sciences. This work was also partially supported by NSF grant CCF-1704859.
CURRICULUM VITAE

Tiago Rogério Mück

EDUCATION

Ph.D. in Computer Science 2018
University of California, Irvine
Irvine, CA

M.Sc. in Computer Science 2013
Federal University of Santa Catarina
Florianópolis, SC, Brazil

B.Sc. in Computer Science 2010
Federal University of Santa Catarina
Florianópolis, SC, Brazil

WORK AND RESEARCH EXPERIENCE

Graduate Research Assistant 2013–2018
University of California, Irvine
Irvine, CA

Research Intern 2017
ARM, Inc.
Austin, TX

Interim Engineering Intern 2016
Qualcomm Technologies, Inc.
San Diego, CA

Graduate Research Assistant 2010–2013
Federal University of Santa Catarina
Florianópolis, SC, Brazil

Visiting Student Researcher 2010–2011
Friedrich-Alexander-University of Erlangen-Nürnberg,
Erlangen, Germany

Undergraduate Research Assistant 2008–2009
Federal University of Santa Catarina
Florianópolis, SC, Brazil

Hexagon Agriculture
Florianópolis, SC, Brazil

TEACHING EXPERIENCE

Teaching Assistant 2014–2018
University of California, Irvine
Irvine, CA

REFEREED PUBLICATIONS
Adaptive-Reflective Middleware for Power and Energy Management in Many-Core Heterogeneous Systems 2018

CHIPS-AHOy: A Predictable Holistic Cyber-Physical Hypervisor for MPSoCs 2018

Design Methodology for Responsive and Robust MIMO Control of Heterogeneous Multicores 2018

Design methodologies for enabling self-awareness in autonomous systems 2018

Gain scheduled control for nonlinear power management in CMPs 2018

SPECTR: Formal Supervisory Control and Coordination for Many-core Systems Resource Management 2018

Exploring fast and slow memories in HMP core types 2017

PoliCym: Rapid prototyping of resource management policies for HMPs 2017

Exploiting Heterogeneity for Aging-Aware Load Balancing in Mobile Platforms 2017

SPARTA: Runtime Task Allocation for Energy Efficient Heterogeneous Many-cores 2016

Cross-layer virtual/physical sensing and actuation for resilient heterogeneous many-core SoCs 2016
Run-DMC: Runtime dynamic heterogeneous multicore performance and power estimation for energy efficiency
Proceedings of the Tenth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis - CODES ’15

SmartBalance: A Sensing-Driven Linux Load Balancer for Energy Efficiency of Heterogeneous MPSoCs
Proceedings of the 52nd Annual Design Automation Conference on - DAC ’15

SOFTWARE

MARS  https://github.com/duttresearchgroup/sparta
Middleware for Adaptive Reflective computer Systems
ABSTRACT OF THE DISSERTATION


By

Tiago Rogério Mück

Doctor of Philosophy in Computer Science

University of California, Irvine, 2018

Professor Nikil Dutt, Chair

Emerging mobile SoCs are increasingly incorporating heterogeneity in order to provide energy-efficiency while meeting performance requirements. Effective exploitation of power-performance tradeoffs in heterogeneous many-core platforms (HMPs), however, requires intelligent management at different layers, in particular at the operating system level. Operating systems need to continuously analyze the application behavior and find a proper answer for questions such as: What is the most power efficient core type to execute the application without violating its performance requirements? or Which option is more power-efficient for the current application: an out-of-order core at a lower frequency or an inorder core at a higher frequency? Unfortunately, existing operating systems do not offer mechanisms to properly address these questions and therefore are unable to fully exploit architectural heterogeneity for scalable energy-efficient execution of dynamic workloads.

In this thesis we present our vision of a holistic approach for performing resource allocation decisions and power management by leveraging concepts from reflective software. The general idea of reflection is to change your actions based on both external feedback and introspection (i.e., self-assessment). From a system perspective, this translates into performing resource management actuation considering both sensing information (e.g., readings from performance
counters, power sensors, etc.) to assess the current system state, as well as models to predict the behavior of the system before performing an action. In this context, this thesis makes the following contributions:

- It describes MARS, a Middleware for Adaptive Reflective computer Systems. MARS consists of a toolchain for creating resource managers that allows users to easily compose models and policies that interact in a hierarchy defined by the granularity of the actuations performed in the system. MARS is implemented and evaluated on top a real Linux-based platform. Furthermore, MARS also provides an offline simulation infrastructure for fast prototyping of policies and large-scale or long-term policy evaluation.

- It proposes a performance/power modeling approach for HMPs which takes into account the effect of both microarchitecture-level components as well as system-level components such as the operating scheduler.

- It proposes a runtime task mapping approach for energy efficient HMPs. Energy efficient task-to-core mapping is done by combining on-chip sensor data and models of the underlying operating systems components implemented within MARS. It achieves average energy reductions of 23% when compared to the state-of-art task mapping techniques, while achieving similar performance.

- It proposes aging models to provide reliability-aware task-mapping for mobile HMPs.

These contributions have shown that the extensive use of models to predict how the system will react to actuations is a promising scheme to pave the path towards more energy efficient heterogeneous systems. In this context, a framework which enables reflective resource management is an important building block to tackle the growing complexity of battery-powered devices.
Chapter 1

Introduction

The Moore’s law [77] predicts that the number of transistors in integrated circuits doubles about every two years. The Moore’s law has proved accurate for several decades, as advances from the semiconductor industry have created a long era of continuous improvements in computer systems performance, reduction in power consumption, and reduction in costs [58]. However, this trend of continuous improvement is coming to an end as one of the main mechanisms behind Moore’s law, the Dennard Scaling [28], no longer holds. Dennard Scaling states that transistors’ operating voltage scale down with transistors’ size. In the last decade, however, the operating voltage has not been significantly decreasing on new technologies due to leakage and reliability issues, thus it’s no longer possible to keep packing more transistors in the same chip area without significantly increasing the power density [35], leading to the so called “power wall”.

In order to avoid hitting the power wall, novel computer architectures and system-level design principles have been devised to improve power efficiency. In this context, heterogeneous processing elements have become increasingly popular as a way to extract more performance with a limited power envelope. This growth in popularity can be observed across multiple
domains, ranging from portable battery-powered devices to supercomputers. For instance, Figure 1.1 shows that computer systems with heterogeneous processing elements dominate the GREEN500 supercomputer ranking [114]. These computer systems typically combine general purpose cores with graphical processing units (GPUs). GPUs consume low energy per calculation and are particularly energy efficient for applications with very high and regular parallelism.

Figure 1.2: ARM’s big.LITTLE HMPs integrate high-performance and power-efficient cores for supporting both and high and low load scenarios, and a GPU for graphical and massively parallel workloads [4]
On the other end of the spectrum, emerging mobile battery-powered SoCs are also increasingly incorporating heterogeneity in order to provide energy-efficiency. Figure 1.2 shows ARM’s big.LITTLE architecture [3] used, for instance, on the latest Samsung’s Exynos [108] and Nvidia’s Tegra SoCs [89]. big.LITTLE employs single-ISA heterogeneity, in which cores of the same instruction-set architecture (ISA) are deployed with different configuration and/or microarchitectures (for example, in-order cores operating at low frequency vs. out-of-order cores operating at high frequency). In the mobile domain, this shift towards heterogeneity has also being motivated by the need to support highly diverse and complex workloads that typically exhibit dynamically varying resource demands. For example, Figure 1.3 shows a growing number of available applications on this devices.

1.1 The problem

Effective exploitation of power-performance tradeoffs in HMPs, however, requires intelligent management at different layers, in particular in the operating system [100]. Operating systems need to continuously analyze the application behavior and find a proper answer for questions such as: What is the most power efficient core type to execute the application without violating its performance requirements? or Which option is more power-efficient for
the current application: an out-of-order core at a lower frequency or an inorder core at a higher frequency? Linux is the prevalent operating system deployed in domains that use HMPs (e.g. as of 2017, Linux-based systems make up 99% of supercomputers [128] and 86% of mobile devices [41]). Unfortunately, Linux does not offer mechanisms to properly address these questions and therefore is unable to fully exploit architectural heterogeneity for scalable energy-efficient execution of dynamic workloads. Let’s consider, for instance, Linux’s main CPU resource allocation and power management subsystems shown in Figure 1.4. The scheduler (defines the task-to-core mapping), cpuidle (controls the processing unit’s power states) and cpufreq (performs dynamic voltage-frequency scaling — DVFS) subsystems work in isolation, at different time scales and often at cross-purposes with each other [68]. This lack of coordination may lead the system to an energy-inefficient state in which the scheduling and power management subsystems take antagonistic decisions.

Figure 1.4: Linux’s main CPU resource allocation and power management subsystems[68]

Some existing Linux extensions do address these issues to a limited extent. For instance, the Global Task Scheduling (GTS) [3], migrates tasks between high-performance and low-power core types when the task load reaches a certain threshold. GTS-based policies have been implemented on Linux and deployed by multiple SoC vendors that support the big.LITTLE technology (e.g., Linaro’s/ARM’s big.LITTLE MP implementation used by Samsung Exynos [96], MediaTek’s CorePilot [72], Qualcomm’s Energy Aware Scheduling [85]). These policies,
however, are customized for certain heterogeneous platforms without being adaptable to other platforms. Furthermore, they exclusively focus on task mapping/scheduling decisions and still do not provide any coordination with the the underlying power management subsystems.

![Graph showing throughput and power comparison between Linux and optimized configurations.]

Figure 1.5: Two concurrent instances of a multithreaded x264 application [10] running on an ODROID XU3 [48] with Linux’s ondemand DVFS governor and GTS scheduler [3]. The first instance runs with a input rate of 5 frame/s, while the second instance runs with 15 frames/s. The left-hand plots show the total throughput and power while both applications are running. The right-hand plot shows the same metric after the second instance finishes. The Optimal configuration is the configuration with the lowest power that has a similar or better IPS than the Linux baseline (found by exploring all possible task-to-core mappings and core frequency combinations.

In order to highlight the potential energy savings that can be achieved by, for instance, coordinating task mapping and DVFS, we perform a preliminary evaluation of the x264 application [10] on the ODROID XU3 platform [48], which has a 8-core big.LITTLE-based HMP (for more details about the ODROID platform, please refer to Chapter 4, Section 4.1). We run two separate multithreaded instances of x264 (with four threads each) running at different input rates, thus generating multiple tasks with fairly distinct load profiles. Figure 1.5 plots the total throughput and power of both applications and shows that the vanilla Linux configuration for HMP (which uses the GTS scheduler) consumes from 66% to 226% more power than the best task-to-core mapping and DVFS setting which achieves the same
level of throughput as vanilla Linux,

### 1.2 Proposed solution

This thesis presents a vision of a holistic approach for performing resource allocation decisions and power management by leveraging concepts from reflective software [123, 62, 13, 61, 16]. The general idea of reflection is to change your actions based on both external feedback and introspection (i.e., self-assessment). In our context, this translates into performing resource management actuation considering both sensing information (e.g., readings from performance counters, power sensors, etc.) to assess the current system state, as well as models to predict the outcome of the actuation before performing it.

By combining multiple predictive models, one can also get an estimation of how certain actuations impact other system components. For instance, Figure 1.6 illustrates a scenario in which a task mapping policy queries a DVFS policy model in order to make an informed mapping decision. Runtime task-to-core mapping and load balancing is typically performed in coarser time periods (e.g., 200ms) compared to DVFS (e.g., 50ms). A task mapping decision therefore affects DVFS actions which are typically driven by the core load (e.g., Linux’s on-demand governors), which is a function of the task mapping and the task scheduling (i.e., the time slice given to each task when multiple tasks are mapped to the same core). At the lowest level, a model of the underlying hardware platform can be used to predict other metrics such as power consumption. Given all the information collected across the model hierarchy, the task mapping can then make the appropriate decision that optimizes the system towards the desired goal.
1.2.1 Thesis contributions

In this scenario, this thesis proposes MARS, a Middleware for Adaptive Reflective computer Systems. MARS consists of a toolchain for creating reflective resource managers. With MARS, users can compose models and policies that interact in a hierarchy defined by the granularity of the actuations performed in the system. Figure 1.7 gives an overview of MARS.

At its heart, the reflective model framework provides policies with system introspection that is necessary to take informed decisions and provide run-time adaptability. Policies and
models obtain sensing information (e.g. performance counters, power, temperature, etc.) and perform actuations (e.g. changing core frequency, changing task mapping) through the Virtual Sensing/Actuation Interface. This interface abstracts away platform-specific details and allows policies to be deployed, unmodified, on top of any platform supported by the interface.

The remaining of this thesis provides a detailed description of MARS’s contributions and the case studies implemented using MARS:

- **Chapter 2** provides a background on the use of reflection and positions MARS with respect to these works, as well as the state-of-the-art on on-chip resource management.

- In **Chapter 3**, we take a deep dive into the framework and implementation of reflective resource management policies.

- As shown in Figure 1.7, MARS components lie within middleware between between the user applications and the underlying platform. The sensing/actuation interface can be implemented to interact with the platform either directly or through an existing operating system. **Chapter 4** describes the interface implementation on top of Linux systems.

- This thesis also presents an offline trace-based simulator that allows policies to be quickly prototyped, debugged, and validated on top of arbitrary platform configurations. **Chapter 5** describes the offline simulation infrastructure contributions.

- **Chapter 6**, describes MARS’s proposed baseline model and evaluate its accuracy. Other than the policy models, the baseline model is fundamental component of MARS’s reflection mechanism (presented in Chapter 3).

- **Chapter 7** presents an energy-efficient task mapping approach on heterogeneous architectures. We leverage models of operating system components such as the DVFS
governors and the scheduler to predict performance and power while exploring new task mappings (similar to the flow shown in Figure 1.6). Chapter 7 describes the task mapping heuristic in details.

- **Chapter 8** extends the task mapping policy from Chapter 7 with aging-awareness to extend the lifetime of mobile devices. Chapter 8 introduces device aging models and uses the offline simulation infrastructure to evaluate the effect of task mapping policies on long-term device aging.

- **Chapter 9** leverages the offline simulation infrastructure to perform a design space exploration of novel HMP architectures. This exploration shows possible opportunities of further improvements in energy efficiency by increasing heterogeneity on single-ISA HMPs beyond the dual-microarchitecture model used in big.LITTLE.

The cases studies show that MARS is a promising scheme to pave the path towards more energy efficient heterogeneous systems. While the case studies presented in this thesis focus mostly on mobile scenarios, we believe that MARS’s approach can be applied to a wider range of systems (e.g. server, HPC) and also opens up several directions for future work. These directions are presented with this thesis’ conclusions in **Chapter 10**.
Chapter 2

Background and Related Work

This thesis is not the first work to utilize the term “reflection” in the context of computer systems, as well as predictive/model-based resource management. This chapter provides a background on the use of reflection and positions MARS with respect to these works, as well as the state-of-the-art on on-chip resource management.

2.1 Reflection in computer systems

Reflection can be defined as the capability of a system to reason about itself and act upon this information[123]. A reflective system usually maintain a representation of itself within the underlying system, which is used for reasoning. In many works on reflective systems, this representation is also referred to as meta level, while and the system itself as base level[123, 62, 13, 61, 16]. Operations to introspect and modify the meta level are commonly referred to as the Meta Object Protocol (MOP) [61]. Reflection first emerged in the context of object-oriented programming languages to support the design of more flexible systems. Examples of reflective languages include: CLOS[59], Java[91], OpenC++[23], and Python[36].
In this context reflection provides means for programmers to programatically inspect the structure of objects (e.g., data members available in a class, their type, available functions, etc.) and modify them.

Middlewares for developing distributed applications is another field where reflection has been applied. Earlier use of reflection in this context can be seen in OpenORB[13] and DynamicTAO[62]. In these middlewares, reflection is used as a principled (as opposed to ad-hoc) mechanism for introspection and adaptation to changes in the network environment. Other works on this track include [16] [71] [66] [39] [125] [45] [104]

These works provide greater adaptability to distributed middlewares, however they cannot be directly used in the context of on-chip resource management. The reflection mechanisms used in these works impose a rather heavy workload that would cause significant performance deterioration in devices with limited resources and there is always a trade-off issue between performance and scope of adaptability[45]. Policies implemented using MARS are meant to augment or replace mechanisms typically implemented within the operating systems and time sensitive. Thus, a light-weight approach is needed to allow reflection for such policies. To the best of our knowledge, MARS is the first framework that incorporates reflection at the time scale of operating system components.

2.2 Cooperative on-chip resource management

While a significant amount of work has been done on the management of specific resources (e.g. DVFS[22, 88, 27, 115, 92, 31], task mapping[93, 26, 103, 44, 106, 134, 116, 129], memory allocation[54, 83, 117, 118]), most do not provide means for coordinating the management of different resource types.

There have been, however, other works that target coordinated management of resources.
For instance, [74][70][86] propose unified approaches for jointly performing task mapping and DVFS. These works propose ad-hoc heuristics that would need to be redesigned in order to take into account different or additional resources. MARS, on the other hand, allows multiple resource management policies to be seamless integrated through reflective models.

Some works use Machine Learning techniques for resource management. [12] uses Artificial Neural Networks (ANN) to manage multiple shared on-chip resources in a coordinated fashion to achieve a high-level objective. This method provides a formal reasoning and data driven identification of relationships that can be used for tuning system configurations at runtime. However, it requires a learning phase with a large amount of measurement data to prepare the predicting algorithm for a specific platform and does provide robustness against corner cases not included in the training. More recent work, proposes resource management in cloud systems using deep reinforcement learning[137] and as a learning tool for improving efficiency of control theoretic approaches[76].

[97] and [99] use Multiple-Output Multiple-Input (MIMO) control theory[122] for managing on-chip resources. MIMO is a robust solution and provides a systematic way incorporate new resources into the design. However, MIMO is not scalable[82] and cannot handle discrete system logic. [99] addresses the scalability issue by using Supervisory Control Theory (SCT)[101] to coordinate a hierarchy of MIMO controllers. Single-Input Single-Output control theory has also being used in this context. For instance, [53] and [76] use a single control signal which is used as input for a look-up table that selects the proper allocation. Control theory based solutions strong points are their formalism and robustness, however they have limitations in the sense that they can monitor a very limited number of signals sensed from the system. MIMO does not scale well as more inputs are added and SISO supports a single input. SCT-based approaches can address this problem by using multiple MIMOs as shown in [99], however the design presented in [99] consists of multiple homogeneous controllers (all controllers manage the same type of resource — core frequency and number of active
cores) and it’s not clear how to systematically incorporate new resource types since the SCT manager is manually tailored for each system. A related work that provides coordination of multiple control loops is presented in [129]. An *Orchestrator* is proposed to decide which policy should be selected, however, it also does not handle conflicting policies managing different resource types.

### 2.3 ODA and predictive resource management vs reflection

MARS is the first application of reflection to perform low-level resource management (e.g. task mapping and DVFS). However, many other works use models for performance/power predictions to guide their resource management decisions, specially in the domain of task mapping for HMP platforms. For instance, [63, 105, 130, 2, 69, 70, 111] all propose task mapping heuristics that use offline-trained models to predict the performance and/or power of a new task mapping based on current sensed data. These works implement the *Observe-Decide-Act* (ODA) paradigm\(^1\)[52], which can be seen as a form of reflection (despite the absence of the term within these works). However, there is a fundamental difference between MARS reflection mechanisms and ODA. In ODA decisions are made based either on past observations or predictions made from past observations. These predictions do not consider future actions or events that could happen until the next iteration of the ODA loop. These actions are usually triggered by other resource managers running at a finer time scale than the ODA loop (recall the Task Mapping vs DVFS example from Chapter 1). Without this high-level reasoning, one can consider ODA loops *reflective* as opposed to *reflective*. Figure 2.1 illustrates this distinction.

