Self-Adaptivity Towards Tomorrow's Heterogeneous Computing Systems

BY

ETTORE M. G. TRAINITI
B.S., Politecnico di Milano, Milan, Italy, 2013

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2016

Chicago, Illinois

Defense Committee:

John Lillis, Chair and Advisor
Wenjing Rao
Marco D. Santambrogio, Politecnico di Milano

ACKNOWLEDGMENTS

It might sound obvious but I want to thank the most patient people on Earth, my parents. Not only they let me explore the unknown fields of my dreams and kept up with my blazing high expectations, but they patiently supported me in any way they could, demonstrating that behind apparently ordinary faces there are instead extraordinary human beings.

To all the people I have met during these years, in sparse order friends, relatives, professors, classmates and random strangers, I would like to say thanks, because of the amazingly good and struggling bad times you made me experience I became who I am today.

Sincerely,

EMGT

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Program Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DVFS	Dynamic Voltage and Frequency Scaling
FPGA	Field Programmable Gate Array
GPU	Graphic Processing Unit
HB	Heartbeat
HMP	Heterogeneous Multi-Processing
HSA	Heterogeneous System Architecture
HT	Hyper-Threading
HW	Hardware
IPC	Inter Process Communication
NoC	Network on Chip
NUMA	Non-uniform Memory Access
ODA	Observe-Decide-Act
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing

LIST OF ABBREVIATIONS (continued)

OS	Operating System
SMT	Simultaneous Multi-Threading
SoC	System on Chip
SW	Software
VM	Virtual Machine

SUMMARY

During the last decades, Information and Communication Technologies adoption rate increased enormously. Computational resources have been used for the most various applications, from handling our everyday communication channels to crunching numbers for solving problems essentials to human kind improvement and evolution. The power consumption footprint that generates from those technologies is now a major concern given its not negligible impact.

Computing Systems play a key role in finding the solutions we are looking for (e.g. drug researches [1], simulations [2], disaster prevention [3]) but currently deployed ones have been designed around workloads that are now not common. While in the past they served rather static workloads with certain performance requirements, nowadays the variety of the on-demand computing scenario is instead characterized by dynamic workloads, each having different performance constraints and criticality. The advent of cloud technologies and the unpredictability of the different continuously changing workloads demand a redesign of these systems for a better exploitation of today's available computational resources.

A promising and widely investigated approach to deal with the problem briefly introduced above is what has been called Heterogeneous Systems Architecture (HSA [4]), the idea of combining different families of computational resources (CPUs, GPUs, FPGAs) to achieve higher computational power and efficiency. By better exploiting the peculiarities of each component architecture it is possible to tackle down many of the challenges posed by the on-demand computing scenario.

SUMMARY (continued)

Nowadays Embedded platforms include multiple kind of resources, e.g the ODroid XU3 [5] board exposes two families of quad-core CPUs and a hexa-core GPU. The peculiarity of Embedded Systems is the following one: they are designed with power efficiency and performance constrains in mind. Combining many of them, like the University of Southampton did in 2013 with 64 Raspberry-Pi nodes, leads to cheap and increasingly powerful distributed computing systems.

Even though HSA seems to be the computing panacea the community is looking for, the complexity it introduces is a burden that the end-users of these systems cannot easily cope with.

This research thesis aims at addressing all the previous mentioned limitations i.e. efficiency,dynamic workloads and the complexity barrier by leveraging Self-Adaptivity. Self-Adaptivity allows the system to autonomously decide which specialized computing resources have to be used to achieve a more efficient execution based on user-defined optimization goals, such as performance, energy and reliability while drastically lowering the complexity for the end-users.

Our research had to cover many aspects that are self-awareness, monitoring infrastructures, decision mechanisms and actuation phases to build a component able to enact the vision proposed by self-adaptivity. The Orchestrator is the component responsible to observe each running application, decide which resource configuration better satisfies the performance and goal constraints exposed by the application itself and act on this decision. This feedback loop is the core of the Orchestrator, the ODA loop. Observation is done by using the Heartbeats API, a lightweight framework aimed at performance monitoring. Each application hence tells the system how it is performing. Deciding on which of the available resources the applications should

run on is determined by a run-time selectable resource management policy. Lastly acting on the decision happens through mechanisms like task mapping and execution migration.

Based on the applications performance requirements, the current system usage and the kind of different implementations available, the Orchestrator configures the system to meet as many application goals as possible.

The experimental results of our solution show that the decision mechanisms of the Orchestrator are able to achieve relevant improvements over standard non self-adaptive approaches. Our fine strategies not only led to applications executions closer to the declared goals but also achieved power consumption benefits that would have otherwise been left unexploited.

The structure of this work is organized as follows :

- Chapter 1 defines the context of our work and briefly describe its motivations
- Chapter 2 provides a general background overview needed for this work
- Chapter 3 presents the most important works related to adaptive systems
- Chapter 4 discusses the problem we want to address and our solution
- Chapter 5 shows the way our work is implemented, based on the descriptions of the previous chapter
- Chapter 6 illustrates the results obtained by our solution in different scenarios
- Chapter 7 covers the results of our solution together with its limits that can be a starting point for future researches.

CHAPTER 1

INTRODUCTION

In this chapter, we provide the reader an introduction to self-adaptivity for computing systems. In Section 1.1 we present the working context, in order to better understand the problems we are solving with our approach and our goals. Section 1.2 shows the idea behind monitoring for performance awareness. Then in Section 1.3 the tasks of a self-adaptive system are shown. Finally in Section 1.4 our solution is briefly discussed.

1.1 Context definition

The number of computing devices is expected to triple by 2020 [6]. Each single device not only contributes to the energy consumption that our society has to sustain but also adds its share to our world carbon footprint. New methodologies and technologies have to be developed to solve these challenges.

In a computing world where workloads are increasingly becoming dynamic, due to the on-demand scenario triggered by distributed services and cloud computing, a better exploitation of currently available hardware resources can drastically improve our systems' efficiency. A system should basically adapt to better respond to the workloads it is executing by knowing how each single workload is characterized. To do so the applications have to be monitored and based on these information the system will take its decisions. This involves monitoring, profiling of applications at run-time, decision strategies and the ability to move tasks from one

resource to another. Complexity, evidently, is a major fall-back of this approach. If a system could take care of all of those tasks by itself, we could forget about complexity and achieve our efficiency goal. What we are looking for is self-adaptivity, our solution.

1.2 Application monitoring for self-adaptivity

In order to tackle down efficiency with self-adaptivity, it is essential to monitor the running applications and extract their performances.

There are many approaches to monitoring that range from measuring the number of instructions executed in a period of time to CPU utilization. Cache miss rates can also be used as an alternative to perform the same task. Measuring using those ways though does not provide any insight on the application desired performance.

We used a software interface called Application Heartbeats [7] that extremely simplifies monitoring while showing relevant performance information. This API measures application progress towards its set goal by computing, over a period of time, how many heartbeats were emitted at significant points during the application execution. This is essential for the externally driven and self-tuning concepts behind self-adaptivity.

1.3 Adaptive Operating Systems

The availability of operating systems and run-time support for coprocessors and reconfigurable fabrics is essential for the success of HSAs. Operating Systems are be the key element for HSA adoption, hence they are supposed to leverage different execution paradigms and make the development of applications for HSA easier. They should also embrace self-awareness through flexible monitoring infrastructures to change their resources configurations to exploit the un-

derlying hardware at its best. Such a broad and complete solution is not yet available. By combining the Heartbeats monitoring Framework, policies and actuators, we developed the component that tackles down those problems : the Orchestrator.

1.4 The Orchestrator approach

Our component leverages the idea of autonomic computing to enact self-adaptivity.

An Observe-Decide-Act (ODA) is at the core of the Orchestrator. Hardware resources are monitored to collect information on their status. Applications information, like goals and throughput, are gathered and collected at run time. This constitutes the observation part.

All of these data are then fed to the decision mechanism which takes care of automatically determining the mapping of the applications to the computational resources to satisfy the applications' constraints. Any decision strategy can be added to the system for future expandability.

Once the Orchestrator has taken its decision, the apps are moved to the assigned resources based on the configuration outputted.

CHAPTER 2

BACKGROUND

This chapter presents the background needed to understand this work and its objectives. We start with the concept of Heterogeneous System Architecture in Section 2.1. We then discuss about different computing resources in Section 2.2. In Section 2.3 we present some systems that embrace the idea of HSA. Finally in Section 2.4 we present the reason why we chose an hybrid approach.