\(^1\)control theory approaches also fall into this category
Consider two resources managers: one for task mapping and one for DVFS, both implemented as ODA loops in which a decision is made periodically. The Observe phase collects sensed data from the previous period, which is used for decision making (Figure 2.1a), i.e., decisions are always taken based on past behavior. In this case of the task mapping ODA loop, future invocations of the underlying DVFS ODA loop are not taken into account. In MARS (Figure 2.1b, the decision making is also able to issue queries such as “what if the task mapping changes”, which return predicted observed data for the next period that automatically takes into account the underlying policies.
2.4 Self-adaptive systems

Self-adaptive systems have flourished as a research area since IBM’s autonomic computing initiative [57] and have become even more prevalent today with the popularity of the *Internet of Things*. Self-adaptive software can be defined as “software that evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible”[64]. This thesis does not address self-adaptive systems directly, as this systems encompass many high-level concepts such as self-configuration, self-healing, self-awareness, self-optimization, and others, also referred to as *self-*[107] ([32] has a comprehensive review self-adaptiveness). However, an infrastructure for system introspection and reflective behavior is an important building block for such systems [21], which highlights the potentials contributions of MARS in this context.

Previous work in this area proposed models and introspection mechanisms that are more general and aimed towards designing self-adaptive applications [55, 110, 109], while MARS focuses explicitly on managing on-chip resources and is transparent to applications. MARS is more closely related to the concepts proposed by the *Cyber-Physical System-on-Chip*(CPSoC)[111]. CPSoC (Figure 2.2) combines a sensor-actuator rich Computing-Communication-Control (C3) centric paradigm[65] with that of an adaptive and reflective middleware (a flexible hardware-software stack and interface between the application and OS layer) to control the manifestations of computations (e.g., aging, overheating, parameter variability, etc.) on the physical characteristics of the chip itself and the outside interacting environment.

CPSoC’s proposed reflective middleware consists of multiple ODA loops (in a manner similar to [99] which proposes multiple MIMO controllers). In this context, MARS is able to further extend the capabilities of CPSoC by providing a systematic way of ODA loop coordination.
Figure 2.2: CPSoC architecture with adaptive Core, NoC, and the Observe-Decide-Act Loop as Adaptive, Reflexive Middleware [111]
Chapter 3

A Framework for Adaptive-Reflective Resource Management

In this chapter we describe the main concepts proposed by MARS. MARS consists of a toolchain for creating resource managers that allows users to easily compose models and policies in a coordinated manner. The principle of this coordination is based on the assumption that resource management policies run at different granularities. Consider, for instance, Figure 3.1, which shows the typical timeframe for task mapping, DVFS, and scheduling.

Figure 3.1: Resource management policies make decisions at different granularities

At the finest granularity, we have the operating system scheduler, whose goal is to pick a task to execute on a given core. A new decision must be made whenever a new task is created, a tasks finishes, a task’s time slot expires, an interrupt is raised, etc., leading to a
timeframe between decisions in the order of a few microseconds. At a coarser granularity we have the DVFS policy, which typically executes periodically to analyze the system load and select a new appropriate frequency. The DVFS period is typically in the order of 10-100 milliseconds. At the coarsest granularity, the task mapping policy runs periodically to define a new task-to-core assignment. Migrating a task from a core to another has significantly more overhead than changing the CPU frequency in a typical HMP [135], therefore this policy period is usually in the order of 100-1000 of milliseconds. In this scenario, it’s clear that an informed task mapping decision, for instance, has to take into account the effects of its changes on the behavior of the underlying DVFS and scheduling policies (as already briefly discussed in Chapter 2, Section 2.3). Furthermore, the granularity of actuations dictates how complex these policies can be. For instance, a scheduling decision has to be taken in the sub microsecond range in order not to disrupt the system. Task-to-core mapping, on the other hand, affects the system performance over a long timespan, therefore the overheads of using complex models to make such decisions can be mitigated by the potential benefits of an informed decision.

![MARS framework overview](image)

**Figure 3.2: MARS framework overview**

Based on these observations, we designed MARS as shown in Figure 3.2. MARS interacts with all layers of the system stack to orchestrate the management of resources. MARS is
mainly composed of four parts:

1. **Sensors and Actuators.** The sensed data that consists of performance counters (e.g. instructions executed, cache misses, etc) and other sensory information (e.g. power, temperature, etc.) collected to assess the current system state and to characterize workloads. Table 3.1 provides a list of typical sensors and actuators available at each level of the system stack. As shown in Section 3.1, sensing data is read and actuations are issued though a *virtual sensing interface* that provides platform-independent sensor and actuator abstractions.

2. **Resource management policies** implemented by MARS’s users

3. **Reflective system model** used by the policies to make informed decisions. The reflective model has the following subcomponents:

   (a) Models for *policies implemented by the underlying OS kernel* used for coordinating decisions taken within MARS with decisions taken by the OS.

   (b) Models for *user policies*, that are automatically instantiated from any policy defined within MARS.

   (c) The baseline *performance/power model*. This model takes as input the predicted actuations generated from the policy models and produces predicted sensed data.

4. **The high-level policy manager** that is responsible for reconfiguring the system by adding, removing, or swapping policies to better achieve the current system goal.

The components within the reflective system model interact in a hierarchy defined by the granularity of the actuations performed in the system, as discussed previously. For instance, Figure 3.3 illustrates the scenario from Figure 3.1 cast within MARS. When the task mapping policy wishes to migrate a task, it first submits a query the reflective model asking *what’s the performance of task A if it is migrated?* (1). Given the new migration, a model for the DVFS
Table 3.1: Examples of sensors and actuators available across the system stack

<table>
<thead>
<tr>
<th>Level</th>
<th>Actuators (A)</th>
<th>Sensors (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application (A)</td>
<td>Degree of parallelism, algorithmic choice</td>
<td>Performance, Quality-of-Service</td>
</tr>
<tr>
<td>Kernel (K)</td>
<td>Task-to-core mapping, task priority, memory allocation</td>
<td>CPU time, utilization, context switch counters, open file, number of tasks</td>
</tr>
<tr>
<td>Hardware (K)</td>
<td>Voltage/frequency scaling, clock gating, power gating, cache sizing</td>
<td>Performance counters (e.g. cache misses, instructions executed, branch misspredictions, etc), power, reliability, critical path delay degradation</td>
</tr>
</tbody>
</table>

governor used by the OS is executed to predict the core frequency (2). This information is then passed on to the performance/power model which predicts the task performance (3). Finally, the predicted metrics are used by the policy to perform the actuation through the actuation interface (4).

Figure 3.3: Task mapping policy implemented on MARS interacting with a model of the DVFS policy used by the OS kernel

The remaining of this Chapter describes MARS’s components in more details. Section 3.1 describes the sensing/actuations interfaces and the implementation of policies on top of MARS. Section 3.2 describes how the policy models interact with the performance/power model to implement reflection queries. Section 3.3 describes the high-level policy manager.
3.1 MARS interfaces

The first step to enable such adaptive-reflective middleware is providing a framework with a well-defined, simple, and portable interface to enable designers to develop their own resource management policies and policy models. Figure 3.4 shows a UML class diagram with a simplified overview of the framework and the provided interface.

MARS is implemented in the C++ language and follows an object-oriented paradigm. User-defined policies and models defined by creating subclasses of Policy and Model, respectively. Notice that Policy is also a subclass of Model, which allows user-defined policies to be also executed as models to predict future actuations. A user-defined subclass of Policy_Manager is responsible for composing policies and models together. Portability is provided by having Model inherit all platform-dependent implementation details from

![Figure 3.4: MARS interfaces UML diagram. Grayed components define policies implemented by the user.](image-url)
Sensing_Actuation_Interface. Model also inherits from Reflective_Interface the functions necessary for issuing queries to the reflective models. We support two different implementations of Sensing_Actuation_Interface: 1) a lightweight Linux-based implementation for deployment on real systems; and 2) an offline implementation for debugging and simulating executions under the control of the management policies. We provide more details on the Linux-based implementation as well as our motivations for an offline interface in Chapters 4 and 5, respectively.

In order to describe the functions provided by Sensing_Actuation_Interface, Reflective_Interface, and Policy_Manager, we show the implementation of a simple reflective DVFS policy in Figure 3.5. Simple_DVFS_Policy attempts to find the most energy-efficient (defined in terms of instructions executed per second (IPS) per Watt of power consumed) frequencies for the system. For simplicity, in this example, we simply try all available frequencies and use feedback from the reflective model to select the most energy-efficient one.

The first component the new manager class needs to implement is a mandatory setup() function which responsible for creating and registering which policies will be executed (using the registerModel function). At this point user-defined models (if any) are also registered (using the registerModel function). The actual DVFS policy is defined by the Simple_DVFS_Policy. Simple_DVFS_Policy inherits from Policy and implement its behavior in the execute function (lines 13-29). When inheriting from Policy one must also define the policy invocation period (line 4). This period defines how often the policy is invoked as well as the hierarchy of the policies within the reflective model. The period also establishes a sensing window for the policy. In this example, when the execute function is invoked, the sensed information (e.g., performance counters, power sensors, etc.) aggregated over the latest 50ms sensing window can be obtained.

Within the execute function, a special sys_info object provided by the framework is used to have access to the resource available in the system. The sys_info also provides information
```cpp
class Simple_DVFS_Policy : public Policy {
const int PERIOD_MS = 50;

Simple_DVFS_Policy() : Policy(PERIOD_MS) {}

double try_frequency(frequency_domain &fd, double freq_mhz) {
  tryActuate<ACT_FREQ_MHZ>(fd, freq_mhz);
  double ips = senseIf<SEN_INSTR_TOTAL>(fd) / senseIf<SEN_TIME_TOTAL_S>(fd);
  double power = senseIf<SEN_POWER_W>(fd);
  return ips / power;
}

void execute() {
  for(auto fd : sys_info().freq_domains) {
    double best_freq = fd.freq_min;
    double best_eff = 0;
    auto ranges = actuationRanges<ACT_FREQ_MHZ>(fd);
    for(auto freq = ranges.min; freq <= ranges.max; freq += ranges.steps) {
      double freq_eff = try_frequency(fd, freq);
      if(freq_eff > best_eff) {
        best_freq = freq;
        best_eff = freq_eff;
      }
    }
    actuate<ACT_FREQ_MHZ>(fd, best_freq);
  }
}

class Simple_DVFS_Manager : public Policy_Manager {
  void setup() {
    registerPolicy(new Simple_DVFS_Policy());
  }
};
```

Figure 3.5: Reflective DVFS management policy

about the topology and relationship between resources. Figure 3.6 illustrates the system view provided by the `sys_info` object for one the platform we used through Chapter 6–9. In Figure 3.6, the policy iterates through all `frequency domain resources` (Line 14). Then, for all supported frequencies, the `try_frequency` (Lines 6-11) function is called to assess the energy efficiency of the given frequency and select the best setting. The `try_frequency` function makes use of the `actuate/tryActuate` and `sense / senseIf` to perform the following assessment:

- **actuate**: this function sets a new value for a system actuator. It takes as arguments a reference to the system resource to be actuated upon and the new actuation value. The function’s template parameter selects the proper function implementation to perform the requested actuation action. Two additional related functions are also provided:
actuationVal returns the current actuation value set and actuationRanges return the valid range of actuations values for the specified resource.

- **sense**: this function takes a reference of a system resource and returns the sensed metric for that resource during the sensing window period defined for the current policy being executed. Similar to actuate, the function’s template parameter selects the proper function implementation to obtain the requested sensed data. For example, in the scope of the try_frequency function in Figure 3.5, sense<SEN_INSTR_TOTAL>(fd) returns the total number of instructions executed by all processing elements associated to the frequency domain fd in the last 50ms sensing window.

- **tryActuate**: Using the same syntax as actuate, this function updates the underlying models used to predict the next system state given the new actuation value. It does not set the actual actuation value. A tryActuationVal analogous to actuationVal is also provided.

- **senseIf**: this function has the same semantics as sense, but returns predicted sensed information for the next sensing window, given a new actuation set by tryActuate.

Note that these functions are implemented using template specialization such that every valid combination of the sensing information/actuation action parameter and resource type has its own specialization. Using this approach, sensing/action requests are mapped to their correct implementation at compile-time, thus reducing the framework’s run-time overhead. Invalid operations (e.g., setting an invalid frequency level for a power domain) can be also detected at compile-time, facilitating the debugging.

As described previously, queries to the model (done through tryActuate / senseIf) are propagated to finer-grained policies. In our current example, any other policy registered with a period smaller than 50ms is simulated given the new frequency set in Line 7. This is performed by reexecuting the same policy code registered using registerPolicy. When
executing a policy as part of the model, `actuate / sense` behave like `tryActuate / senseIf`. Finally, the developer may use the same infrastructure to model the aspects implemented within the underlying system. The `registerModel` function is provided to facilitate this process (shown in Figure 3.4). When registered using `registerModel`, the policy code will only be executed as part of the model to predict performance and power, but it will not directly actuate on the system.

Figure 3.6: System info example for an Exynos5422 HMP with ARM’s big.LITTLE. Cores are organized in clusters with the same clock frequency and the same core type. Domain #0 includes the LITTLE cores, while #1 contains the big cores. Power sensing is also done per-cluster. The `sys_info` object is also updated dynamically with objects representing the tasks currently running in the system.

### 3.2 MARS internals

In order to show the inner workings of the framework, let’s first consider the implementation of the `sense` and `actuate` functions shown in Figure 3.7. As mentioned in the previous section, these functions are defined as template functions, with the template parameters being the type of sensor or actuator (`SensingType SEN_T` and `ActuationType ACT_T`, respectively) and the type of the resource (`ResourceT`). The resource type is extracted from the type the `rsc` parameter (which is typically a pointer to one of the objects defined by `sys_info`), while the sensing/actuation types are explicitly given when `sense/actuate` are called. These
types are predefined by the framework (as a simple C++ enumeration) and each have a specialization of the SensingTypeInfo or ActuationType classes that provide information about the type. For instance, Figure 3.8 show the specializations of SensingTypeInfo for the types used in Figure 3.5. The type information includes mainly: the type of sensing values (lines 2 and 7) and how that sensing type is aggregated across the sensing window (lines 3 and 8). For instance, in the example of Figure 3.5, calling \texttt{sense<SEN\_INSTR\_TOTAL>(fd)} would return the sum of all instruction executed in the last 50\,ms, while \texttt{sense<SEN\_POWER\_W>(fd)} would return the mean power consumption in the last 50\,ms. Notice that, as mentioned in the previous section, \texttt{sense} and \texttt{actuate} map to \texttt{senseIf} and \texttt{tryActuate}, respectively, when a policy is executed as a model. This distinction is made by check the return value of Reflective\_Engine::isReflecting() (the Reflective\_Engine class will be explained next). Finally, if a policy is not executing as a model, their call are mapped to the sensing/action interface that access the real hardware. Details of this interface are provided in
Chapters 4 and 5.

![UML Diagram](image)

Figure 3.9: MARS internal components UML diagram. Details of some components already shown in Figure 3.4 are omitted.

Figure 3.9 shows an overview of MARS’s internal components for implementing reflection. The `Reflective_Engine` class is the main component responsible for implementing the `senseIf/tryActuate` functions called by user policies through `Reflective_Interface`. In order to explain the interaction between `Reflective_Engine` and the other components, let’s consider the example shown in Figure 3.10. This example expands upon Figure 3.5 and registers the `Simple_Map_Policy` to execute with `Simple_DVFS_Policy`. `Simple_Map_Policy` finds the first task-to-core mapping that meets a specified throughput goal for a task (for the sake of simplicity it assumes a single task in the system). In this case `Simple_Map_Policy` executes every 100ms, while `Simple_DVFS_Policy` executes every 50ms.

Figure 3.11 shows a sequence diagram with the interactions between the components from Figure 3.9 during one execution of the `Simple_Map_Policy`. The trigger for the execution comes from the platform-dependent implementation of `Sensing_Actuation_Interface`, which keeps track of time and calls the `windowReady` function from `Policy_Manager` every 100ms to notify that `Simple_Map_Policy`’s sensing window is ready and the policy should execute.
class Simple_Map_Policy : public Policy {
    const int PERIOD_MS = 100;

    Simple_Map_Policy() : Policy(PERIOD_MS) {} 

    void execute()
    {
        auto task = sys_info().tasks[0]
        double ips = sense<SENSTR_TOTAL>(task) / sense<SEN_TIME_TOTAL>(task);
        if (ips < GOAL)
            for (auto core : sys_info().cores) {
                tryActuate<ACT_TASK_MAP>(task, core.id);
                ips = senseSelf<SENSTR_TOTAL>(task) / senseSelf<SEN_TIME_TOTAL>(task);
                if (ips >= GOAL) {
                    actuate<ACT_TASK_MAP>(task, core.id);
                    break;
                }
            }
    }
};

class Simple_Map_Policy : public Policy_Manager {
    void setup()
    {
        registerPolicy(new Simple_Map_Policy());
        registerPolicy(new Simple_DVFS_Policy());
    }
};

Figure 3.10: Task mapping policy example. Whenever the desired IPS is lower the a specified goal, a new task mapping is explored.

Before calling Simple_Map_Policy::execute(), Policy_Manager sets up a Context object. This object stores information about the current execution context such as the ID of the current sensing window, a reference to the current policy, and whether or not a model is being executed in reflection mode.

When execute is called, Simple_Map_Policy first reads the current throughput to assess the need to optimize (Figure 3.10, line 8). In Figure 3.11, the sense calls are combined for the sake of simplicity. Assuming ips < GOAL, Reflective_Engine::tryActuate is used to try a new task mapping (this call is actually made using the Reflective_Interface and then forwarded to Reflective_Engine. This forwarding is omitted for simplicity). In the tryActuate implementation, Reflective_Engine will just update its internal data structures to register the actuation. When senseSelf is called, Reflective_Engine checks if any other policy or model should execute during the next sensing window of Simple_Map_Policy. For instance, assuming Simple_Map_Policy::execute was called at time 100ms, then we need to check whether or not another policy will execute between time 100ms–200ms. In this
Figure 3.11: UML sequence diagram of one iteration of Simple_Map_Policy

case, Simple_DVFS_Policy will execute at time 150ms, so Reflective_Engine updates the context object, setting the isReflecting flag, and calls Simple_DVFS_Policy::execute. As explained previously, with isReflecting set, policies behave as models and actuations performed will just be stored internally (actuationDB object, Figure 3.9) and won’t affect the actual system (notice the multiple interactions Simple_DVFS_Policy has with Reflective_Engine at this point were omitted for simplicity). Once Simple_DVFS_Policy execution is complete, Reflective_Engine uses the estimated actuations as input to the Perf_Power_Model to predict a set of baseline metrics used to calculate the information requested by senseIf. Notice that subsequent calls to senseIf won’t trigger the reflective executions of Simple_DVFS_Policy until Simple_Map_Policy tries to change the system again with tryActuate.

The metrics predicted by Perf_Power_Model are always made for a specific task considering it runs on a specific processing with a fixed set of estimated actuation values (e.g. the core frequency estimated after running Simple_DVFS_Policy as a model). This allows us to estimate performance/power for other components of the system by combining metrics for
all tasks mapped to those components. The performance/power models are presented in details in Chapter 6. From the perspective of the Reflective Engine, Perf_Power_Model is expected to provide the following predicted metrics for a task (Perf_Power_Model_Data in Figure 3.9):

- average number of instructions executed (IPC)
- average power consumed
- task load (or the % of core processing time the task is expected to utilize)

These metrics are sufficient for calculating all other SEN_* values. For instance, SEN_INSTR_TOTAL for a task is computed by Reflective_Engine as follows:

```plaintext
...  core = actuationDB[task][ACT_TASK_MAP];
  frequency = actuationDB[core][ACT_FREQ_MHZ];
  return window_period * baselineModel.predict(task, core, frequency).load *
          frequency * baselineModel.predict(task, core, frequency).ipc;
```

### 3.2.1 Reflection flow summary

In summary, the reflection flow works as follows:

1. Policy models predict which actuations are going to be performed in in the future. If no models or other policies were registered, we just assume the latest values read from actuationVal/tryActuationVal won’t be changed during the future sensing window.