2.1 Heterogeneous System Architecture

Heterogeneous system computing is the idea of combining different families of computational resources and use them to accelerate the performed tasks. Each resource not only is designed for a specific task but exposes architectural differences that are essential to achieve the efficiency they are required to deliver. Our expectations from systems indeed include computational tasks that have to be completed as fast as possible (think of simulations and predictions) and real time application that have to perform over a certain threshold for usability (e.g videos and games). Many devices currently available on the market are intrinsically heterogeneous. Modern smartphones are made of Systems on Chip (SoC) that consist of multiple core architectures, GPU for animations and User Interface, DSPs for task offloading and customized chips for better efficiency. Consider an architecture from late 2006, the Cell [8] processing system delivered in Sony Playstation 3 [9] consoles. The Cell processor consisted of a single core PowerPC

processing element combined with eight Synergic Processing elements (SPE), units specialized in vectorized floating point operations. The PlayStation 3 also included a Graphic Processing Unit used for rendering and shading geometry. Overall the system constantly deals with three different computing elements to deliver entertainment to its end users. Each computational resource in a HSA can be exploited for delivering efficiency due to its intrinsic architecture characteristics. GPUs are great for 2D and 3D tasks but they are also highly efficient for massively parallel tasks given their SIMD nature. The downside of exploiting and managing a HSA is to program applications for it, being able to exploit its heterogeneous hardware characteristics and the complexity of its possible configurations. Universities and Consortium like the HSA foundation [4], are working towards easening the process of programming for HSA and get the best efficiency out of them.

2.2 Computing Resources

The core of any computation is the resource that actually performs this tedious task. A resource can be either physical or abstracted through virtualization, hence be a virtual resource. We are going to introduce the peculiarities of the three most commonly used computing engines : CPU, GPU, FPGA. Depending on their domain usage, companies tend to integrate more of a family than others. A rendering farm would focus more on GPU technologies while a storage intensive system focuses more on CPU ones. Virtualizing computing resources is instead an approach to lower the complexity of the management of those system. The number of virtual cores used for a task can be increased or decreased based on resource utilization or

resources availability. In the following subsections we will focus on physical resources, leaving the abstraction for the next chapters of this research work.

2.2.1 Central Processing Unit

CPU have been the main way of doing computation for decades. Since the release of one of the first commercial microprocessor, the Intel 4004 in 1971, the efforts putted by the semiconductor companies to increase the performances of this component has never stopped to give amazing results. Being really close to Moore's law, the number of transistors in those device doubled approximately every 18 months. Until end of the Intel Pentium 4 era, increasing the operating frequency has been the main way to gain performances. Programmers were fooled by the fact that their software, even not optimized, would run faster and faster due to hardware improvements. Once the power wall has been hit, hardware makers leveraged heavily ways to extract more instruction level parallelisms, task parallelism and optimized processor pipelines to respond to the performance improvement demanded by the market. Simultaneous Multi-Threading technologies were a first approach to this problem. Further developments, e.g. dual silicon die processors, lead to the solution of using multi-core architectures where, instead of pushing the frequencies higher, the structures of a processor is replicated multiple times. One of the first dual core processor is the IBM Power4 released in 2001. Nowadays commercial server platforms can embrace multi-socket solutions with 18-cores processors for each socket. Following Flynn's Taxonomy [10], the shift from Single Instruction Single Data (SISD) to Multiple Instruction Multiple Data (MIMD) characteristic of multi-core and multiprocessors systems, demand for a software redesign to leverage the parallelism needed for the next performance

milestone. New solutions branded as Many Integrated cores architecture, like the 61-core Intel Xeon Phi, are emerging and are parts of the fastest super computers [11].

2.2.2 Graphic Processing Unit

GPU have been commonly classified as Single Instruction Multiple Data (SIMD) architectures. Being employed in visual applications where the same operations have to be performed over and over on a multitude of data, they evolved with massively parallel tasks in mind. Specialized units for shading and rendering have multiplied over time to handle the increased complexity of geometry scenes used by scientific simulations and games. Due to their architectural design, the idea of exploiting them for parallel computing has boomed, leading to the General Purpose GPU (GPGPU) paradigm. Initially their usage was tedious, involving visual color coding for information encoding and results extractions. In the last years abstraction frameworks like Nvidia CUDA [12] and the Open Computing Language(OpenCL)[13] made their use easier for programmers. Architectural design changes lead GPU to be composed of blocks of SIMD execution unit, which can then be considered MIMD architectures. To better leverage the intrinsic parallelism of GPU, though, the applications have to be redesigned with this paradigm in mind. Once this happens, the high energy efficiency nature of these computing resources can be fully exploited.