2. the actuations predicted by step (1) are fed to Perf_Power_Model. The metrics returned are then used to predict the requested information returned by senseIf.
In Chapter 2 we mention concerns about the overhead incurred by reflective middlewares. In the case of MARS, the reflection mechanisms are tailored towards on-chip resource management to reduce the overheads. MARS’s overheads come from mainly three sources:

1. The sensing interface
2. The baseline performance/power model
3. The complexity of the policies used (since policies are re-used as part of the reflective model)

In Chapters 4 and 6 we evaluate the overhead incurred by (1) and (2), respectively. The overhead for (3) is strongly dependent of each specific use case of MARS. In Chapters 7 and 8 we measure the overheads for our specific use cases.

### 3.3 High-level policy manager

The motivation behind the high-level policy manager is to provide better resource management autonomy in response to changing system goals or execution scenarios. For instance, consider current smartphones. These devices typically operate in two scenarios: 1) the device is plugged to an external power source; 2) the device is powered by battery. In the case of (1), policies can simply focus on maximizing applications’ QoS, while for (2), QoS should be balanced with energy efficiency. A third scenario in which the battery charge is critically low is also possible. In this case, policies should focus on minimizing power consumption so the device can continue to operate. Furthermore, directly intervention from the device user may cause the scenario change as well. Creating a single policy that is able to manage all these scenarios and goals can lead to a overly complex and possibly inefficient implementation. Instead, one may develop multiple policies optimized for specific cases, and then have a
“high-level manager” dynamically the most appropriate one at run-time.

MARS was designed keeping in mind the presence of a high-level manager, though the case studies shown in this thesis we do not provide a high-level manager implementation. For an exemplar implementation of our concept of a policy, please refer to SPECTR[99]. SPECTR uses formal supervisory control theory to dynamically reconfigure lower level resource managers according to the current system goal.
Chapter 4

MARS implementation on Linux

In this Chapter we provide an overview of the environment in which MARS is deployed as well as the toolchain and infrastructure we developed to support portable development and evaluation of runtime resource management policies.

An overview of our toolchain is shown in Figure 4.1. The main component is MARS, which encapsulates the resource management policies. The middleware run as part of a user-space daemon process which monitors the system and triggers the policies as described in the previous chapters. As mentioned previously, all interactions between the policies and the system are performed exclusively through the virtual sensing/actuation interface, thus allowing the policies to be instantiated on top of different implementations of the interface.

When the middleware is deployed on top of a real platform, we use the Linux-based implementation of the virtual interface. Each interface must provide the mechanisms necessary for the implementation of the sense and actuate functions, as well as the periodic triggering of the policy functions. Figure 4.2 shows how these functions are implemented on Linux-based systems. Accessing the platform’s hardware sensors (performance monitoring units) is done by a portable kernel module. Our current implementation uses kernel tracepoints[29] to
enable performance event monitoring on a task-by-task basis, which requires sampling at the granularity of tasks’ context switch, creation, and deletion. At the user-level, the interface implementation uses `ioctl` system calls to setup the sensing windows for each registered policy function. The policy functions are then executed in the scope of a separate thread that uses blocking `ioctl` calls to synchronize with the kernel module. The kernel module automatically aggregates sensed data on a per-window basis and stores the information in shared memory, so that it can be read directly by `sense` calls with low overhead. The `actuate` functions are implemented on top of standard Linux system calls and modules. For instance, we use CPU affinity system calls to control task-to-core mappings, and the `cpufreq` to control core frequencies.
4.1 Evaluation platforms

4.1.1 ODROID-XU3

In order to evaluate MARS’s Linux implementation, we deploy MARS on the Odroid-XU3[48] development board shown in Figure 4.3. Odroid-XU3[48] features a Samsung Exynos5422 HMP, which implements a 8-core big.LITTLE-based HMP. Cores are organized into clusters of the same type (i.e. a 4-core “big” cluster and a 4-core “little” cluster). Cores in the same cluster share the the same L2 cache and the same frequency domain. Odroid-XU3 also features per-cluster power sensors and temperature sensors for the GPU and for each core in the big cluster. Figure 3.6 in Chapter 3 shows the representation of these resources within MARS. Table 4.1 summarizes the main microarchitectural parameters of the Exynos5422 HMP.

We have extensively used the Odroid-XU3 on our experiments throughout Chapters 6–8. In all experiment, the board runs Ubuntu 16.04 and Linux kernel version 3.10.
### Table 4.1: Exynos 5422 core parameters

<table>
<thead>
<tr>
<th>Parameter (Core type)</th>
<th>big (Cortex A15)</th>
<th>LITTLE (Cortex A7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue width</td>
<td>4 (OoO)</td>
<td>2 (Inorder)</td>
</tr>
<tr>
<td>L1$I$/D size (KB)</td>
<td>32/32</td>
<td>32/32</td>
</tr>
<tr>
<td>Max VF</td>
<td>2.0GHz/1.25V</td>
<td>1.4GHz/1.25V</td>
</tr>
<tr>
<td>Min VF</td>
<td>0.2GHz/0.90V</td>
<td>0.2GHz/0.90V</td>
</tr>
</tbody>
</table>

28nm technology node.

1: Per cluster shared L2 caches.

1: LQ/SQ, IQ, ROB, and reg. bank size information is not available.

### 4.1.2 GEM5

The gem5 simulator[11] is a modular platform for computer-system architecture research, encompassing system-level architecture as well as processor microarchitecture. We use gem5 to create heterogeneous cores by changing the architectural simulation parameters of the Alpha 21264 superscalar architecture modeled by gem5. This allowed us to evaluate MARS on HMP platforms with more than two core types. We configured gem5 with private L1 and L2 caches connected to the main memory through a shared bus. Table 4.2 summarizes the main microarchitectural parameters for all core types used in gem5 simulations. For obtaining power data, we have integrated the McPAT power model[67] directly with the gem5 simulation framework, which allow us to obtain power data online and reduces simulation overhead. We also extended gem5 with a sensing interface that allows McPAT power information as well as other gem5’s statistics to be accessed at run-time as hardware sensors and performance.
counters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Huge</th>
<th>Big</th>
<th>Medium</th>
<th>Little</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue width</td>
<td>8 (OoO)</td>
<td>4 (OoO)</td>
<td>2 (OoO)</td>
<td>1 (Inorder)</td>
</tr>
<tr>
<td>LQ/SQ size</td>
<td>32/32</td>
<td>16/16</td>
<td>8/8</td>
<td>8/8</td>
</tr>
<tr>
<td>IQ size</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>ROB size</td>
<td>192</td>
<td>128</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Int/float Regs</td>
<td>256</td>
<td>128</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>L1$I/D size (KB)\textsuperscript{1}</td>
<td>64/64</td>
<td>32/16</td>
<td>16/16</td>
<td>8/8</td>
</tr>
<tr>
<td>L2 size (KB)\textsuperscript{1}</td>
<td>512</td>
<td>256</td>
<td>128</td>
<td>64</td>
</tr>
</tbody>
</table>

\textsuperscript{1}22nm technology node.
Supported VF pairs:
2GHz/1V, 1GHz/0.7V, 500MHz/0.6V.
\textsuperscript{1}Per core private L2 caches.

Table 4.2: Core parameters for gem5 simulations

gem5 provides two simulation modes:

1. system call emulation mode (SE), in which applications execute without an operating system. In this mode gem5 captures system calls issued by the applications and simulates them at the functional level.

2. full system mode (FS), in which a complete operating system boots on top of gem5 and system calls are executed natively.

We used gem5 on FS mode only. SE mode provides faster simulations, however it’s less accurate and multithreaded workloads are not supported (due to the lack of an operating system scheduler). Also, we only used gem5 detailed microarchitectural model, which provide near-cycle-accurate simulation (gem5 also provide simpler, faster, core models, however those only provide function ISA-level simulations).

We currently use gem5 to more thoroughly evaluate the accuracy of the models proposed in Chapter 6 (since it provides more than two core types) and to evaluate the scalability of the task mapping approach proposed in Chapter 7. In all experiments, gem5 boots Linux kernel version 2.6.27.6.
4.2 Interface overheads

Table 4.3 shows the overhead of our sensing interface and the runtime daemon which runs MARS. We executed 8 instances of a cpu-bound microbenchmark on the Odroid XU3 platform (4 instances on the big cluster and 4 instances on the little cluster) and measured the total number of instructions executed by all cores in the whole system and the number of instructions executed by the microbenchmarks only. The microbenchmark consists of multiply-accumulate operations executed in a loop which maximizes the cpu load (Chapter 6 provides a more detailed description of the microbenchmarks). For this experiment, the execution frequencies of the big and little clusters are fixed to 1.2GHz and 1.0GHz, respectively.

<table>
<thead>
<tr>
<th></th>
<th>Linux-only</th>
<th>MARS(100ms)</th>
<th>MARS(10ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ubench instr.</td>
<td>3.9807 × 10^{12}</td>
<td>3.9807 × 10^{12}</td>
<td>3.9807 × 10^{12}</td>
</tr>
<tr>
<td>System instr.</td>
<td>3.9820 × 10^{12}</td>
<td>3.9911 × 10^{12}</td>
<td>3.9957 × 10^{12}</td>
</tr>
<tr>
<td>Overhead</td>
<td>0.03%</td>
<td>0.26%</td>
<td>0.37%</td>
</tr>
</tbody>
</table>

Table 4.3: Linux interface overheads with a 100ms sensing window and a 10ms sensing window

Table 4.3 shows the system overhead for three cases: when only the vanilla Linux kernel is used and when MARS is used with sensing windows of 100ms and 10ms. With 10ms sensing window, MARS’s overhead is very small (0.37%). There is larger increase in overhead when we compare the vanilla kernel vs MARS(100ms window), as opposed to MARS(100ms window) vs MARS(10ms window). This is due to the fact that MARS’s kernel module always samples performance counters during tasks context switches regardless of the window length.
Chapter 5

MARS offline simulator

5.1 Why an offline simulator?

As we have extensively discussed thus far, intelligent system-level resource management (e.g. task allocation and partitioning) is required in order to exploit the power-performance benefits of HMPs. Developing such intelligent management policies is a major challenge as there is a huge gap between designing a new policy and deploying it on a real platform. Many policies proposed by previous works were only evaluated using custom offline simulation infrastructures [94, 44]. New policies can be quickly created and validated using this approach, and their behavior on arbitrary, large-scale systems can be quickly simulated. However, deploying such policies on real systems brings up several new challenges not yet addressed by the original proposal, such as: sensing/tracing performance information at run-time; implementation and overhead of actuators; and the impact of other system components and orthogonal policies.

These challenges can be mitigated by designing the new policies directly on top of a real platform [112, 87] or by using full system simulators such as gem5 [11]. However, the benefits
of an offline prototype are lost. Issues that are typically straightforward to fix in a prototype running as a user-space process can be highly problematic when they occur, for instance, within a module running in kernel-space. Debugging information is more limited in kernel-space, thus resulting in a longer development process. Additionally, evaluating the new policy in terms of scalability and generality becomes much harder, since real platforms have fixed configurations and full system simulations of large-scale systems are often unfeasible due to extremely long simulation times. Other approaches, such as Sniper[17], provide a middle-ground by trading-off simulation speed and scalability for accuracy, while providing OS models and interfaces for implementing custom resource management policies. However, policies developed using such approaches still need to be re-implemented in order to be deployed on a real system.

In this scenario, we leverage MARS abstractions and implement it on top of a trace-based offline simulator(Figure 5.1). With the offline simulator and the Linux-based implementation present in Chapter 4, we can close the gap between proposing a new policy and implementing it on a real system. In the next sections, we describe the offline simulation (Section 5.2 and evaluate its accuracy (Section 5.4). Section 5.5 discuss other works that target the same

Figure 5.1: Offline simulation toolchain

40
scenarios as MARS’ offline simulator.

## 5.2 Offline simulator implementation

The offline simulator takes traces collected from applications executed on the target system (real platforms or full system simulators) and provides an execution flow that replays those traces considering an arbitrary platform configuration and actuation settings. Figure 5.2 shows the offline simulation flow. Each aspect of the simulation flow is explained in more detail below.

![Figure 5.2: Trace-based simulation flow](image)

**Trace Generation**: Traces are captured by executing applications individually as a single thread on each core configuration supported in the target system. This includes all architectural variations and all supported actuation settings. For instance, considering the ODROID platform described in Chapter 4, Section 4.1, one needs a set of traces collected on one of the *big* cores for all supported frequencies, and another set collected on one of the *LITTLE* cores. An application trace consists of a time series that includes performance and power measurements of the application. Trace collection is implemented within MARS itself by running it with a single special policy implementation which just saves the contents of the sensed data across all sensing windows. The length of the sensing window when traces are
captured defines the minimum length of the sensing windows supported later by the offline simulation.

**Offline Trace Playback:** The first step in trace simulation is defining the number of instances of each application to be replayed and their arrival time in the system. The `sys_info` object is then created based on the user specification for the platform to be simulated, and the `PolicyManager` class specialization that implements the policies is created. After initialization, the traces are played back at the granularity of the smallest sensing window according to the flowchart shown in Figure 5.2:

1. Trace information is used to obtain the maximum amount of processing time a task would use during the sensing window and its duty cycle (i.e. their busy/idle periods for tasks that are not fully cpu-bound).

2. If multiple tasks are mapped to the same core, we simulate a scheduler using LinSched [15]; LinSched is used to obtain the cpu time that would be allotted by the Linux scheduler to each task during the simulated sensing window.

3. The execution of each task is now emulated again for the allotted time and simulation statistics (e.g., executed instructions, simulated cycles, etc.) are collected to update the sensed data for each sensing window before the window handler implementing the policies is called. Once the user policies execute, control returns to the offline simulator and the steps 1-3 are repeated for the next sensing window. The trace that is used for each task on steps 1-3 is selected based on current actuation values set by the policies. When switching traces given a new actuation value, the number of simulated instructions of each task is used to find the appropriate point to continue in the new trace.

The offline simulator is also integrated with Hotspot[121] for temperature estimation, since
temperature depends on the task placement and cannot be captured by individual traces. We also integrate aging models for the case study described in Chapter 8.

5.3 Offline simulator limitations

Currently our trace-based offline simulator does not model contention on shared data and thread synchronization, so we support only multiprogrammed workloads. Additionally, we also do not model contention on the main memory bus and on the shared caches. Despite not modeling these aspects, our evaluation in Section 5.4 shows that our simulator has sufficient accuracy for multiprogrammed workloads, though we believe these limitations can be exacerbated for large-scale many core simulations. Although MARS goals with offline simulation is not towards accuracy, but to provide a first order evaluation of actuation policies, analytical cache and bus contention models such as [20] can be integrated in order to provide better accuracy. As future work, we also plan to support multithreaded workloads by extending our tracing methods with additional information to capture dependency between tasks and their synchronization points.

5.4 Validation experiments

To verify the accuracy of the offline simulations, we have implemented a DVFS policy and a task mapping policy on MARS’s framework and executed them on both the Odroid-XU3 as well as the offline simulator. The DVFS policy mimics Linux’s ondemand DVFS governor, while the task mapping policy mimics ARM’s GTS scheduler[3]. The DVFS policy executes on a sensing window of 50ms, while the task mapping policy executes on a sensing window of 200ms.
Table 5.1: Odroid XU3 results vs offline simulation results accuracy. Table entries show the total execution time (Odroid) and simulated execution time (Offline sim.) in seconds for each execution.

Table 5.1 shows the results. We perform three distinct runs using mixes of MiBench[47] benchmarks: with all tasks fixed on the big cluster; all tasks fixed on the little cluster; and with tasks being able to migrate between clusters. For all cases, a total of four tasks were created and initiated at the same time. To minimize the effect of transient background tasks on the Odroid board, we executed each case 10 times and selected the lowest observed execution time. As shown in Table 5.1, we are able to simulate the execution of policies offline with reasonable accuracy. For all cases (except dijkstra), the difference between the simulated execution time and the real execution time is below 10%. The offline simulator produces mostly smaller simulation times. This can be caused by the limitations described in Section 5.3. For example, dijkstra contains a large number of loads and conditional branches, which are sources of variation not accounted for in our simulator. The offline simulator also does not include the task creation overhead present in the real system, as well as the overhead of performing the actuation overheads (task migration and DVFS). These limitations can be addressed in future works by including additional profiling steps when performing the application tracing in order to capture the inherent overheads of the underlying system.
5.5 Related work on simulation of resource management policies

Several works have been proposed with similar goals to MARS’s offline infrastructures. Simulators like LinSched [15] and AKULA [138] are useful to easily prototype and debug OS scheduling policies before they are deployed in a real system. LinSched [15] allows one to run the entire Linux scheduling subsystem as a user-level process prior to deployment on a real system. AKULA [138] provides interfaces for the early validation of task-to-core mapping policies using a trace-based simulation infrastructure similar to MARS. However, these works do not directly support generic actuation policies for HMPs. MARS allows system-level policies to be prototyped and simulated offline under arbitrary execution and scalability scenarios. REEact[132] proposes generic interfaces for implementing portable actuation policies at the user-level, however it lacks the offline infrastructure for debugging and quick validation. Some application-level system simulators such as Sniper[17], MCSimA+[1], and CMPSched$im [79], also provide interfaces for custom resource management policies, however, the interfaces are implemented by the specific simulators only and bringing the policy to a real platform would require an additional implementation cycle.
Chapter 6

MARS’s Performance/Power Model

This chapter describes the proposed models for performance/power prediction used in MARS (Figure 6.1). This model takes as input the predicted actuations given by policies and policy models, and produces metrics that can be used to calculate predicted sensed data. In Section 6.1 we describe the mechanisms that guided us on the model development. Sections 6.2—6.4 describe the proposed models and Section 6.6 evaluate the accuracy.

Figure 6.1: MARS overview
6.1 The need for performance/power modeling

As discussed in the previous chapters, being able to properly assess the performance/power impact of resource management decisions plays a major role in efficiently managing the system, specially on HMPs in which different resource have very distinct performance/power tradeoffs. In this context, some works have the use of a static application profile[5, 19, 24, 34, 78, 113, 86] used at runtime to perform this assessment. This approach has a fundamental limitation of not being able to handle unexpected runtime workload variability or unknown applications. Other works employ techniques known as sampling [7], in which every task is periodically executed on all resource types in order to collect performance and power statistics. This approach, however, imposes a high overhead on the system.

Figure 6.2: Performance in terms of instructions per cycle (IPC) throughout time of bodytrack (a) and x264 (b) PARSEC benchmarks [10] running on out-of-order (OoO) cores with different issue widths and frequencies. Workloads simulated on gem5 as described in Chapter 4, Section 4.1. (c) show the performance when both benchmarks are executed together sharing the same core.

State-of-the art works address these issues by using models that are able to estimate or predict workload behavior by extracting performance metrics at runtime [2, 69, 98, 86, 130, 111],
however, such works suffer from limitations in the sense that they do not jointly consider all aspects that drive the performance and power of processing cores in the system. In order to illustrate these limitations, Figure 6.2 shows the performance footprint of two different applications running on two different core types. In Figure 6.2-a the cores are fully loaded with workload variability resulting from different levels of ILP throughout execution. In this case, performance is driven by the factors that affect ILP (e.g., instruction mix, cache misses, branch mispredictions, etc.). In Figure 6.2-b the application presents a very high and mostly constant ILP but with idle periods in which no computation is performed (e.g. when x264 waits for a new frame to encode). In this case, the processing time utilization or load is the performance driving factor. We can also see how both factors (ILP-driven IPC and load) differ across core types. Previous works have limitations when estimating performance for the case of Figures 6.2-a and 6.2-b as they consider either only IPC [63, 105] or only load [3, 60, 95]. Some works average out both aspects into a single metric (e.g. average task IPC considering both active and idle periods) for estimating task performance without considering the effect of scheduling policy [2, 69, 86, 130, 111], which drives the resulting individual load of each task when multiple tasks share the same core. Figure 6.2-c shows these effects.

In the case of Figure 6.2-c, one would expect that the aggregated behavior of both tasks sharing the same core would be the average of the behavior of the individual tasks. This indeed happens in the slower core, in which both tasks are get the same share of processing time. However, in the faster core, x264 low utilization causes it to be prioritized by the scheduler, while bodytrack receives the remaining processing time. This is the result of Linux’s *Completely Fair Scheduling* (CFS) policy, which prioritizes low utilization tasks. Therefore, the effects of the scheduler is an important aspect to consider when predicting performance and power.

Note that, in the scope of MARS, the scheduler can be modeled in two different ways: 1) as
a resource management policy whose main actuation knob is assigning cpu time to tasks; 2) as a resource sharing mechanisms which is indirectly influenced by other actuations knobs (e.g. task priority) and workloads characteristics. We choose the (2) approach and make the scheduler part of the baseline performance/power model. Our choice is motivated by the fact that the scheduler is the core component of the underlying OS, therefore it’s difficult to completely override its decisions using a user-level policy. From a system perspective the scheduler behaves more like a resource, with knobs that can be set (e.g. priorities ), instead of a policy.

Throughout Sections 6.2—6.5 we address the issues though generic models that unify per-task ILP factors, per-task load contribution, and scheduling policy for performance and power estimation in HMPs.

6.2 Model definitions

6.2.1 Architecture model

We consider the case of HMP systems comprising of one or more core types. Core types have the same instruction-set architecture (ISA) and are differentiated by their microarchitectural features that yield distinct performance and power footprints. We define the set of cores as $C = \{c_1, c_2, \ldots, c_n\}$, the set of core types as $Y = \{d_1, d_2, \ldots, d_o\}$, where $y : C \rightarrow Y$ gives the type $d$ of a particular core $c$. We assume cores support multiple voltage/frequency (VF) pairs and DVFS. For DVFS purposes, we defined the set of clusters as $Z = \{z_1, z_2, \ldots, z_s\}$, such that every core belongs to a cluster and all cores in the same cluster are set to the same VF pair. Core-level DVFS is defined by having clusters contain a single core, i.e., $|C| = |Z|$. The set of frequencies supported by a core of type $d$ is defined as $F_d$. 
6.2.2 Workload model

MARS’s models assume each core can run multiple tasks and that processes or applications are encapsulating threads similar to a Pthread model, so there is no formal or explicit dependency between threads. Within the Linux scheduling subsystem, processes and threads are all treated as a task entity and scheduled independently. For uniformity, in this paper the term task is used interchangeably for both single-threaded processes and for threads of the same process. We assume that tasks mapped to the same core share processing time according to the CFS scheduling policy and that all threads have the same priority.

6.2.3 Execution model

As described in Chapter 3, policies execute periodically. We name each epoch of execution of a policy its sensing window. When a policy calls the sense functions, the returned values correspond to the aggregated value in the during the lastest sensing window (for instance, when a policy that executes every 50ms calls sense<SEN_INSTR_TOTAL>, the returned value is the total number of instructions executed in the last 50ms, please refer back to Chapter 3 for details). Let $\tau(k)$ be the period that corresponds to the $k^{th}$ sensing window. Policy decisions are taken at the beginning of each sensing window such that decisions taken for the $k^{th}$ window are based on information collected during $\tau(k - 1)$ and the result of predictive models. We use $X_{c_j} = \{p_i \in P\}$ to denote the set of all tasks $p_i$ that executed on a core $c_j$ during $\tau(k - 1)$.

6.2.4 Task characterization model

Consider a task $p_i$ and a core $c_j$ such that $p_i \in X_{c_j}$; by the $k^{th}$ window, MARS collected the following performance counters an sensor information on a per-task and per-core basis:
• total amount of executed instructions ($I_{total}$), active cycles ($time_{active}$), L1 and L2 cache misses per instruction ($mrL1I$, $mrL1D$ and $mrL2$), and branch mispredictions per instruction ($mrBr$)$^1$.

• $w^{p_i c_j}$, which is the average power of $c_j$ during the execution of $p_i$ $^2$

Additionally we also define:

• $IPS^{p_i c_j}$ and $IPS^{p_i c_j}$ are the average throughput of $c_j$ in terms of instructions per cycle and instructions per second, respectively, for all scheduling quanta used by $p_i$ during $\tau(k-1)$

• $U^{c_j}$ is the core average load or core utilization, while $U^{p_i c_j}$ is the load share of $U^{c_j}$ that corresponds to $p_i$, i.e., $p_i$’s processing time utilization at core $c_j$

With this information we can obtain the average core throughput $IPS^{c_j}$ as a function of per-task IPS and load share:

$$IPS^{c_j} = \sum IPS^{p_i c_j} U^{p_i c_j}, \forall p_i \in X^{c_j}$$ (6.1)

Likewise, power consumption is directly related to per-task power and load share, as well as total core load:

$$W^{c_j} = \sum w^{p_i c_j} U^{p_i c_j} + (1 - U^{c_j}) W_{idle}^{c_j}, \forall p_i \in X^{c_j}$$ (6.2)

where $W_{idle}^{c_j}$ is the consumed power when the core is idle on a quiescent state, i.e., when there are no task ready to execute on the core (not to be confused with microarchitectural idle/stall

$^1$contemporary heterogeneous architectures such as ARM’s big.LITTLE support simultaneous sampling of these counters [46, 98]

$^2$some real platforms (e.g. ODROID) do not have power sensors per-core, or do not allow them to be sampled at context switch granularity, or both. In Section 6.4 we describe how we work around this limitation.
cycles). For the sake of simplicity, these models are defined assuming a fixed execution frequency for $c_j$ during $	au(k - 1)$. In practice, multiple instances of the model are generated for each frequency supported by $c_j$ frequency domain. The appropriated model instance is picked based on $c_j$ average frequency during $	au(k - 1)$.

### 6.2.5 Prediction model

As described in Chapter 3, Section 3.2, in order the support senseIf we need to be able compute metrics, such as the ones from Equations 6.1 and 6.2, for all possible resource allocation combination (e.g. task-to-core mapping and core frequency) Therefore, the values of $IPS^{p_{c_j}}$, $w^{p_{c_j}}$, and $U^{p_{c_j}}$ for all cores $c \neq c_j$ are required. The main goal of the prediciton model is to estimate these values based on sensed data obtained form core $c_j$.

The IPS of a task is a function of its IPC which in turn varies according to the core type and the ILP characteristics of the task, as can be observed in the examples in Figures 6.2-a and 6.2-b. In Section 6.3 we describe the prediction approach for obtaining $IPS^{p_{c_j}}$ and $w^{p_{c_j}}$ for all core types.

The task load share, however, cannot be individually predicted on a per-task basis since the observed load share of a task depends on the idle/active periods of all tasks mapped to the same core as well as the CFS scheduling policy. In order to enable the estimation of the load share for any task-to-core mapping, we introduce an additional metric named task load contribution (tlc) that describes idle/active periods of individual tasks. The load contribution $tlc_{p_{c_j}}$ of a task $p_i$ in a core $c_j$ is defined as the as maximum load $p_i$ can impose at $c_j$. For instance, in a scenario where $p_i$ is the only task on $c_j$, $tlc_{p_{c_j}}$ is the same as $U^{p_{c_j}}$. In Section 6.5 we describe how we can obtain $tlc_{p_{c_j}}$ from information of $p_i$ execution during $\tau(k - 1)$ and it can be used to estimate $U^{p_{c_j}}$ given any fixed task-to-core mapping.
6.3 IPC and power prediction

Consider a task $p_i$ and a core $c_j$ such that $p_i \in X^{c_j}$; by the $k^{th}$ sensing window, our goal is to be able to predict $IPS^{p_i,c_j}$, $w^{p_i,c_j}$ for every other core $c_l$ such that $y(c_j) \neq y(c_l)$. Section 6.5 our approach for obtaining $U^{p_i,c_j}$ given any fixed task-to-core mapping.

Prediction is necessary in order to avoid the overheads of periodically switching tasks between the different core types in order to obtain such information (also known as sampling [7]). In previous works, this type of prediction is most often performed using linear regression-based models [2, 81, 98, 119] due to their simplicity, while others employ a binning-based approach in which metrics sensed at runtime are used to classify workloads into categories whose performance/power are known for all core types[69].

In this work we use a binning-based approach. Our choice is motivated by the fact that linear regression typically requires parameters that have independent effects on the predicted value, which is not the case for performance prediction. Previous works have shown that workloads can be computational- or cache-capacity-bound [130, 49]. In this scenario, variations in computational-related parameters in a linear regression model have a different impact in the predicted metric when a workload is cache-capacity-bound (e.g., the impact of the number of mispredicted branches may have a negligible impact on throughput if the cache miss rate is high).

In order to illustrate this scenario, we executed MiBench [47] and PARSEC [10] benchmarks on our gem5-based experimental platform (refer to Chapter 4, Section 4.1 for details), and collected performance counter information. Using this information, we obtained the Pearson’s correlation coefficient for metrics calculated from performance counters and the workload throughput (IPS). Figure 6.3 plots the average correlation across all core types described in Table 4.2(Chapter 4, Section 4.1). We observe that the cache miss and branch misprediction counters have the highest correlation with IPS. The rightmost bar in Figure 6.3 shows that
Figure 6.3: Correlation between various performance metrics and IPS. The rightmost bar is the correlation for workloads with similar $mrL1D$ (less then < 10% difference).

the correlation of branch mispredictions to IPS becomes near-linear when only workloads with similar cache behavior are considered. This observation motivates us to employ a binning approach in which workloads are classified according to their memory-boundness for performance and power prediction.

### 6.3.1 Bin-based prediction

Figure 6.4 illustrates the basic idea of our predictor. For every core type, we define a predictor composed of different layers. The first layer classifies the workload according to memory-boundness. For every memory-boundness bin, the workload is classified according to compute-boundness metrics. The bins in the last classification layer store the predicted information for all other core types.

We used the correlations shown in Figure 6.3 in order to find the most relevant metrics for memory- and compute-boundness. We selected cache miss and branch misprediction counters as metrics representative of memory-boundness and compute-boundness, respectively, for the purpose of binning. As shown in Figure 6.4, we use two layers for compute-boundness: the branch misprediction is used as a first order metric, while the overall workload IPC is directly
Figure 6.4: Example of workload classification for performance and power prediction from a Big core to a Little core (Table 4.2). $mb = mrL1I + mrL1D + mrL2$ is used to bin according to memory-boundness, while $cb_1 = mrBr$ and $cb_2 = ipc$ are used for compute-boundness.

used in a second compute-boundness layer to cover the remaining IPC driving factors (e.g., instruction mix, TLB misses, etc.).

### 6.3.2 Bin-based predictor training

Similarly to regression models, our predictor also requires offline training to define the classification bin boundaries. The first part of the training process is obtaining training samples. One training sample consists of sensing information collected after running one specific workload on all core types and frequencies. In order to generate a diverse range of training samples, we used the microbenchmarking approach described in Section 6.4.

Algorithm 1 describes the process for finding the classification bin boundaries. *samples* refers to the set of all training samples described previously, while *layer_metric* stands for the metric used to classify samples in a layer. When calculating the bins for the last layer, the final predicted IPC and power is set as the average IPC and power for all samples in the last bin. In order to support DVFS, a different set of bin layers is generated for all combinations
Algorithm 1 Classification bins generation

1: for all $d_{src} \in Y$ do
2: for all $f_{src} \in F_{d_{src}}$ do
3: for all $d_{tgt} \in Y$ do
4: for all $f_{tgt} \in F_{d_{tgt}}$ do
5: \text{FIND\_BINS(first\_layer\_metric, samples, $d_{src}, f_{src}, d_{tgt}, f_{tgt}$)}
6: end for
7: end for
8: end for
9: end for
10: function FIND\_BINS(layer\_metric, samples, $d_{src}, f_{src}, d_{tgt}, f_{tgt}$)
11: \text{bins} ← \text{equally sized bins such that every bin has at least one sample}
12: for all \text{bin} \in \text{bins} do
13: \text{bin\_samples} ← \{s \in \text{samples}|layer\_metric(s, d_{src}) \leq \text{bin\_max}\}
14: if layer\_metric is the last layer then
15: $IPC_{d_{tgt}, f_{tgt}} ← \text{average\_ipc}(d_{tgt}, f_{tgt}, \text{bin\_samples})$
16: $P_{d_{tgt}, f_{tgt}} ← \text{average\_power}(d_{tgt}, f_{tgt}, \text{bin\_samples})$
17: else
18: \text{FIND\_BINS(next\_layer\_metric, bin\_samples, $d_{src}, f_{src}, d_{tgt}, f_{tgt}$)}
19: end if
20: end for
21: end function

of core types and frequencies.

6.4 Predictor training

As mentioned previously, training the predictor requires training sample. One training sample consists of sensing information collected after running one specific workload on all core types and frequencies. Training sample collection is implemented within MARS itself by running it with a single special policy implementation which just saves the contents of the sensed data across all sensing windows (the same policy used to collect the traces described in Chapter 5) as shown in Figure 6.5.

The main challenge for predictor training is therefore finding a reasonable training set. A reasonable training set should cover a wide range of workload behaviors in order to be robust to unknown workloads. In order to provide a systematic training process and to obtain samples which are independent from specific applications used for evaluation, we developed a set of microbenchmarks for training. A microbenchmark is defined as a simple function which
exhibits a specific computational and/or memory behavior (e.g., high/low instruction-level parallelism, high/low cache miss rates, etc.). The microbenchmark are described in the next section.

### 6.4.1 Microbenchmarks

Performance and power are driven mainly thread ILP and load contribution, therefore, applications that exercise both these factor must be used in order properly assess the applicability of performance and power estimation models. Similarly to related work [86, 78, 69], we use the PARSEC benchmarks [10]. Figure 6.6-a shows how the IPC and load contribution of the PARSEC benchmarks varies across the different core types in our platform. PARSEC have good ILP diversity. However, its applications are CPU-bound and has a mostly constant high load, which prevents us from properly evaluating the impact of distinct load

---

The x264-5fps benchmark is a modified version of the original x264 that limits the input data rate in order to illustrate the impact of load.
contribution patterns. For this reason, we have developed a set of synthetic microbenchmarks with attributes that reflect interactive/IO dependent applications. Our microbenchmarks allow us to create threads with controlled active/idle periods and exercise a wide range of load contribution patterns.

Our microbenchmarks have the general structure shown below:

```
repeat
  IDLE_PHASE(idle_time)
  COMPUTATION_PHASE(iterations)
until runtime <= 0
```

The `idle phase` is an inactive period in which the thread blocks. The duration of this period is given as absolute time to simulate IO/blocking periods that do not change according to the speed of the core the thread is running in. The `computation phase` is the active period that repeats some defined computation during the specified amount of iterations. The total duration of this active period scales according to core type speed, therefore increasing the load contribution of the thread on slower cores. In order to exercise different ILP patterns.
within the active period of the microbenchmarks, we combine different operations resulting in the distinct phases shown in Figure 6.6. The operations are described below:

- A sequence of independent multiplications and additions (MACs), yielding high ILP;
- Nested recursive function calls (RF), yielding low ILP;
- A large switch-case statement (Big switch) that randomly returns one out of $2^{11}$ values, yielding low ILP due to high instruction cache miss rates;
- Nested if-then-else statements, yielding low to moderate ILP.

Additionally, some of these operations can be performed over either integer (Int) or floating point (FP) data. The input data can be either from a small local array (yielding good data locality (GDL) and low data cache miss rates) or randomly fetched from a large array (yielding bad data locality (BDL) and high data cache miss rates).

Figure 6.6-c shows the microbenchmarks we have obtained by combining the computational phases with different inactive periods. The microbenchmarks described as high ILP, low ILP, and very low ILP, have their computational phase composed by Int.MACs/GDL, Int.RF/GDL, and Int.MACs/BDL, respectively.

The full set computational phase options is used to systematically generate training cases for calibrating our performance and power prediction models described in Section 6.3. A training sample is obtained by collecting aggregated sensing information for the sequential execution of a combination of computational phases characterized by Figure 6.6-b. Currently we use our microbenchmarks to generate a total of 1536 unique workload combinations. Each workload combination is executed on all core types for all possible voltage/frequency pair and the result is used as input for the classification bins generation.
6.5 Task load prediction under resource sharing

As discussed in Section 6.2, the load share of a task in a given core not only depends on the task characteristics, but also as a function of the task’s idle/active periods, the idle/active periods of the other task in the same core, and the scheduling policy. For this reason we have defined the task load contribution (tlc) as a individual task metric that describes the maximum load a task can impose on a core given that it has full-time access to the core. The task’s tlc can then be used to estimate the actual task load share for any possible combination of tasks running on the same core.

6.5.1 Load contribution estimation

Estimating the task’s load contribution is challenging as there is not a direct relationship between the measured load shares and the tlc. In this work we assume tasks have the same priority and are scheduled under the CFS policy. Under these assumptions, consider a core $c_j$ and all tasks $p_i \in X^{c_j}$, i.e., all tasks mapped to $c_j$ by the $k^{th}$ sensing window. The task load contribution $tlc_{c_j}^{p_i}$ can be estimated by Algorithm 2.

```
Algorithm 2 Task load contribution estimation
1: if $U_{c_j}^{p_i} < 1$ or $|X^{c_j}| = 1$ then
2:   for all $p_i \in X^{c_j}$ do
3:       $tlc_{c_j}^{p_i} \leftarrow U_{p_i}^{c_j}$
4:   end for
5: else
6:   $eqshare \leftarrow \frac{1}{|X^{c_j}|}$
7:   for all $p_i \in X^{c_j}$ such that $U_{p_i}^{c_j} < eqshare$ do
8:       $tlc_{c_j}^{p_i} \leftarrow U_{p_i}^{c_j}$
9:   end for
10:  for all $p_i \in X^{c_j}$ such that $U_{p_i}^{c_j} \geq eqshare$ do
11:     $tlc_{c_j}^{p_i} \leftarrow \max \left( U_{p_i}^{c_j}, \frac{csw_{p_i}^{p_i}}{csw_{p_i}^{voluntary} + csw_{p_i}^{involuntary}} \right)$
12: end for
13: end if
```

Algorithm 2 lines 1-4 covers the base definition of load contribution as described in Section 6.2. If the core is not fully loaded or there is only one task mapped to the core, then all tasks
load share $U^{p_i,c_j}$ are already the maximum load they can impose on the core. Otherwise, we first establish that all tasks have the right to an equal share of processing time (according to the CFS policy — line 6). All tasks that do not fully utilize their rightful share of processing time have already maximized the load they can impose to the core (line 8). For all other tasks, we estimate $tlc_{p_i}^{c_j}$ from a ratio between the tasks’ context switch counters $csw_{p_i}^{ involuntary}$ and $csw_{p_i}^{ voluntary}$ (line 11) provided by the Linux kernel. $csw_{p_i}^{ involuntary}$ counts how often the scheduler evicted $p_i$ from its core so another task could execute. $csw_{p_i}^{ voluntary}$ counts how often $p_i$ voluntarily yielded the core to the scheduler (e.g. when blocking or waiting for some event due to a sleep system call). The rationale behind this approach is that a task is more frequently evicted from its core as $tlc_{p_i}^{c_j}$ tends to 1.

Algorithm 2 estimates $tlc_{p_i}^{c_j}$ for all $p_i \in X^{c_j}$, however, as shown in Figures 6.6-a and 6.6-c, $tlc_{p_i}^{c_j}$ varies across core types since the active phase of a task demands more computational time in slower cores. Given that $p_i$'s $IPS_{p_i}^{c_j}$ is known for every other core $c_i$ such that $y(c_j) \neq y(c_i)$, we can predict $tlc_{p_i}^{c_i}$ by scaling $tlc_{p_i}^{c_j}$ as follows:

$$tlc_{p_i}^{c_i} = \frac{tlc_{p_i}^{c_j} * IPS_{p_i}^{c_j} / IPS_{p_i}^{c_j}}{(1 - tlc_{p_i}^{c_j}) + (tlc_{p_i}^{c_j} * IPS_{p_i}^{c_j} / IPS_{p_i}^{c_j})}$$

(6.3)

### 6.5.2 Load prediction under CFS

We estimate the task load share through a high-level model of Linux CFS scheduler’s behavior. The rationale behind CFS is to assign an equal amount of processing time to all tasks mapped to a core. However, a task may not use all of its allotted time. This unused residual time is redistributed to the remaining tasks that demand more than the originally allotted processing time share. Through empirical observations, we verified that the residual time is redistributed proportionally to the tasks’ load contribution. Figure 6.7 illustrates this behavior by showing how the load share varies across different combinations of the microbenchmarks.
from Figure 6.6-c in Section 6.4.1. Given a new set of tasks $X'_{c_j}$ to be mapped to core $c_j$ during the $k^{th}$ as, Algorithm 3 models this CSF behavior to obtain the load share for each $p_i \in X'_{c_j}$.