2.2.3 Field Programmable Gate Array

Hardware custom design for specific tasks have always been the most performant and efficient way to do computation. Imagine having a customized chip for each application you run, not only this would be unfeasible but it would also have an enormous cost. Designing custom

hardware is a really difficult task, additionally it takes years for the end user to benefit from them. Broadly speaking and over simplifying, both CPU and GPU are immutable hardware designed around general usage scenarios. An architecture characterized by reconfigurability and lower design complexity is represented by the class of devices called FPGA. The end user is free to implement and accelerate in hardware the most heavy parts of its computational flow, without spending millions of dollars for an ASIC implementation prone to not fixable errors due to mistakes during the design phase. Through languages like Verilog and VHDL, software specifications are used to define hardware implementations. Writing such specification requires a deep knowledge of the principles behind FPGAs and it is a skill that takes years to master. The High Level Synthesis (HLS) advent simplified substantially this task, leaving the programmer to know few details to implement the custom accelerations. HLS hence abstracts the development for FPGA and speeds up the time needed to have an actual working implementation on hardware. Due to their mutable nature, FPGA can be any of the previously introduced architecture (SISD, SIMD, MIMD). The reconfigurable approach to accelerate workloads execution is becoming the de-facto foundation for our world tomorrow's computing.

2.3 Computing Systems

In the previous section we described three families of computational resources commonly used in nowadays systems. What we are going to do in this section is introduce systems that leverage heterogeneity to deliver their performances. We also discuss the features that we chose to exploit to deliver our approach to HSA.

2.3.1 Embedded Systems

This is the category of systems that end-users directly deal with most of the time. We will focus on intrinsic HSA ones, e.g. smart-phones and single-board computers. Raspberry Pi [14] is a board widely diffused. It has been designed with the goals of affordability and low power consumption. On a credit card size PCB it integrates a CPU, a GPU and 512MB of RAM. This system is able, by leveraging its GPU, to deliver reliable performance for Full HD video playback and basic gaming. Embedded systems are characterized by high efficiency and static workload optimization.

2.3.2 High Performance Computing Systems

HPC systems are instead characterized by server class components engineered for performance. Diverse families of processors, like the Intel Xeon and the AMD Opteron, are the usual base for a high performance system. Top notch computing nodes integrate multiple sockets motherboards with multi-core processor, an enormous amount of memory and accelerators for parallel or custom computing. During the last years there has been an increasing trend of building HPC nodes that expose in different combinations of CPUs, GPUs, Many core and FPGA computing resources.

2.3.3 Distributed Systems

A distributed system is the idea of exploiting a multitude of computational nodes to resolve a problem which has been divided in chunks. Supercomputers are basically distributed systems composed by hundreds or thousands of nodes mutually interconnected. For example Tianhe-2, the fastest supercomputer on Earth, is composed by 16,000 nodes with a cumulative core count

of 3,120,000 [11]. This gives the reader some numbers the administrator of those systems have to deal with and the complexity that those imply.

2.4 Our approach

The solution we are going to propose draws from the key features of each category of computing system. In 2013, at the University of Southampton, a cluster of 64 Raspberry Pi [15][16] was built. The idea was to investigate how a distributed system composed by embedded platforms could be used for tasks usually relegated to supercomputers. Power saving techniques used in embedded systems can make a substantial improvement towards efficiency in larger computing systems. Our approach wants to exploit the characteristics of embedded systems, the high performance scenarios of HPC and the scalability of distributed systems by using self-adaptivity to substantially decrease the complexity of managing such systems while targeting computational efficiency.

CHAPTER 3

STATE OF THE ART

Here we present the state of the art in the relevant areas covered by this work that are run-time support for self-adaptation, heterogeneous computational resources management in OSes, adaptive hardware architecture development and monitoring infrastructures. In Section 3.1 we cover the topic of self-adaptive computing systems. Section 3.2 goes through adaptive OSes and run-time support for self-adaptivity. Section 3.3 discuss about decision policies and strategies for self-optimizing computing systems. In Section 3.4 we present the classifications and approaches to the automatic identification of applications hotspots. Finally in Section 3.5 we analyze automated monitoring.

3.1 Self-adaptive computing systems

Today there are several projects working towards the definition and development of self-aware/adaptive systems, able to a certain degree of adapting their behavior and resources to achieve a given goal regardless the continuously changing environmental conditions and demands (e.g. [17], [18], [19]). The common baseline of such efforts and as well as others is to focus on one specific adaptation aspect to optimize an identified goal.

In [20] the authors describe the motivation and background to conduct research in Mobile Autonomic Networks. Hence they put their focus on tackling down power, performance and resource-metering challenges in mobile computing.

In [21] the authors propose a framework for defining metrics to measure adaptivity in cloud computing that leverages benefit-based approaches.

In [22] the authors propose single-ISA heterogeneous multi-core architectures oriented at obtaining performance gains over naive resource assignments. Their approach outperform the best static assignment of multi-core resources.

In [23] PANACEA is presented. PANACEA is a framework based on annotations that have to be inserted at design and coding time for self-healing systems. Specifically these annotations act as an interface for managing, configuring, run-time monitoring and healing of the annotated components.

[24] explores the core building blocks of complex autonomic distributed Internet services by taking in to account agile reorganization and dynamic reconfiguration of network services.

In [25] the authors present a decentralized algorithm for anomaly detection in self-monitoring distributed systems.

[26] presents approaches towards performance and environment monitoring for continuous program optimization in operating systems.

PetaBricks [27], in conclusion to self-adaptive computing systems examples, is a language and compiler for algorithmic choices in dynamic compilation environments.

All the works just introduced show the efforts of the research communities towards specific independent autonomic goals. As a consequence, there is still no integrated idea of self-adaptive system, in terms of an HSA able to autonomously manage its resources and the applications execution towards a user-selected goal.

3.2 Adaptive OS and run-time support for self-adaptability

The research community has seen studies of adaptive OS components becoming more common during the last years.

SEEC [28] is a framework for self-aware computing developed at MIT CSAIL Lab. The idea proposed by SEEC leverages machine learning and control based decisions to optimize a given system towards throughput goals declared by the applications. Although being a very interesting and complete framework, it lacks OS integration and heterogeneous architectures support.

Metronome [29] is a framework to enhance commodity operating systems with self-adaptive capabilities. It extends GNU/Linux with heuristics able to dynamically change at run-time the allocation of CPU cores and CPU time. Metronome downside is the fact that it leverages just one metric to measure throughput performances.

BarbequeRTRM [30], the core of an interesting run-time resource manager, targets multi-core and many-core platforms. Although its management capabilities are remarkable for its context, it assumes that only one task runs on the system and it leverages a static partitioning of workloads. It also provides an orchestrator, the entity in charge of managing the workloads, but unfortunately run-time self-adaptation was not taken in to account.

Shifting back to our focus, HSAs, few approaches have been presented in literature.

PTask [31] is a set of operating system abstractions to support and manage GPUs thanks to a dataflow programming model. By abstracting hardware details, programmers are able to achieve better results and lower latency.

CHIMERA [32], a proposed off-the-shelf hybrid computing solution, builds on top of commercial

components to provide higher computational performances for computational bound problems. It is surely an interesting approach to HSA but it is currently affected by a high development time due to the technology mapping done at design time.

An operating system designed for FPGA-based reconfigurable computers relevant in literature is called BORPH [33]. It runs on the Berkeley Emulation Engine 2 [34] architecture and it is implemented as an extension of the Linux Kernel.

Lastly, K42 [35] is a research OS started in 1996 that heavily leverages adaptation and monitoring. All the approaches we have just gone through are some important achievements and solutions towards heterogeneous computing systems.

By combining the key points of each of those works a new approach able to exploit processors and accelerators bringing higher performances and higher resources utilization, making developers' life easier could be achieved.

3.3 Adaptation policies/strategies

Decision mechanisms are at the heart of self-adaptive and self-optimizing computing systems.

In [36] the authors explore the approach of applying genetic algorithms for dynamic reconfiguration. Specifically they aim at managing dynamic reconfiguration of a data distribution network overlay for remote data mirroring on a collection of servers. The algorithm successfully balances its objective functions goals that consist in minimizing the cost while maximizing network performances and data reliability.

Considering the works that address resource management, most of them focus on single resource

handling while only few tackle multiple interacting resources.

In [37], what the authors do with their approach is to respond to dynamic changes by periodically re-assigning shared system resources among applications. This allows the system to respond at each fixed decision-making interval. For their decision mechanism they exploit the learning results of Artificial Neural Networks (ANN) to which sensor data are given. Given the nature of ANN, their training data can be computed and collected at run-time reflecting the possible changes of the operating conditions.

K42, the research OS previously introduced, lacks the support for decision mechanisms, although all of the requirements for its optimal functioning, i.e. monitoring and adaptation capabilities, are supported.