Figure 6.7: Resulting load shares when tasks with different load contribution are scheduled under CFS. Low, medium, and high contribution tasks correspond to the Low load/High ILP, Medium load/High ILP, and High load/High ILP microbenchmarks. Measured in a Huge core at 2GHz.

© 2015 IEEE

Algorithm 3 Load share prediction

1: $U_{c_j} \leftarrow 0$
2: for all $p_i \in X'_{c_j}$ do
3: $U_{p_i, c_j} \leftarrow tlc_{p_i}$
4: $U_{c_j} \leftarrow U_{c_j} + U_{p_i, c_j}$
5: end for
6: if $U_{c_j} > 1$ then
7: $eqshare \leftarrow \frac{1}{|X'_{c_j}|}$
8: residual $\leftarrow 0$
9: tlcSum $\leftarrow 0$
10: for all $p_i \in X'_{c_j}$ do
11: if $tlc_{p_i} < eqshare$ then
12: residual $\leftarrow$ residual $+$ (eqshare $- tlc_{p_i}$)
13: else
14: tlcSum $\leftarrow$ tlcSum $+$ tlc_{p_i}
15: end if
16: end for
17: for all $p_i \in X'_{c_j}$ such that $tlc_{p_i} \geq eqshare$ do
18: $U_{p_i, c_j} \leftarrow eqshare + \frac{tlc_{p_i}}{tlcSum} \times residual$
19: end for
20: $U_{c_j} \leftarrow \sum U_{p_i, c_j}$
21: end if

$\triangleright$ Recalculate the core total load
6.6 Accuracy evaluation

Bin-based prediction error: Figure 6.8 shows the average error in predictions across all core types. The runtime prediction of IPS and power incurs an average error ranging from 1% to 6% across all core types in Table 4.2 (Chapter 4, Section 4.1). We consider this error to be small when compared to previous works that propose performance/power prediction schemes [2, 69, 81]. Comparing to previous works, Annamalai et al [2] reports errors ranging from about 10% to about 16% when using a linear regression model to predict IPC/Watt across two core types. Pricopi et al [98] employs a similar predictor for ARM’s big.LITTLE architecture. The authors report errors of 13.4% and 16.7% for big→LITTLE and LITTLE→big performance predictions, respectively. Liu et al [69] uses an approach similar to ours on an architecture with four core types, with a reported error of 14% and 7% for IPS and power, respectively. Liu et al bins according to PARSEC benchmarks, i.e., a task whose observed performance matches one of the profiled PARSEC benchmarks on one core type is assumed to match the performance of that benchmark on all core types.

![Figure 6.8: Average error and standard deviation of IPS and power prediction across multiple core type combinations.](image)

Load contribution estimation errors. We have executed random combinations of microbenchmarks described in Figure 6.6-c in Section 6.4.1, and measured the average error
of the load contribution estimation. As shown in Figure 6.9, we achieve an average error of about 8% across combinations of 2 to 16 tasks. The error tends to increase as more tasks are packed to a single core since less events occur in the csw counters for each individual task. Figure 6.9 also shows that the difference between predicted and the real tlc increases as the real tlc tends to 0.5. This is due to the fact that as tlc approaches 1, the csw counters yield more precise estimations, while, as the tlc approaches 0, the estimation is more likely to be done as shown in Algorithm 2, line 8. Please note that Figure 6.9 does not include the task mixes whose tlc is estimated by Algorithm 2 lines 1-4, since, according to the definition of tlc, for this cases the estimation is always correct. Therefore, in practice, the average estimation error is expected to be smaller.

![Figure 6.9: Average error of task load contribution estimation on fully loaded cores by using context switch counters](image)

Compared to previous work, MARS is the first to estimate per task load contribution. A form of per-task load tracking has been included in a recent version of the Linux kernel [102] is used by ARM’s GTS[3]. This approach keeps a moving average of the load share of each task; and the total core load considered for load balance purposes is the sum of the accumulated load of all tasks in the core. This approach allows ARM’s GTS[3] to compare the relative load of different cores, but does not yield an absolute load value. Our approach allows the estimation of the real core load that results from a task mapping.

Figure 6.10 shows the average load share prediction error for the same cases shown in
6.7 Discussion and limitations

The performance/power model in MARS is responsible for capturing the effects of the hardware architecture and resource sharing on the workloads, given a fixed set of resource allocation and actuation settings. For the specific case of the models proposed in this Chapter, the main resource type considered is single-ISA processing cores and the effects the scheduling algorithm has on tasks sharing the same core. This model has proved sufficient since these are the main factors driving performance and power on our case studies present in Chapters 7 and 8, however we envision the extension of MARS’s model in multiple directions in order to increase it’s accuracy and support more general scenarios:

**Contention on the shared memory subsystem** Tasks can interfere with each order when they share cache lines and memory controllers. This is specially true for memory-intensive applications, however our current classification-based prediction approach does not take this into account as it only considers tasks in isolation. A sharing model akin to the CFS model can be integrated to address this limitation. For instance, analytical
cache and bus contention models such as [20] could be used.

**More scheduling policies** Currently we consider user-level Linux tasks, which are schedule using the CFS policy and typically have the same priority. Our scheduling models could be generalized to also consider multiple priorities and the interplay between CFS and the other scheduling policies in Linux (e.g. the round-robin and FIFO realtime policies used to schedule kernel-level tasks).

**Support for general memory models** We currently use cache-coherent shared memory architectures in our evaluations, and this is reflected in our performance/power model assumptions. However, for distributed architectures and manycore on-chip systems that use Network-on-Chips and/or multiple memory controllers, the distance between the location of the data and where the processing is happening plays a major role.

**Beyond single-ISA heterogeneity** Many systems today combine CPUs with GPUs, FPGAs, DSPs, and other application-specific accelerators. Processing-in-Memory (PIM) based accelerators have also been extensively explored as promising solutions for memory-intensive applications. In this sense, extending MARS's models to support beyond single-ISA heterogeneity would greatly increase its potential benefits.
Chapter 7

Energy-efficient Task Mapping for Heterogeneous Multiprocessors

This chapter presents *SPARTA: a runtime task allocation approach for energy efficient HMPs*, as a case study of MARS. SPARTA’s runtime adaptivity provides support for workloads that consist of numerous tasks with diverse behavior that may enter and exit the system at any time. SPARTA achieves energy efficient allocations by leveraging MARS’s sensing infrastructure and models presented in Chapter 6 to identify opportunities to maintain a task’s performance while reducing its energy consumption. SPARTA is evaluated on top of both platforms described in Chapters 4 and 5: the ODROID-XU3 and offline simulation. In the Odroid platform, we compare SPARTA with GTS (the current Linux solution for HMP task mapping), while in the simulated platform we also compare with the MTS[69, 70] heuristic and a optimal energy-efficiency heuristic. As shown in the results in Section 7.3, SPARTA delivers the same or better performance then GTS and MTS (which are performance-driven heuristics) with better energy efficiency. Across all workload mixes presented in [30], SPARTA achieves average energy reductions of 23% when compared to MTS, and 16% compared to Linux’s GTS running on the Odroid platform.
Section 7.1 describes the main ideas behind SPARTA while Section 7.2 describes their implementation using MARS. The experimental evaluation is presented in Section 7.3, while Section 7.4 provides related work on task mapping approaches for HMPs for further reading.

7.1 SPARTA rationale

SPARTA stands for Sense-Predict-Allocate Runtime Task Allocator consists of three phases as shown in Figure 7.1: sensing, classification/prediction, and allocation. The bottom part of
Figure 7.1 gives an overview of the relationship between this three phases. An *Epoch* is the time period between classify/predict and allocate phases, while the sensing phase is executed at the same rate as the Linux scheduler, so each epoch covers multiple Linux scheduling periods. The upper part of Figure 7.1 provides and example of SPARTA’s runtime task allocation. In this example, we consider three distinct tasks executing on a 4-core HMP containing one core for each heterogeneous type described by Table 4.2 (Chapter 4):

1⃝ During the sensing phase, hardware performance counters and power sensors are periodically sampled on each core to monitor the characteristics of the executing workload. The following counters are sampled at the same rate as the Linux scheduler (typically 10-20ms) and individually summed up for each task at the beginning of each epoch: total amount of executed instructions ($I_{total}$), active cycles ($time_{active}$), L1 and L2 cache misses per instruction ($mrL1I$, $mrL1D$ and $mrL2$), and branch mispredictions per instruction ($mrBr$).

With this information, we can define the *throughput* of a task $p$ that executed in a core $c$ in terms of *instructions per second* as $p.ips = \frac{c.I_{total}}{c.time_{active}} \times c.freq$. Note that $p.ips$ denotes the average throughput of task $p$ across all scheduling periods that $p$ executed. The average *effective throughput* of $p$ across the entire epoch is defined as $p.ips \times p.load$, where $p.load = \frac{c.time_{active} \times c.freq}{EpochLength}$ is the share of processing time used by task $p$ during an epoch of duration $EpochLength$.

2⃝ At the beginning of a new epoch, each task’s performance and power is predicted for all core types using the sensed information. This phase allows SPARTA to obtain task metrics (e.g. $p.ips$, $p.load$) for cores in which the task has not executed on.

3⃝ The final step is to determine the global task allocation for the next epoch. The first order goal is to meet the *target throughput* of the task and then maximize energy efficiency\(^1\).

A task’s target throughput is the maximum achievable throughput of the task on the fastest core.

\(^1\)in terms of throughput(IPS) per Watt, which can also be thought of as instructions-per-Joule
core type. We consider a task’s target throughput as being met if the task observes negligible effective throughput improvements when mapped to a faster core (i.e. Little → Big). When multiple cores can achieve target throughput, the task is allocated to the most energy efficient one. For instance, in Figure 7.1, $t_2$ effective IPS is saturated in the Big and Huge types, meaning that these cores achieve $t_2$ target throughput. This saturation is the typical case for interactive task with computation—IO/sleep cycles. By providing a faster core to this task, it’s IPS increases only during the computation cycles, which in turn decreases the core load, thus limiting the task’s effective throughput. For tasks that we do not observe IPS saturation (e.g. $t_1$ in Figure 7.1), we assume that in the best case its target throughput can only be achieve by the fastest core (Huge). This is the typical case for non-interactive compute-intensive tasks with only computation cycles.

Section 7.2 how these ideas are implemented within MARS’s reflective model.

7.2 Implementation on MARS

SPARTA implemented on top of MARS is shown in Figure 7.2. This implementation is akin to the task mapping example shown in Figure 3.3 in Chapter 3. SPARTA task allocation

Figure 7.2: SPARTA implementation in MARS

SPARTA implemented on top of MARS is shown in Figure 7.2. This implementation is akin to the task mapping example shown in Figure 3.3 in Chapter 3. SPARTA task allocation
heuristic is implemented as a resource management policy within MARS. MARS’s sensing windows provide epoch-based sensing and policy execution describe in Section 7.1. When exploring new task allocations, the heuristic issues \texttt{tryActuate} call to inform the reflective system model about the changes. The task’s IPS and system power are then obtained using \texttt{senseIf} function in order to decide if the new mapping is accepted or not.

The reflective system model includes a model for the DVFS governor implemented in the underlying operating system. The DVFS governor model estimates a new frequency for the all core given a change in the task mapping. The new mapping, estimated frequencies and current sensed data is used as input for the baseline performance/power model (Chapter 6) which estimates the final IPS and power. The next subsections describe the DVFS model used and the task allocation heuristic.

7.2.1 DVFS model

We target Linux-based systems, so we implement the DVFS model based on on Linux’s \texttt{ondemand} governor. The ondemand governor defines a set of performance levels ordered from the lowest to the highest, with each level corresponding to a \(<\text{frequency}, \text{voltage}\>\) pair. Levels are changes based on core load thresholds: when the core load is above the \textit{up threshold}, performance is set to maximum level; when the load falls below the \textit{down threshold}, the performance level is decreased to the next lower level. In Linux, the \texttt{cpufreq} module provides an interface to setup thresholds how often the core load is checked. In our implementation we run the DVFS with a sensing window of 50\textit{ms} and use 90\% and 40\% as the up and down thresholds, respectively.
7.2.2 Task allocation algorithm

Given a set of executing tasks and available cores, an allocation is an assignment of all tasks to a core. As described earlier, the SPARTA allocator’s goal is to minimize the power consumption of a workload while maintaining its target throughput. This optimization problem is NP-hard[40], and is therefore infeasible to complete at runtime every epoch, even for small configurations. Therefore, we developed a heuristic solution based on list scheduling[25]. The SPARTA Allocator presented in Algorithm 4 uses task target throughput (TT) properties to significantly reduce the complexity of finding an allocation. As described previously, a core is considered to achieve target throughput for a task if the predicted effective throughput on the core matches the maximum observable effective throughput for that task across all cores. Using this notion, the rationale behind the SPARTA allocator is to map each task to the most power-efficient core (in terms of IPS/Watt) that achieves the target throughput.

The first steps of Algorithm 4 are to initialize the per-task TT (lines 5–10). $p.\text{ips}[c]$ is the task effective IPS when mapped to core $c$. In this initial phase, we need to find the maximum effective IPS to set it as the TT, so we temporally unmap all tasks and then map them one-by-one to each core using tryActuate and then senseIf to obtain $p.\text{ips}[c]$. Next, we compute the task target throughput $p.TT$ and the number of core types that meet it (line 12). Note that we correct $p.TT$ by a constant $0 < \alpha < 1$. We currently use $\alpha = 0.95$ to account for prediction errors.

Based on the throughput constraint, tasks are then separated into three lists — tasks with achievable target throughput on all core types (full_TT_list), tasks with achievable target throughput on a subset of core types (partial_TT_list), and tasks with unachievable target throughput (no_TT_list) (lines 13–21). All task lists are ordered for allocation by descending maximum achievable IPS/Watt for TT-satisfiable tasks, and by maximum achievable IPS for TT-unsatisfiable tasks (lines 24–26). The lists are allocated in-order to assign the tasks
Algorithm 4 SPARTA Heuristic

function SPARTA(tasks $P$, cores $C$, core types $Y$)
▷ find target throughput and constraints
for all $p \in P$ do
  $p.TT \leftarrow 0$
  for all $c \in C$ do
    if $p.TT < p.ips[c]$ then
      $p.TT \leftarrow p.ips[c]$
    end if
  end for
  $p.TT \leftarrow p.TT \times \alpha$
▷ Obtain target throughput constraints
  $p.constraint \leftarrow \{d | \{p.TT \leq p.ips[c], \forall c \in C\}$
  if $|p.constraint| = |C|$ then
    full_TT_list.add($t$)
  else
    if $|p.constraint| > 1$ then
      partial_TT_list.add($p$)
    else
      no_TT_list.add($p$)
    end if
  end if
end for
▷ Sort respective task lists according to constraint
sort_tasks_constrained/IPS/Watt(partial_TT_list)
sort_tasks_IPS(no_TT_list)
sort_tasks_IPS/Watt(full_TT_list)
▷ Allocate all tasks
Allocate(partial_TT_list)
Allocate(no_TT_list)
Allocate(full_TT_list)
end function

function ALLOCATE(task_list)
  $p.mapping \leftarrow \emptyset$
for all $p \in task_list$ do
  ▷ Maximize IPS/Watt given TT constraint
    $p.mapping \leftarrow c | c \in p.constraint,$
    $p.ips[c] \geq p.TT,$
    $\frac{p.ips[c]}{p.power[c]}$ is maximized
  ▷ Maximize IPS if cannot meet constraint
  if $p.mapping = \emptyset$ then
    $p.mapping \leftarrow c | c \in C,$
    $p.ips[c]$ is maximized
  end if
end for
end function
constrained to fewest potential core types first, i.e. first the partial TT list, followed by the no TT list and full TT list (lines 28–30).

The allocate function (line 33) progressively maps all of the tasks in task list. For the tasks with achievable target throughput, each task is allocated in order to a core with maximum IPS/Watt that achieves target throughput (line 37). Tasks with unachievable target throughput are allocated to the core that maximizes throughput (line 40). When executing this function tryActuate is used to propagate the mapping for the current task, without changing the mapping of previously already mapped tasks. This way, using senseIf to obtain $p.ips[c]$ returns the current effective IPS considering all tasks already mapped and the behavior of the DVFS governor.

**SPARTA’s complexity:** Algorithm 4 has complexity of $|P|^2$ for the initial sorting phase and $|P| \times |C|$ for the allocation phase. If we assume an average case performance of $O(|P| \times log(|P|))$ for sorting, then the SPARTA’s runtime is bound by $O(|P| \times |C|)$.

### 7.3 Experimental evaluation

Our evaluation is done on two distinct platforms: 1) the ARM big.LITTLE based Exynos 5422 SoC deployed on the ODROID-XU platform[48], which contains a SoC with four big and four little cores (Chapter 4, Table 4.1); and 2) a offline simulation (Chapter 5) larger scale HMPs using traces collected from gem5 containing the four core types described in Chapter 4, Table 4.2.
Table 7.1: Benchmark Mix composition

7.3.1 Workloads

Our experimental workloads are constructed using two different approaches. Workloads are made up of benchmarks from MiBench and PARSEC benchmark suites, as well as synthetic microbenchmarks (described in Chapter 6, Section 6.4.1).

The first set of workloads (Mixes 1, 5, 10, and 15 in Table 7.1) are diverse: the benchmark mixes comprise a combination of the most compute- and memory-bound benchmarks from PARSEC, as well as synthetic microbenchmarks that exercise different levels of compute and memory boundness. For each suite, we analyzed each benchmark’s performance gain in terms of IPS when increasing core size, and selected the subset that benefited most for the respective resource types; Mix 1 consists of these PARSEC benchmarks. Mixes 5, 10, and 15 consist of synthetic workloads using microbenchmarks that can be either compute- or memory-bound (CB or MB) and impose a high, medium, or low average cpu utilization (HU, MU, or LU).

The second set of workloads (all other Mixes in Table 7.1) represent realistic homogeneous use-cases. Mixes 7, 14, and 11 consist of network, automotive/industrial, and consumer benchmarks from MiBench respectively. Mixes 12 and 3 each consist of a single benchmark from PARSEC representative of computer vision and data mining respectively. Mixes 4, 6, and 2 are composed based on the analysis performed by [37], which characterizes mobile use cases in terms of core utilization. We use the our microbenchmarks to define four typical mobile workloads based on the resulting utilization of big and little cores: Mix 4 is a typical load; Mix 6 is a typical load with a heavy task; Mix 2 is a heavy load. Mixes 8, 9, and 13
consist of a combination of x264 with different frame-rate inputs in order to invoke further target throughput variation.

For all experiments, the total number of threads is half the number of cores in the configuration, which we believe is a valid use-case for heterogeneous many-cores.

### 7.3.2 Simulation results

We first evaluate SPARTA using the offline simulator. We compare SPARTA against the following allocators: MTS, GTS, and an algorithm that maximizes energy efficiency. MTS is an implementation of the allocation algorithm with no power budget that we consider the most comparable state-of-the-art solution. GTS is ARM’s big.LITTLE allocation technique that we adapted to work for more than two core types. Since GTS the only solution adopted in mainstream Linux, we will use it as a baseline. The goal of both MTS and GTS is to maximize throughput. Maximum Energy Efficiency (Max EE) is a optimal brute-force allocation for maximizing IPS/Watt. For the following experiments we use an oracle predictor – predictor with no error – in order to evaluate only the allocation techniques. Additionally, we use our SPARTA prediction scheme for all allocators, which means that although MTS is not DVFS-aware by design, in our experiments it benefits from our improved DVFS-aware predictor. GTS is unaffected as it is reactive and therefore does not use any prediction.

Figure 7.3 shows the IPS and IPS/Watt of all allocators for all benchmark mixes on two 8-core configurations normalized to the GTS baseline. In **Platform A**, the eight cores are made up of two Huge, Big, Medium, and Little cores each. **Platform B** is made up of four Big and four Little cores similarly to the Odroid platform. For benchmark Mixes 8-15 on Platform A, SPARTA is able to match the performance of the state-of-the-art technique while increasing energy efficiency in most cases. We observed an average of 26% increase in energy efficiency over GTS for Mixes 8-15 while increasing throughput by 57%. Similarly,
we observed an average 19% increase in energy efficiency over MTS with 8% performance improvement for these mixes. This can be attributed to the fact that while both MTS and GTS prioritize maximizing throughput, they do not consider the utilization of each thread on different core types. By accounting for core utilization, SPARTA can find the most energy efficient thread-core mapping for threads whose performance is saturated. Mixes 8-15 mostly contain at least one benchmark that does not completely utilize all core types. The results for these workload types show that SPARTA is able to efficiently identify opportunities to maximize energy efficiency without degradation of the target throughput.

Mixes 1-7 in Figure 7.3 represent the alternative scenario: when a tradeoff in performance is required to save energy. This is illustrated clearly by the Max EE allocator’s poor throughput and high energy efficiency for these mixes. The manner in which SPARTA prioritizes the most energy efficient threads during allocation allows us to also beneficially exploit this scenario.
As before, by recognizing when multiple core types can achieve a task’s target throughput, SPARTA is able to improve energy efficiency for cases in which performance is saturated. For benchmark Mixes 1-7, SPARTA observed 27.5% and 6.5% average improvement in throughput compared to GTS and MTS respectively. SPARTA also increased energy efficiency by 33% and 27% on average over GTS and MTS respectively. These benchmark Mixes mostly consist of high performance tasks whose performance is unbound by resource allocation. For tradeoff scenarios in which the target throughput is not commonly met, SPARTA is able to achieve significant energy savings while maintaining performance.

Platform B follows a similar trend to Platform A, but with less significant deviation between SPARTA, MTS, and GTS. This is due to the platform only having two different core types. SPARTA is still able to improve performance by 5.5% and energy efficiency by 11% over GTS on average over all mixes. Compared to MTS, 5% increased energy efficiency comes at the cost of negligible performance degradation (2%).

Figure 7.4: Scalability: average IPS and IPS/Watt for different core configurations normalized to GTS.

Figure 7.4 compares scalability, showing the average performance and energy efficiency of the heuristic allocators for configurations with up to 256 cores and 128 tasks. The values are
averaged over all of the Mixes mentioned previously for each configuration. Each configuration also had two permutations: one with an equal distribution of the four core types (1:1:1:1 ratio), and another with a ratio of 1:3:3:1 cores of types Huge, Big, Medium, Little respectively. For configurations of 32 cores or more, SPARTA and MTS both improved throughput over GTS between 5-10%. SPARTA was able to save up to 17% and 10% energy over GTS and MTS respectively.

Figure 7.4 also includes evaluation of allocators using the SPARTA predictor. The predictor had minimal impact on the allocators. In some cases, predictor error resulted in improved allocations. The predictor did not affect the throughput or energy efficiency achieved by SPARTA or MTS by more than +/-5% relative to the oracle versions. The SPARTA and MTS relative relationship will always hold because they are equally a function of their predictor’s quality. A more accurate predictor could be incorporated into SPARTA in the future in order to yield increased benefit over GTS.

### 7.3.3 ODROID platform results

Figure 7.5 shows the IPS and IPS/Watt of SPARTA for a subset of benchmark mixes normalized to the GTS baseline. Overall, SPARTA matches the throughput of GTS while improving energy efficiency by 16%. Similarly to Platform B in Figure 7.3, overall most mixes differ minimally or not at all between SPARTA and GTS. In some cases, like Mixes 2 and 10, SPARTA or GTS make slight tradeoffs between performance and energy efficiency. For Mix 13, SPARTA sacrifices 13% throughput for a 40% energy efficiency gain. In the few outlier cases in which the difference is not negligible and no tradeoff is made, SPARTA is able to identify opportunities to improve energy efficiency at no cost (Mixes 8 and 9) or improved performance (Mix 11). Based on these results, we can conclude that both the overhead of the SPARTA runtime as well as the predictor error is manageable on real systems.
Figure 7.5: IPS and IPS/Watt of allocators for different benchmark mixes normalized to GTS on the Odroid platform.

**SPARTA’s overheads and scalability:** We measure the run time of each SPARTA phase on the Odroid platform. The total latency of each phase is relatively low compared to the 200ms epoch length. The total time spent sensing during an epoch on all cores averages to 28µs. Across the 8-core benchmark runs described previously, the measured latency of the classification/prediction and allocation phases is 124µs and 190µs respectively (measured on big cores). Figure 7.6 plots how the latencies scale for larger systems and includes the thread migration overhead (assuming 25% of threads migrate every epoch and that each migration takes about 20µs [1GHz][135] ). The total overhead imposed by SPARTA is negligible for up to 16 cores. The classify and predict phase is the most cumbersome of the SPARTA overhead sources, but it still bellow 5% of the epoch length for up to 64 core configurations. For platforms with 100s of cores the overhead is high, but it can be mitigated by the fact that the execution of prediction and allocation phase occurs in parallel with the execution of every other task in the system. Nevertheless, this is a limitation of our current implementation of SPARTA that we plan to address in future work.
7.3.4 Evaluation summary

Overall, the experiments have illustrated SPARTA’s applicability across both contemporary HMPs as well as potential future HMPs with increased resources and resource types. The SPARTA predictor can predict behavior on a per-task basis for diverse workloads consisting of unknown tasks in conjunction with multitasking and DVFS. The allocator successfully identifies opportunities to reduce workload energy consumption by considering target throughput.

7.4 Summary and related work

SPARTA is a throughput-aware runtime task allocation approach for HMPs that leverages MARS's framework to achieve energy efficiency. Our experimental results showed energy reductions up to 23% when compared to state-of-the-art alternatives while maintaining performance on simulated platforms, and 16% compared to Linux running on a real mobile system. This holds for large configurations, saving 10% energy with no throughput degradation for up to 256 cores. For these configurations, the measured overhead of SPARTA was <1% of the epoch length, enabling energy savings in large-scale architectures with up to hundreds
of cores.

### 7.4.1 Further reading

Several works have been proposed in the context of task mapping to heterogeneous platforms. The *in kernel switcher* (IKS) [95] is an early solution developed for ARM’s big.LITTLE [46]. IKS enables only one core type to be used at a time and migrates all tasks when the overall system load reaches a certain threshold. ARM’s GTS [3] improve IKS by bringing the per-core utilization awareness to the scheduler. However, they optimize for throughput only and do not consider performance variations from different memory/compute-boundness levels. Furthermore these works are limited to a specific architecture with two core types. Koufaty [63], Saez [105], and PIE [130] introduce performance models to predict workload behavior at runtime considering the different computational/memory capabilities of each core type, but do not account for power. Annamalai *et al.* [2] aims at maximizing the total system energy efficiency without taking performance degradation into account. MTS [69] attempts to maximize the overall system throughput given a maximum power envelope. MTS, however, does not take into account throughput requirements of individual tasks (e.g. priority is given to tasks that provide the highest improvement in overall throughput) nor support multiple tasks mapped to the same core. Procrustes [70] applies MTS to the context of heterogeneous cores with dynamic microarchitectures (e.g. ElasticCore [126]). SmartBalance [111] provides a task mapping heuristic that optimizes the overall energy efficiency of the system. It finds near optimal mappings in terms of throughput/Watt, however it doesn’t consider throughput requirements of individual tasks. Muthukaruppan *et al.* [86] proposes a price theory power management that uses application heartbeats [51] as the QoS metric and aims at minimizing power while maintaining throughput. A similar approach is proposed by Shafik *et al.* [115] and Das *et al.* [27], which both perform QoS-aware DVFS by using reinforcement-learning and control theory. In SPARTA, we similarly use throughput as the QoS metric, and consider
QoS satisfied if the maximum achievable throughput (or target throughput) is being met. [86, 27, 115] focus mostly on DVFS aspects ([86] focuses on clustered architectures such as ARM’s big.LITTLE, while [27, 115] consider only homogeneous cores), while SPARTA focuses on the orthogonal issue of efficient task mapping to HMPs. Furthermore we use a QoS notion based on the predicted task throughput across core types, which does not require the use of external frameworks or modifications on the application.

SPARTA efficiently predicts both task performance and power at runtime, but, in contrast to previous works, SPARTA incorporates per-task QoS into its allocation decisions in order to identify power savings without sacrificing performance. Also, current solutions for performance and power prediction [130, 2, 86, 111, 70, 81] either do not support a generic task and architecture model (i.e. limit the number of tasks that can be mapped to a single core or the number of core types) or do not consider dynamic frequency variability due to ondemand DVFS. SPARTA addresses these issues with MARS’s reflective models that synergistically couple task mapping, frequency scaling, and a model of the Linux CFS scheduler to support the generic Linux task model.

It’s worth reminding that SPARTA is proposed in the scope of a generic Linux environment and single-ISA shared memory HMPs. Other works such as [134, 116, 129] address task partitioning in the scope of heterogeneous processing elements with multiple ISAs (e.g. CPUs vs DSPs vs FGPAs). These techniques work under specific assumptions for the runtime environment and programming models (e.g. availability of multiple implementations for the same function, application ability to reconfigure itself, no shared memory between elements, etc.), therefore are not directly comparable to SPARTA.
Chapter 8

Aging-Aware Load Balancing in Mobile Platforms

In Chapter 7, we described SPARTA, a technique for effectively exploiting power-performance tradeoffs of HMPs by mapping a task to the most power efficient core type that satisfies its performance requirements. SPARTA produces promising energy savings, however, it may bring some issues from a reliability perspective due to the overutilization of certain core types. For instance, [127] shows that the current model for task load balancing in Linux leads to premature aging and wearout of overutilized cores in multicore platforms as shown in Figure 8.1 (a). This happens due to device-level aging mechanisms such as Bias Temperature Instability (BTI) [9] and Hot Carrier Injection (HCI) [14]. These mechanisms are highly dependent on factors such as temperature and core utilization (i.e. stress), which causes transistor-level delay degradation and reduces core performance throughout its lifetime.

In this Chapter we address this issue though ADAMANTE, an Aging Driven tAsk MApPliNg tool that uses aging, performance, and power sensing and prediction in order to map tasks
Figure 8.1: Traditional task mapping (a). Aging-aware task mapping with DVFS (b). Due wearout caused by the aging imbalance in (a), the circuit critical path delay may violate its operating frequency guardband.

© 2017 IEEE

to the most appropriate core type while balancing out aging towards increased lifetime as shown in Figure 8.1 (b). ADAMANT is an extension of SPARTA and augments the models proposed in Chapter 6 with core-level critical path delay degradation aging models for reliability assessment. Experimental results on typical mobile workloads executing on a big.LITTLE architecture demonstrate up to 2x improvement in lifetime with negligible overheads.

The remaining of this Chapter is organized as follows: Section 8.1 further motivates the need for aging-aware task mapping; Section 8.2 and 8.3 presents ADAMANT’s aging models and extensions to SPARTA’s task mapping algorithm; Section 8.4 present our experimental evaluation; Section 8.5 closes the Chapter with our conclusions.
8.1 Motivation and Related work

In multicore scenarios, task mapping/scheduling policies that are unaware of aging may lead to reduced lifetime due to the over-utilization of the same core. In current heterogeneous architectures such as ARM’s big.LITTLE [46], task mapping policies attempt to improve energy efficiency by establishing an affinity between workloads and different core types [3, 111, 70, 136], which further intensifies the aging imbalance. For instance, the GTS[3] policy migrates tasks between big and little cores when the task load reaches a certain threshold.

Figure 8.2 illustrates this by showing the difference of delay degradation (aging imbalance) between the most and the least aged cores in an 8-core big.LITTLE heterogeneous architecture and a 4-core (big-only) homogeneous architecture (simulated on gem5 using the big and little cores described in Table 8.1). After three years, the difference in degradation between the most aged core and the least aged core is \( \sim 46\% \) on the homogeneous platform, while on the heterogeneous platform this difference reaches 100\% for the same workload, despite having a larger number of cores to map tasks to.

Figure 8.2: Relative difference between the most and the least aged cores in homogeneous and heterogeneous architectures. \textit{typical} workload (Table 8.4). GTS scheduling [3]

© 2017 IEEE
8.2 Aging modeling

ADAMANT inherits the sensing definitions and execution model described in Chapter 7. Additionally, now we also assume the availability of per-core critical path aging sensors [56] which enable us to evaluate aging in terms of the relative critical path delay degradation at any given time $t$: $\Delta D_{rel}^{C_j}(t) = D^{C_j}(t)/D_0^{C_j}$, where $D_0^{C_j}$ is the initial critical path delay (measured when the system is first deployed) and $D^{C_j}(t)$ is the current critical path delay of core $c_j$ at time $t$, such that $D^{C_j}(t) < D^{C_j}_{gb}$, where $D_{gb}$ is the maximum delay in which the core can operate at its maximum frequency. Throughout the chapter, we use the terms frequency guardband or guardband slack to refer to the difference $D_0^{C_j} - D_{gb}^{C_j}$; we also refer to $D_{gb}^{C_j}$ as the core’s guardband delay. Critical paths may age at different rates, therefore multiple aging sensors may be deployed [131]. In the remainder of this paper, we assume that $\Delta D_{rel}^{C_j}(t)$ corresponds to the most aged critical path.

8.2.1 System-level delay degradation model

For platforms that do not feature critical path sensors or for offline estimation of aging effects, aging can be assessed using aging models for critical path delay degradation. Using aging models at runtime to replace sensors is usually overhead-less since transistors age at a slow rate and therefore it is not necessary to update $\Delta D_{rel}^{C_j}(t)$ at the granularity of mapping epochs (typically between 50ms-300ms). Throughout the rest of this paper, $\tau_{aging}$ refers to the period of core aging assessment, i.e., the aging epoch, while $\tau_{aging}^k$ denotes the time at the $k^{th}$ aging epoch. When using this aging estimation model, we assume the availability of temperature sensor that all allow us to obtain the average core temperature $T^{C_j}(t)$.

Aging-induced delay degradation in transistors is caused mostly due to BTI and HCI effects [9, 14], therefore we can define the delay degradation during the period of duration $t$ as $D^{C_j}(t) =$
\[ D_{0}^{c_j} + \Delta D_{\text{BTI}}^{c_j}(t) + \Delta D_{\text{HCI}}^{c_j}(t), \] where \( \Delta D_{\text{BTI}}^{c_j} \) and \( \Delta D_{\text{HCI}}^{c_j} \) is the delay degradation due to BTI and HCI since \( D_{0}^{c_j} \) was measured. Given that technology and fabrication-dependent factors are known, the degradation of the delay of the cores’ critical paths can be estimated at runtime using the BTI models from [9] and HCI models from [14] by determining duty cycle \( \delta \), switching activity \( \alpha \), and temperature for all transistors in the critical path. Since such information is not available at system-level, we follow a similar approach to [90], in which we assume all transistors in a component experience similar stress rate and temperature, thus similar aging rates. For core-level aging estimation, temperature can be obtained from sensors, while the average \( \delta_{c_j}(t) \) and \( \alpha_{c_j}(t) \) for a core \( c_j \) throughout a period of duration \( t \) can be defined as [90]:

\[
\delta_{c_j}(t) = \frac{\text{time}_{\text{active}}^{c_j}(t) + \text{time}_{\text{idle}}^{c_j}(t)}{\text{time}_{\text{active}}^{c_j}(t) + \text{time}_{\text{idle}}^{c_j}(t) + \text{time}_{\text{pg}}^{c_j}(t)}
\]  \hspace{1cm} (8.1)

\[
\alpha_{c_j}(t) = F_{c_j}(t) \ast \frac{\text{time}_{\text{active}}^{c_j}(t)}{\text{time}_{\text{active}}^{c_j}(t) + \text{time}_{\text{idle}}^{c_j}(t) + \text{time}_{\text{pg}}^{c_j}(t)}
\]  \hspace{1cm} (8.2)

where \( \text{time}_{\text{active}}^{c_j}(t) \) denotes the amount of time the core is running a task, \( \text{time}_{\text{idle}}^{c_j}(t) \) denotes the amount of time the core is clock gated, and \( \text{time}_{\text{pg}}^{c_j}(t) \) denotes the amount of time the core is on a power gating state during the period \( t \). This models the core at the worst case aging since we assume that duty cycle \( \delta_{c_j}(t) = 1 \) during the entire period the core is not power-gated and that switching activity \( \alpha_{c_j}(t) = F_{c_j}(t) \) during the entire period the core is not clock-gated. Next, we describe in more details how BTI and HCI aging are periodically computed as a function of \( \delta \) and \( \alpha \).

**BTI aging impact:** BTI is a two-phase effect: stress and recovery. During stress phase the transistor is ON and traps may be generated at the interface between channel and gate oxide. This effect is accentuated at higher temperatures and gradually increase the threshold voltage \( V_{th} \). On the other side, when the transistor is OFF, a recovery phase starts in which
some traps are filled, thus leading to a partial decrease of $V_{th}$. Based on the models from [9, 90, 42], we define the delay degradation due to BTI during period of duration $t$ as:

$$\Delta D_{BTI}^{c_j} (t) = A_{BTI} \times (\delta(t) \times t)^n \times e^{\frac{-E_a}{k T_{cj}(t)}} \times (V_{cj}(t) - V_{th}) \times e^{(V_{cj}(t) - V_{th}) \times e^{\frac{-E_a}{E_0}} \times D_{cj}^0}$$

(8.3)

where $A_{BTI}$ and $E_0$ are technology dependent factors, $n$ is a constant depending on the fabrication process, $k$ is a Boltzmann’s constant, and $E_a$ is the activation energy.

In order to capture the recovery phase of in BTI degradation, $\Delta D_{BTI}^{c_j} (t)$ is always computed for the period $t$ since the system was first deployed ($D_{cj}^0$ was measured) and the current aging epoch $\tau_k^{k}$, averaging out the stress and temperature. Therefore, we need to consider the aging history from the previous aging epoch $\tau_{k-1}^{k}$ for each core $c_j$ to compute the delay due to changes in temperature and stress rate. Equation 4 shows that BTI degradation is a weighted function of stress and temperature for current aging epoch $\tau_k^{k}$ based on history (previous aging epoch $\tau_{k-1}^{k}$):

$$\Delta D_{BTI}^{c_j} (\tau_k^{k}) = \Delta D_{BTI}^{c_j} (\tau_{k}^{k}) + \frac{\tau_k^{k} - \tau_{k-1}^{k}}{\tau_k^{k}} \times \delta(\tau_k^{k}) \times e^{\frac{-E_a}{E_0}} \times D_{cj}^0$$

(8.4)

where $\tau_{k}^{k}$ denotes the time period since the system was first deployed and the $k^{th}$ aging epoch. Using this equation we can keep track of aging history as well as recovery phase in case that stress or temperature are decreased in the new epoch.

**HCI aging impact:** HCI is a dynamic mechanism that happens when the transistor is switching, when accelerated electrons inside the channel collide with the oxide interface, creating electron-hole pairs. Hence, the current-voltage characteristic of the transistor changes and leads to increase in $V_{th}$. The models from [14, 90, 42] define the delay degradation due
to HCI for the period $t$ since the system was first deployed as:

$$\Delta D_{HCI}^{c_j}(t) = A_{HCI} \times t^{0.5} \times \alpha(t) \times F_{c_j}^{c_j}(t) \times e^{\left(\frac{-E_a}{kT}x(t)\right)} \times e^{\left(V_{c_j}(t)-V_{th}\right)} \times D_{0}^{c_j}$$  \hspace{1cm} (8.5)$$

where $A_{HCI}$ is a technology dependent factor and $E_a$ is the activation energy. For HCI degradation there is no recovery period, so define $\Delta D_{HCI}^{c_j}(t)$ as the HCI delay degradation of the current epoch and the accumulated aging saved from previous epochs:

$$\Delta D_{HCI}^{c_j}(\tau_{aging}^k) = \Delta D_{HCI}^{c_j}(\tau_{aging}^{k-1}) + \Delta D_{HCI}^{c_j}(\tau_{aging}^k - \tau_{aging}^{k-1})$$  \hspace{1cm} (8.6)$$

where $\tau_{aging}^k$ denotes the time period since the system was first deployed and the $k^{th}$ aging epoch.

### 8.3 Aging-aware task mapping

As an extension of SPARTA, the objective of ADAMANt mapping phase is to find a task mapping that satisfies each task’s performance requirements while reducing the total system power consumption and balancing aging across cores. The optimization goal of SPARTA is redefined as follows:

**Problem formulation:** Given the set of tasks $P(t_k) = \{p_1, p_2, \ldots, p_m\}$ that are active at the $k^{th}$ mapping epoch, our goal is to find the set of all tasks $p_i$ to be mapped to a core $c_j$ during the next mapping epoch, defined as $X_{c_j}$, such that the total power consumption is minimized, tasks performance constraints are satisfied, guardbands are not violated, and all tasks are mapped to cores. The objective function is defined by Equation 8.7:
minimize \sum_{c_j \in C} W^{c_j}(t)

constr : IPS^{p_i c_j}(t) \times U^{p_i c_j}(t) \leq IPS^{p_i \max}, \forall p_i \in X^{c_j}, \forall c_j \in C

constr : D^{c_j}(t) < D_{gb}^{c_j}, \forall c_j \in C

constr : P = \bigcup_{c_j \in C} X^{c_j}

where \( t \) is the time period until the next mapping epoch. We consider the performance constraint met if the task’s effective throughput, defined as \( IPS^{p_i c_j} \times U^{p_i c_j} \) (i.e. the average throughput including periods the task was not executing), cannot be increased by moving the tasks to a faster, i.e. the effective throughput is saturated (defined as \( IPS^{p_i \max} \)). This saturation is the typical case for interactive tasks with computation—IO/sleep cycles. By providing a faster core to such task, it’s IPS during the computation cycles (\( IPS^{p_i c_j} \)) increases, which in turn decreases the computation time and the core utilization (\( U^{p_i c_j} \)), thus limiting the task’s effective throughput. For tasks that we do not observe IPS saturation, we assume that in the best case its performance constraint can only be met by the fastest core. This is the typical case for non-interactive compute-intensive tasks with only computation cycles.

Our choice of IPS saturation metric as the performance target is motivated by the fact that in our workload model tasks do not explicitly define deadlines or performance targets.

8.3.1 Task mapping heuristic

ADAMANT extends the SPARTA’s heuristic solution. The rationale of the task mapping heuristic is to map each task to a core that satisfies the target throughput \( IPS^{p_i \max} \) and results in the smallest increase in total power while keeping the relative aging of cores balanced. The general idea of list scheduling is to order the processes/task to be scheduled/mapped, according to their priorities and map them in order. This heuristic is described in Algorithm 5.
Algorithm 5 ADAMANt task mapping algorithm

1: ⊿ Obtain tasks max. IPS as performance target
2: for all $p_i \in P$ do
3:   $ IPS_{\text{max}}^{p_i} \leftarrow \max \{ IPS^{p_i,c_j}(t) \times U^{p_i,c_j}(t), \forall c_j \in C \} \times \omega$
4:   $ A^{p_i} \leftarrow \{c : IPS_{\text{max}}^{p_i} \leq IPS^{p_i,c_j}(t) \times U^{p_i,c_j}(t)\}$
5:   task_list.add($p_i$)
6: end for

7: sort $\frac{IPS/Watt}{A^{p_i}}$ (task_list)

8: ⊿ Power minimization given aging penalty and perf. constraint
9: for all $p_i \in \text{task_list}$ do
10:   $c \leftarrow c_j : IPS_{\text{max}}^{p_i} \leq IPS^{p_i,c_j}(t) \times U^{p_i,c_j}(t) \times \Delta D_{\text{norm}}(t) \wedge W_{\text{system}}(t)$ is minimized
11: if perf. constraint cannot be met. Find the best effort perf. given the aging penalty
12:   if $c = \emptyset$ then
13:      $c \leftarrow c_j : IPS^{p_i,c_j}(t) \times U^{p_i,c_j}(t) \times \Delta D_{\text{norm}}(t)$ is maximized
14: end if
15: $X^c \leftarrow p_i$
16: end for

First, the throughput of all tasks across all core types is predicted (as described in Chapter 7, Section 7.2.2) and the saturation IPS is set as the throughput constraint (line 3). Note that we correct the throughput constraint by a constant $0 < \omega < 1$. We currently use $\omega = 0.95$ to account for prediction errors. We compute the availability $A^{p_i}$ for a task $p_i$ as the number of core the task can be mapped to without violating its performance constraint. Tasks are added to a list which is sorted according to the maximum energy efficiency that can be obtained (in terms of IPS/Watt) and the given task availability (lines 5, 7), so tasks that are more constrained are mapped first. Finally, the tasks are mapped in the order defined by the sorted list to the core that satisfies the described conditions above (lines 9–16).

In order to account for aging we apply a penalty $0 \leq \Delta D_{\text{norm}}(t) \leq 1$ in the computed performance when optimizing towards the objective. The aging penalty is defined by Equation 8.8 as the core’s relative delay degradation normalized to the difference between the maximum and minimum relative degradation across all cores in the system. Equation 8.8 computes the final penalty by balancing the normalized delay degradation with the difference between the current delay $D^{c_j}(t)$ and the guardband delay $D_{gb}^{c_j}$. The $D_{gb}^{c_j}$ ratio is taken into account in order to avoid an excessive penalty for cases when there is aging imbalance but the guardband slack is high. The computation of the overall system power also incurs an aging penalty as shown in Equation 8.9. $\sigma$ is a constant that can be used to control the
tradeoff between aging balancing and power optimization as well as to set an upper bound to the power penalty applied. We employ \( \sigma \) to avoid excessively high \( W_{system} \) values as \( \Delta D_{norm}^{c_j} \) approaches 0. The rationale is to avoid having a single core completely dominating the value of \( W_{system} \), which may have a significant adverse impact on the mapping energy efficiency since power variations caused by other cores would be negligible. We empirically set \( \sigma \) to 0.1.

\[
\Delta D_{norm}^{c_j}(t) = 1 - \frac{\Delta D_{rel}^{c_j}(t) - \min(\Delta D_{rel})}{\max(\Delta D_{rel}) - \min(\Delta D_{rel})} \\
+ \frac{D_{c_j}(t) - D_0}{D_{gb} - D_0} \\
\]  

(8.8)

\[
W_{system}(t) = \sum_{\forall c_j \in C} \frac{W_{c_j}(t)}{\max(\Delta D_{norm}^{c_j}(t), \sigma)} \\
\]  

(8.9)

### 8.4 Experimental results

#### 8.4.1 Setup

We used the offline simulator described in Chapter 4 to evaluate the effects of task scheduling in long term wearout without long periods of live execution. In this evaluation we consider an 8-core platform with two core types: large 4-way out-of-order (OoO) cores with speculative execution (Big cores) and small in-order cores (Little cores), which are representative of current HMPs for mobile devices. The traces for the offline simulations were collected using the gem5[11] implementation of these cores (the parameters of both core types are described in Table 8.1). The reason we used gem5 instead of the ODROID platform (which
implement similar core types) is the unavailability of area and floorplanning information necessary to perform temperature estimation offline using Hotspot [121]. Our gem5 simulation infrastructure is integrated with McPAT [67] (for power estimation), which provides the area information required by Hotspot. Hotspot’s hotfloorplan tool is used to generate a chip floorplan before temperature estimation. Table 8.2 lists the parameters provided to Hotspot.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Big</th>
<th>Little</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue width</td>
<td>4 (OoO) 2 (in-order)</td>
<td></td>
</tr>
<tr>
<td>LQ/SQ size</td>
<td>16/16 8/8</td>
<td></td>
</tr>
<tr>
<td>IQ size</td>
<td>32 16</td>
<td></td>
</tr>
<tr>
<td>ROB size</td>
<td>128 64</td>
<td></td>
</tr>
<tr>
<td>Int/float Regs</td>
<td>128 64</td>
<td></td>
</tr>
<tr>
<td>L1 I$/$D$ size (KB)</td>
<td>32/32 32/32</td>
<td></td>
</tr>
<tr>
<td>L2 $ size (KB)(^1)</td>
<td>512 128</td>
<td></td>
</tr>
<tr>
<td>VF pairs</td>
<td>2GHz,1V 1.5GHz,0.8V</td>
<td>1.5GHz,0.8V 1GHz,0.7V</td>
</tr>
<tr>
<td></td>
<td>1GHz,0.7V</td>
<td>500MHz,0.6V</td>
</tr>
</tbody>
</table>

Table 8.2: Chip area and thermal specification

Table 8.3 lists the values for the constants required by the aging models described in Section 8.2. In order to obtain these values, we consider that worst case delay degradation for BTI and HCI mechanisms happens when duty cycle is equal to 1 (transistor is always ON) and \((\alpha \times f_{max})\) is equal to 1Ghz, respectively. In [8], it is shown that BTI and HCI impacts are 15% and 5%, respectively for 65nm. This means BTI is dominant and its impact is three times higher than HCI. These impacts get worse for technologies beyond 65nm [18]. Furthermore, In [90], BTI and HCI impacts are 9.4% and 8.5% HCI respectively. Additionally, In [6] says
10% degradation for 32nm in 3 years. Based on these information, we compute the fitting parameters $A_{BTI}$ and $A_{HCI}$ assuming BTI and HCI in worst case degrade critical paths by 12% and 4% in 3 years, respectively, given a fixed temperature of 380$K$.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{th}$</td>
<td>0.25V</td>
</tr>
<tr>
<td>$A_{BTI}$</td>
<td>$2 \times 10^{-4}$</td>
</tr>
<tr>
<td>$E_0$</td>
<td>1</td>
</tr>
<tr>
<td>$n$</td>
<td>1/6</td>
</tr>
<tr>
<td>$k$</td>
<td>$8.617 \times 10^{-5}eV/K$</td>
</tr>
<tr>
<td>$E_a$</td>
<td>0.49eV</td>
</tr>
<tr>
<td>$A_{HCI}$</td>
<td>$5.8 \times 10^{-4}$</td>
</tr>
<tr>
<td>$E_b$</td>
<td>0.49eV</td>
</tr>
</tbody>
</table>

Table 8.3: Aging model parameters

### 8.4.2 Workloads

In our experimental evaluation, we define workload based on the analysis performed by [37], which characterizes popular mobile applications in terms of potential for thread-level parallelism and core utilization. They verified that in a typical scenario only one core is utilized 70% of the time in an 8-core big.LITTLE platform, even when multiple applications are running simultaneously. Based on the observations from [37], we used the $x264$ ($x2$), $bodytrack$ ($bt$)(compute intensive) and $blackscholes$ ($bs$) (memory intensive) applications from PARSEC [10] to devise the mobile-like core utilization scenarios shown in Table 8.4. The applications are executed periodically in order to generate the average load shown in Table 8.4 during an aging epoch. We define two typical mobile scenarios: $typical$ and $typical(heavy)$, while the other cases represent more extreme scenarios in terms of both stress and system idleness.
<table>
<thead>
<tr>
<th>Typical</th>
<th>Typical(heavy)</th>
<th>Light load</th>
<th>Heavy load</th>
<th>All cores used</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt/5/0.04/0.17</td>
<td>bt/5/0.04/0.17</td>
<td>bc/8/0.06/0.11</td>
<td>x2/8/0.97/1.00</td>
<td>x2/4/0.97/1.00</td>
</tr>
<tr>
<td>x2/1/0.30/0.96</td>
<td>x2/1/0.97/1.00</td>
<td>x2/1/0.97/1.00</td>
<td>x2/4/0.30/0.96</td>
<td></td>
</tr>
</tbody>
</table>

PARSEC benchmarks used: blackscholes(bs), bodytrack(bt), x264(x2) simmedium inputs. For x264, frames are read at a 5 FPS rate.

Table 8.4: Workload patterns: benchmark name / number of tasks / max. load per task on a big core / max. load per task on a little core

Figure 8.3: Delay degradation. In the (f)Idle case, only background OS services are running with no active tasks

© 2017 IEEE

8.4.3 Results and discussion

We used our framework to simulate three years of aging on our 8-core platform composed of a cluster of four Big cores and a cluster of four little cores. We compare ADAMANt against Linux GTS scheduling for heterogeneous architectures. For fairness of comparison, we also extend GTS with aging-awareness (GTS AW). GTS AW uses our aging penalty $\Delta D_{\text{norm}}$ as

Figure 8.4: Performance degradation under DVFS frequency capping

© 2017 IEEE
a baseline *virtual load* when performing load balancing (i.e., aged cores tend to have less compute intensive tasks mapped to them). GTS AW illustrates the case in which aging mitigation is applied only within clusters of cores of the same type. For these experiments we assume a guardband of 8% with respect to the maximum frequency of each core. The aging epoch length is 1 *day*, task mapping epoch is 200*ms*, and the DVFS epoch is 50*ms*. $\sigma$ (Equation 8.9) is set to 0.1. We simulate aging for a time frame of 3 years.

Figure 8.3 shows the relative delay degradation for the representative workloads described in Table 8.4. For the *Typical(heavy)* workload (Figure 5.a), GTS violates the guardband within less than a year of runtime. With ADAMANT, we are able to increase the time to guardband violation $\sim 100\%$ (about 8 months). Figures 8.3.b-c show aging scenarios under high stress. For these cases, the guardband is violated at the very beginning of execution due the simultaneous utilization of all *Big* cores. ADAMANT is able to adapt and mitigate aging by prioritizing the use of the *little* cores. It’s worth mentioning that under nominal behavior, the system would be throttled due to violation of thermal constraints [120]. Figures 8.3.d-f show the low stress scenarios. ADAMANT delay degradation is slightly higher than GTS since the $\text{rel}_{\text{norm}}\Delta d$ penalty is also proportional to the distance of the sensed delay and the guardband violation delay. For these scenarios, ADAMANT is able to adapt to the available guardband slack and prioritizes other metrics for task mapping (such as IPS and energy efficiency) as shown in Figure 8.4.

Figure 8.4 shows the effects of aging when aging-aware DVFS is applied, and also compares ADAMANT with GTS in terms of performance and energy efficiency. For these cases, the frequency is reduced when the guardband is violated in order to allow continuous operation. In general ADAMANT yields better performance and energy efficiency. For example, for the *typical* case (no guardband violation) ADAMANT improves performance by $\sim 10\%$. For the case when frequency capping was applied, ADAMANT improves performance by $\sim 7\%$ after three years of aging. One exception can be seen for the *typical* case, in which ADAMANT
improved performance by trading off energy efficiency. In this case the $bt/5/0.04/0.17$ should have been mapped to the little core cluster, however, it is possible that due to $IPS_{\text{max}}^p$ mispredictions, some tasks were moved to the big cluster, thus reducing the energy efficiency.

Figure 8.4 also shows that GTS and GTS AW yield the same performance and energy for both aging scenarios. GTS first chooses to which cluster a task will be migrated and then performs either load balancing (GTS) or aging-aware load balancing (GTS AW) within the cluster, therefore performance and/or power degradation resulting from forcing a task to a different cluster is not expected since GTS AW won’t take this decision. For the cases GTS AW is able to reduce the aging rate (Figures 8.3.a-c), we also observe identical performance after 3 years for both GTS and GTS AW since both will have the same frequency capping in the Big cluster (from 2GHz to 1.5GHz — Table 8.1) after 3 years.

**Overheads:** ADAMANt may incur aging estimation overhead in addition to the overheads discussed in Chapter 7. In a platform that utilizes critical path aging sensors, delay degradation monitoring time is very fast and imposes negligible overhead [43, 33]. Otherwise, induced aging delay degradation can be estimated using the models from Section 8.2. Estimating the aging for 8 cores in our experimental platform takes $60\mu s$ on average. If temperature sensors are not available, temperature must be estimated by using, for instance, Hotspot [121], which would incur an additional overhead of $10ms$. The latency of aging estimation can be high when compared to the latency of the mapping phase, but, as mentioned previously, aging only needs to be assessed at the end of an aging epoch (1 day), therefore we consider this overhead to be negligible.
8.5 Summary and conclusions

Heterogeneity in multicore systems provides better energy efficiency but are more prone to unbalanced workload distribution and premature wear-out in mobile platforms. To mitigate this issue, ADAMANT, an aging-aware task mapping approach for HMP is proposed. ADAMANT is orthogonal to other system-level mechanisms such as DVFS by employing sensing and predictive models for runtime workload characterization. Our experimental evaluation using workloads derived from realistic mobile scenarios shows that ADAMANT improves both lifetime and performance when compared to both the vanilla and an aging-aware implementation of Linux’s GTS load balance for HMPs. For typical workloads, ADAMANT improves lifetime by up to 2x without performance degradation. Under DVFS frequency capping scenarios, ADAMANT improves performance by about 7% when compared to the modified aging-aware GTS.

8.6 Related work

Previous works [93, 74, 44] are able to reduce aging imbalance by prioritizing the least aged cores during task mapping in homogeneous architectures. [93] proposed an adaptive runtime task allocation to meet performance constraints while minimizing energy and maximizing system lifetime. They focused on reducing BTI aging impact for soft realtime multimedia applications and do not consider scenarios with DVFS in which the workload is not known. Furthermore, [93] does not consider the effect of HCI which is one of the dominant aging factors. [74] proposes a joint task mapping/DVFS algorithm for multicore mobile scenarios. They use a combination of aging sensors and aging models to define which cores can run foreground performance-demanding tasks (and the target frequency) Their goal is to balance out cores’ aging induced delay degradation while maintaining performance. [44] proposes
a similar strategy assuming a dark silicon scenario [50], i.e., not all cores can be active at the same time due to power/thermal constraints. They leverage this availability of unused cores to dynamically redistribute the workloads according to the chip thermal profile and estimated aging rates. [103] proposes a task mapping approach in which tasks are compiled to multiple versions to tradeoff performance and error resilience. The code version to use is selected at run time considering the cores’ soft error rates and performance variations. [26] also proposes a compile-time approach. Multiple task maps are statically generated for tasks modeled as directed cyclic graphs (DAGs) or synchronous data flow (SDF) models considering different sets of available cores and their mean time to failure. At runtime, one of pre-generated mapping is chosen for each application depending on the total number of active applications. [106] improves over [26] by detecting intermittent faults at run time in order to make mapping decisions.

The aforementioned works perform aging mitigation through task mapping, however, they are not suitable for heterogeneous architectures since they do not consider the difference in power/performance tradeoffs that different core types can provide. Balancing out workloads without considering these tradeoffs may lead to impaired energy efficiency solutions [111, 30]. [73] assumes that all cores provide the same performance (given the same frequency) which is unsuitable for HMPs. Furthermore, their approach assumes core-level DVFS domains, while current mobile platforms employ cluster-level DVFS [46]. [44] requires offline profiling of the target applications, which limits its applicability to general purpose mobile devices (e.g. Smartphones); DVFS is also not supported, instead, the task mapping is chosen based on the cores’ maximum supported frequency given the current aging state and temperature constraints. [26] and [106] also target scenarios in which the tasks are known and their formal DAG/SDF models are available. [103] has similar limitations and uses custom compiler support. ADAMANT, on the other hand supports generic workload without prior characterization and is orthogonal to DVFS (core-level and cluster-level). When compared to state-of-the-art works that address task mapping for heterogeneous architectures [111, 70, 136,
ADAMANT is the first to consider online sensing of BTI and HCI aging mechanisms.

Modern platforms deploy heterogeneous many-core platforms (HMPs) on a single chip in order to support highly diverse and unpredictable workloads while addressing energy efficiency. Typically, heterogeneous platforms deploy architecturally different cores with a range of performance capabilities that impact numerous factors in a uniform manner. For instance, in ARM’s big.LITTLE architecture [3] the big cores have both more cache (4MB vs 1MB L2$) and computational capacity (out-of-order multi-issue pipeline vs in-order pipeline) than the little cores. However, as previous work has shown [130, 49], memory and computational needs can vary independently across applications, and therefore should not be treated uniformly. Maintaining a simple cpu while increasing cache capacity can be a more energy efficient approach to increasing performance compared to simply scaling both cache and cpu size together.

Figure 9.1 shows the throughput in instructions per cycle (IPC) for the typeset and gsm benchmarks of the MiBench benchmark suite when cpu and cache resources are varied independently. Each line in the left plot of Figure 9.1a represents an individual execution of the typeset benchmark while varying only cache parameters with constant cpu settings.
The lines in the right plot represent executions of the same benchmark with constant cache configuration, but varied cpu settings (e.g. Table 9.1). Figure 9.1b contains the same information for the gsm benchmark. Observe that while typeset experiences throughput improvement similarly when increasing both core and cache parameters, gsm only experiences such improvement when increasing core parameters. We can use these observations to determine what resource type bounds each benchmark – that is to say, which resource type yields performance improvements. In this case, we can classify gsm as compute-bound, since only increasing the available cpu resources result in performance increase. Similarly, we can classify typeset as both compute- and memory-bound: as long as inputs are readily available, the performance increases proportionally with both cache and cpu resources allocated. This simple example illustrates the variability of performance impact memory and compute resources can yield depending on the executing workload. For this example we focused on two specific benchmarks, but the observation applies to a variety of benchmarks we discuss.

Figure 9.1: IPC for the (a) typeset and (b) gsm benchmarks when increasing either cache (left) or cpu (right) resources individually.

Figure 9.2a shows the ratio of instructions per Joule (IPJ), which reflects energy efficiency, observed by a subset of PARSEC and Mibench applications when increasing compute and
memory resources independently. Each of the four bars is the ratio of two unique IPJ values: one resource type and size is kept constant for both scenarios, while the other resource type is varied.

The *typeset* and *bodytrack* applications always benefit more from increased cache capacity, while *bitcount* and *susan* experience more improvement when granted increased computational capability. Some applications like *swaptions* simply demand more resources of both types. What these observations tell us is how we can increase resource allocation for an application in the most energy efficient manner. We can combine this with the resource-specific IPC trends to determine the most desirable core configuration for an application. For example, we know from Figure 9.1a that *typeset* benefits proportionally with an increase in both cache and cpu. If we combine that with the information in Figure 9.2a, we know that for this application it is likely more beneficial to increase the cache allocation over the cpu allocation. These per-application averages give us insight into the effects of such decisions, but do not tell the whole story. They only identify and highlight the differences between applications which may be concurrently executed as part of the same workload.

Furthermore, a single application’s limiting factor or resource efficiency may dynamically change during execution. Figure 9.2b shows, through the course of a single execution, that *bodytrack* utilizes compute resources more efficiently at times and memory resources at others. For most of its execution, *bodytrack* utilizes memory resources more energy efficiently (i.e. less cpu, more cache), but periodically it enters a phase in which cpu resources are more energy efficient (i.e. more cpu, less cache). These observations of variability both between and within applications lead us to believe that varying computational and memory resources independently for heterogeneous platforms is beneficial.

In this chapter we perform a design space exploration of *asymmetric heterogeneity*, in which we decouple the scaling of compute and memory resources for architecturally differentiated HMPs and explore its effect on the energy-efficiency and performance of diverse workloads.
Figure 9.2: (a) Improvement in IPJ for various benchmarks when increasing resources in one
dimension at a time, e.g., the first bar for all cases represents the ratio $IPJ_{\text{Big cpu, Big cache}} / IPJ_{\text{Big cpu, Little cache}}$.
(b) IPJ throughout the execution of the bodytrack benchmark on cores with opposite cpu
complexity and cache size.

We perform a study on the achievable energy efficiency of asymmetric HMPs for various
workloads, and compare to the uniform big.LITTLE model. The next segment of this letter
establishes the architectural assumptions, the execution model, and the composition of the
workloads under test. Subsequently, we present and discuss a subset of our compelling
experiments for both statically and dynamically scheduled workloads on HMPs.

9.1 Asymmetric HMP Architecture and Workloads

We consider aggressively HMP platforms consisting of many heterogeneous resources on a
single chip. We explicitly define heterogeneity as, but not limited to, the way it manifests
in the form of both processing elements (cpu) and on-chip memories (cache). A cpu type is
defined by a combination of micro-architectural features, while a cache type is defined by its
capacity. Without loss of generality, this letter considers four cpu types ranging from a small
in-order cpu (Little cpu) to a large 8-way out-of-order (OoO) cpu with speculative execution (Huge cpu). These cores can be paired with private L1 data and L2 caches also ranging from Little to Huge. Tables 9.1 and 9.2 summarize the parameters of each cpu and cache type considered in our experiments respectively. A core type is defined as a unique cpu-cache combination (e.g. the Little-Big core type combines a little cpu type with a big cache type), therefore, in this work we evaluate platforms consisting of up to 16 core types. Each core type supports multiple voltage/frequency (VF) pairs.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Little</th>
<th>Medium</th>
<th>Big</th>
<th>Huge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue width</td>
<td>1(In-Order)</td>
<td>2(OoO)</td>
<td>4(OoO)</td>
<td>8(OoO)</td>
</tr>
<tr>
<td>LQ/SQ size</td>
<td>8/8</td>
<td>8/8</td>
<td>16/16</td>
<td>32/32</td>
</tr>
<tr>
<td>IQ size</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>ROB size</td>
<td>64</td>
<td>64</td>
<td>128</td>
<td>192</td>
</tr>
<tr>
<td>Int/float Regs</td>
<td>64</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
</tbody>
</table>

Table 9.1: Heterogeneous cpu parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Little</th>
<th>Medium</th>
<th>Big</th>
<th>Huge</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1D size (KB)</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>L1D associativity</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>L2 size (KB)</td>
<td>64</td>
<td>256</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>L2 associativity</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 9.2: Heterogeneous cache parameters

Workloads:  Our experimental workloads are constructed using two different approaches. Workloads are made up of benchmarks from MiBench and PARSEC benchmark suites, as well as the synthetic microbenchmarks described in Chapter 6.

The first set of workloads (mixes 0-13 in Table 9.3) represent realistic homogeneous use-cases. Mixes 0, 1, and 6 consist of network, automotive/industrial, and consumer benchmarks from MiBench respectively. Mixes 2-5 each consist of a single benchmark from PARSEC representative of computer vision, artificial intelligence, animation, and data mining respectively. Mixes 7-10 are composed based on the analysis performed by [37], which characterizes mobile use cases in terms of core utilization. We use the same microbenchmarks from mixes 14-18 to define four typical mobile interactive workloads based on the resulting utilization of big
<table>
<thead>
<tr>
<th>Mix 0</th>
<th>Mix 1</th>
<th>Mix 2</th>
<th>Mix 3</th>
<th>Mix 4</th>
<th>Mix 5</th>
<th>Mix 6</th>
<th>Mix 7</th>
<th>Mix 8</th>
<th>Mix 9</th>
<th>Mix 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC32</td>
<td>basicmath</td>
<td>bodytrack</td>
<td>canneal</td>
<td>fluidanimate</td>
<td>streamcluster</td>
<td>jpeg</td>
<td>CB/LU</td>
<td>CB/LU</td>
<td>CB/HU</td>
<td>CB/LU</td>
</tr>
<tr>
<td>blowerfish</td>
<td>bitcount</td>
<td>typeset</td>
<td>(typical)</td>
<td>(typical heavy)</td>
<td>(heavy)</td>
<td>(idle)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>patricia</td>
<td>sport</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dijkstra</td>
<td>mucca</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mix 11</th>
<th>Mix 12</th>
<th>Mix 13</th>
<th>Mix 14</th>
<th>Mix 15</th>
<th>Mix 16</th>
<th>Mix 17</th>
<th>Mix 18</th>
<th>Mix 19</th>
<th>Mix 20</th>
<th>Mix 21</th>
</tr>
</thead>
<tbody>
<tr>
<td>x264 2fps</td>
<td>x264 5fps</td>
<td>x264 15fps</td>
<td>CR/LU</td>
<td>CR/MU</td>
<td>MB/HU</td>
<td>CR/HU</td>
<td>canneal</td>
<td>bodytrack</td>
<td>canneal</td>
<td></td>
</tr>
<tr>
<td>x264 5fps</td>
<td>x264 15fps</td>
<td>x264 30fps</td>
<td>MR/LU</td>
<td>MR/MU</td>
<td>MB/LU</td>
<td>CR/LU</td>
<td>fluidanimate</td>
<td>streamcluster</td>
<td>fluidanimate</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MR/HU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MR/LU</td>
</tr>
</tbody>
</table>

Table 9.3: Benchmark Mix composition

and little cores: mix 7 is a typical load; mix 8 is a typical load with a heavy task; mix 9 is a heavy load; and mix 10 is a light load. Mixes 11-13 consist of a combination of x264 with different frame-rate inputs in order to invoke further target throughput variation.

The second set of workloads (mixes 14-21 in Table 9.3) are diverse: the benchmark mixes comprise a combination of the most compute- and memory-bound benchmarks from PARSEC, as well as synthetic microbenchmarks that exercise different levels of compute and memory boundness. For each suite, we analyzed each benchmark’s performance gain in terms of IPS when increasing core size, and selected the subset that benefited most for the respective resource types; mixes 19-21 consist of these PARSEC benchmarks. Mixes 14-18 consist of synthetic workloads using microbenchmarks that can be either compute- or memory-bound (CB or MB) and impose a high, medium, or low average cpu utilization (HU, MU, or LU).

**Experimental platform:** In order to perform our analysis, we simulated the execution of all benchmark mixes on multiple platforms composed of different combinations of the 16 possible core types considered in this work. We used gem5 integrated with McPAT, as described in Chapter 4, to simulate the heterogeneous platforms. However, running thousands of simulations in gem5 for design space exploration purposes proved to be impractical. Therefore, we used the offline simulator presented in Chapter 5 to evaluate large-scale HMPs efficiently for a variety of platform configurations.
9.2 Experimental Results and Discussion

DSE approach: In order to perform meaningful comparisons, we first define a platform consisting of *four* Big-Big and *four* Little-Little cores as our baseline platform (the common configuration of most big.LITTLE implementations), and use area and performance/power information of the baseline to guide our analysis. Our design space exploration steps are outlined below: *i*) Using McPAT, we find all possible platforms with the same area (with a tolerance of 1%) as the baseline (to trim our search space, we also limit the number of core types used in the same platform to 4). *ii*) We simulate all workload mixes listed in Table 9.3 on all platforms found in step (i) and we obtain the throughput in terms of instructions-per-second (IPS) and energy efficiency in terms of IPJ for each workload mix on each platform configuration. For all runs, the total number of threads is half the number of cores in the baseline configuration, which we believe is a valid use-case for heterogeneous many-cores. *iii*) For each workload mix, we keep only the configurations whose IPS is greater than or equal to the IPS obtained from the baseline configuration (with a tolerance of 1%).

We perform the steps outlined above on two exploration scenarios. First, we fix the core frequencies and task-to-core mapping in order to assess the impact of computation and memory resource variability alone on energy efficiency (Figure 9.3a). The second case (Figure 9.3b) considers a more realistic and dynamic scenario in which task mapping and core frequencies vary at run time. In Figures 9.3a and 9.3b, the bars indicate the energy-efficiency of the baseline configuration (BB=4 LL=4), while the range indicators show the minimum and maximum achievable energy efficiency by all configurations that match the area and performance of the baseline configuration. Table 9.4 specifies the most energy-efficient configurations for the corresponding simulations in Figure 9.3a.

**Static mapping/frequency results:** For the analysis shown in Figure 9.3a, all cores execute at a fixed frequency (1500 MHz) with static task-to-core mapping. In this scenario, we fix
Figure 9.3: Energy efficiency of the baseline configuration. Range bars represent the energy variability across all configurations that have the same performance as the baseline.

Figure 9.4: Selected configurations on typical mobile workloads.

Table 9.4: Most energy-efficient configurations for each workload (the static scheduling/fixed frequency)
the task-to-core mapping to the one that maximizes energy efficiency given the throughput constraint defined by the baseline configuration (found using an optimal brute-force algorithm). Combining the information in Table 9.4 and Figure 9.3a we observe that there are cases in which the baseline IPJ is very near the best achievable IPJ for some workloads (e.g. 2, 3), and in some of those cases the best-IPJ platform configuration is very similar to the baseline configuration (e.g. 7, 8, 19, 21). We also observe cases with diverse configurations that save significant energy compared to the baseline (e.g. 0, 11, 17, 18). Lastly, there are cases in which the energy efficiency range is large (e.g. 0, 7, 11, 17, 18), and small (e.g. 2, 3, 5, 19). This simple exploration highlights the diversity both within and between workloads in terms of IPJ in a range of configurations that fit an area and performance envelope. For some cases (e.g. mixes 2 and 3) there are negligible opportunities for improvements. In these cases, the workloads are composed by four heavy tasks which efficiently use the four *Big-Big* cores in the baseline configurations.

Focusing on Table 9.4, we can see that there exist some configurations that are the most energy-efficient for multiple workload mixes. The first of these configurations is $BB=4\ HL=1$ for mixes 2-5. Each of these mixes consist of a single PARSEC benchmark, which are typically consist of high utilization and large working sets hence mostly large cpus and caches. The second configuration is $HL=2\ ML=1\ LM=1\ LL=4$ and applies to mixes 7, 8, 10, 14, and 16. All of these mixes are diverse and consist of concurrent compute- and memory-bound microbenchmarks with high and low utilization, hence a range of cpu and cache sizes.

**Dynamic mapping/frequency results:** In the DSE performed for this scenario, we consider dynamic task mapping. Many works propose dynamic task mapping mechanisms for heterogeneous architectures (e.g. [130, 81, 70, 3, 86]). We consider Linux GTS dynamic scheduler [3] since it’s the standard approach being used for these platforms. We also consider two DVFS implementations: per-core, where each core’s voltage/frequency pair is determined independently using the Linux ondemand governor; and per-cluster, where cores of the same
type are clustered, and voltage/frequency is assigned at cluster granularity. Figure 9.3b plots the design space for both DVFS case. We can see that the same observations can be made for this set of experiments as for the static case. We can also observe that per-core DVFS consistently leads to better energy efficiency when compared to per-cluster DVFS (the approach used by big.LITTLE). Note that in this paper we focus on heterogeneity asymmetry and do not consider the overheads involved in enabling fine-grained DVFS.

**Mobile workloads case study:** Figure 9.4 identifies an opportunity afforded as a result of the observations from the previous analyses. In this set of experiments, we select a subset of workloads that all experience extended periods in which the cores are not completely utilized (low to medium load scenarios, as shown in [37], are the most typical case in the mobile computing domain), and identify three configurations that are more energy-efficient than the baseline for most of these workloads. Two of these configurations ($BB=1\ LL=12$ and $BL=2\ LM=7\ LL=3$) are more energy-efficient than the baseline in all cases, and save more than 20% energy on average. The $BL=2\ LM=7\ LL=3$ configuration consists mostly of smaller cpu/cache types, and are the most efficient for low to medium load scenarios. The $BB=1\ LL=12$ configuration is able to save 20% energy compared to the baseline on average while limited to the same core types. This shows that by reconfiguring contemporary architectures we can save energy for typical mobile workloads, and with more aggressive heterogeneity we can improve energy efficiency even further.

### 9.3 Summary and Conclusions

Through our analyses, we showed that many workloads do not always require uniform scaling of all resource types. By identifying workloads’ behavior, we can find more energy-efficient heterogeneous core configurations with asymmetric allocations of compute and memory resources. Although the ideal platform configuration for diverse workloads may vary, we were
also able to identify a small set of platforms that consistently reduce the energy consumption of a number of typical mobile workloads when compared to contemporary HMPs. Current octa-core SoCs that implement big.LITTLE architecture (e.g. Samsung’s Exynos Octa, Mediatek’s Helio, and Qualcomm’s Snapdragon 810) deploy the same number of big and little cores for mobile applications. Our analysis has shown that this a good configuration only for very heavily loaded systems, which is uncommon for typical mobile scenarios. Energy improvements of 20% on average can be achieved by reducing the ratio of Big/Little cores (such approach is already deployed by Samsung’s Exynos Hexa and Qualcomm’s Snapdragon 808). Furthermore, by simply decoupling cpu and cache resources, we can configure platforms in the same area and performance envelope as common HMP platforms that increase energy-efficiency by 30% on average. Our experiments on asymmetric heterogeneity also further motivate the use of dynamically configurable microarchitectures and caches ([126, 70]) as well as identify new opportunities for sophisticated dynamic scheduling policies that prioritize energy efficiency by exploiting workloads’ compute- and memory-boundness.
Chapter 10

Conclusions

This thesis described MARS, a middleware and framework for managing multiple resources in the scope of energy-limited heterogeneous multiprocessors. We advocate the extensive use of models to predict how the system will react to actuations before any action is taken, thus creating a reflective system that can more easily adapt to changes at runtime. The cases studies show that this approach is a promising scheme to pave the path towards more energy efficient heterogeneous systems. We believe that MARS opens up several directions for future work:

**Increased levels of heterogeneity:** we currently support single-ISA heterogeneous architectures, thus a natural progression is to include support for CPU+GPU processing, as well as other accelerators such as FPGAs. Jointly managing performance and power given that applications are already partitioned across the heterogeneous elements (e.g., joint CPU+GPU DVFS) can easily be integrated into our reflective middleware, however several challenges arise if such partitioning has to be performed at run-time. Runtime systems and languages such as OpenCL provide mechanisms to generate computational kernel code for different processing elements at runtime, however the partitioning is still fixed by the
developer. Fully automating this process would require models with the ability to predict application performance and power across these heterogeneous resources such as CPUs, GPUs, and FPGAs. Building these models is extremely challenging since there is no correlation between performance and power across these elements. Some existing solutions attempt to address these problems by providing pre-profiled/pre-generated building blocks to create applications (e.g. the RACECAR framework[134]) or fully profiling the target applications and generating the possible partitionings before hand (e.g. Invasic computing[133]).

**Distributed architecture support:** our current implementation for real platforms runs on top of Linux and assumes a coherent shared memory model where the user-level daemon running the policies has a global view of the system. A straightforward implementation of our virtual sensing/actuation interface for a distributed architecture would be to partition the fabric into resource management domains (we partially support this, as shown in Figure 1.6 with the concept of clock domains) and create multiple instances of the daemon for each domain. Naturally policies may need to be modified for the required coordination. From the infrastructure perspective, each daemon would only be able to access accurate sensing information from its own domain, while information from other domains would be available in an approximated or delayed manner. How to trade-off sensing information accuracy for less network traffic or more responsive policies is still an open problem.

**Memory as a resource:** we focus mostly on allocating and managing processing resources. However whether or not the same approach can be applied to more passive components such as memories is still an open issue. The performance and energy efficiency on architectures that rely on non-coherent software-controlled memories is directly affected by the proximity of the processing elements and the data location. The same is also true for typical cache-coherent architectures with multiple memory controllers (though caches might be able to hide the latency of accessing far-away controllers). Abstracting memory allocation request as tasks that need to mapped to a resource would open up opportunities for optimization.
Self-trained models: our current reflective model is a combination of regression-based performance and power prediction[84], heuristic-based models[84, 30] and analytical models[80]. Given that for any sensing window, the prediction error of the previous window can be computed by keeping information from previous prediction, the models can be improved by including error correction mechanisms. For instance state-space models are naturally feedback driven and could be a promising replacement for regression models. However, introducing error correction to heuristics that model highly non-linear behavior (e.g. the completely fair scheduler model from [84]) is a much greater challenge.

Machine learning: machine learning is prospective replacement for heuristic-based and analytical models. Furthermore, the infrastructure provided by our middleware makes it much easier to collect system performance data at various levels of granularity. Unsupervised machine learning technique could be used to mine this data and find patterns useful for optimizing existing policies.

Policy supervisors: Some applications may require formal or stronger guarantees to ensure that certain parameters of the system do not exceed a limit, or remain within a given range of values. Though not addressed in this dissertation, our framework provides the infrastructure to easily implement control theory based policies that provide such guarantees. Furthermore, supervisory control theory is also a promising technique to manage policies that require coordination while providing formal guarantees.
Bibliography