AQuoSA [38] is an extension of the Linux kernel. This extension enhances the OS by enabling formal decision-making and reasoning through a run-time. A control theory based framework is provided by AQuoSA to allow soft real-time applications to meet quality of service (QoS) requirements specified by the user through adaptive CPU reservation. The main limitation of AQuoSA is that it only supports adaptive CPU reservation. The idea behind AQuoSA and its decision-making capabilities are of valuable interest but most of the common applications are not designed as real-time. AQuoSA also lacks support for HSA.

3.4 Application hotspots and heterogeneous offloading

In this section we are going to introduce various research projects in which is shown that leveraging heterogeneity, e.g. by using FPGAs and GPUs, can have a drastic effect on performance and energy efficiency for application spacing from the embedded system world to high

performance computing.

The approaches to GPU programming explained in [39] led to improvements up to several orders of magnitude. Tangible improvements can be achieved but the task of programming for heterogeneous hardware resources is undoubtedly difficult. The classic case is programming for FPGAs, where different programming models, unfamiliar languages and tools have to be used by the developer to translate computational algorithms or kernels in to custom digital circuits. During the last decades, though, the usage of heterogeneous computing increased outside the academic research community. To tackle the difficult approach of programming for heterogeneous architectures numerous project focused on simplifying the programming models and usage of those computing systems.

The efforts to these problems can be categorized in three main approach areas :

- Domain specific tools : this area focused on domain bounded and domain specific methodologies to leverage the heterogeneity of computing accelerators, in particular the intrinsic flexibility of FPGAs. Finite state machines and digital signal processing based approaches have been a primary interest for researchers in this area. Those have also been brought to the market in commercial products like the Mathwork HDL Coder [40], the Xilinx System Generator [41] and the Synopsys Synplify [42]. The drawback of domain specific methodologies are the limits given by the restriction at design time to a particular domain. Currently, there are efforts to generalize domain specific approaches hence making them available to a broader pool of domains. Two relevant examples are the case of stream processing of [43] and loop-pipelining techniques of [44].

- high level synthesis : Creating FPGA implementations of algorithms or kernels out of high level languages specifications have been a focus of a broad community of researchers since the end of the 1980s. The idea is to let developers and programmers write specifications of digital circuits in a similar and slightly constrained way like they actually do with languages like C++ or Java. Commercial examples are Xilinx Vivado HLS [45] and Altera SDK for OpenCL [46]. Despite the promise of simplification gain due to the use of high level synthesis tools, the adoption of such tools is still not wide due to reasons also argued by the authors in [47]. HLS is surely an interesting, currently available, technology to translate small parts of compute intensive applications in digital circuits on FGPAs implementations. It is not, as today, a complete solution for automatic translation of complex applications.

- Just-in-time binary synthesis : A novel approach that is gaining attention is the idea behind binary synthesis just-in-time generation. Instead of translating as usual a source code to the correspondent hardware implementation, binary synthesis leverages binary programs as specification for hardware accelerators. Examples of such approach are reported in [48] where the authors, leveraging a retargetable compiler, automatically translate software binaries to register transfer level specifications that can be programmed on FPGAs. The authors of [49] survey the key technologies needed to exploit binary synthesis. The just-in-time addition to this novel methodology implies instead a transparent to the programmer application-specific accelerator generation at run-time. The success of

this emerging area would benefit heterogeneous computing by making it accessible and easily usable by a broader pool of end-users.

The promising investigations of binary synthesis methodologies open a spectrum of possibilities for computational offloading to accelerators but they are still in their early stages and they have been used on simple embedded systems hence they are not mature enough yet. Approaches like high-level synthesis and domain specific tools have instead been demonstrated in various research projects.

Two example of seminal works on binary translation are the followings. In [50] the authors demonstrate warp processing's ability to transparently transform binary kernels to FPGA implementations that potentially result in speedups up to 100x over executions on microprocessors. In [51] the idea of designing and implementing a warp processor based on the MicroBlaze [52] architecture is exposed by the authors. They claim average speedups of 5.8x compared to CPU only execution. Both of those works although target very simple applications. Furthermore those target custom SoCs with a dedicated attached FPGA architecture. A major improvement of [53] with respect to [50] is the fact that complex loop structures acceleration has been considered and this leads to a better exploitation of FPGA architectural structure. The exploration of the design space of just-in-time hardware acceleration systems is the focus of the works presented [54] and [55] that put less emphasis on actual practical implementations. Those also stressed more on CPUs with reconfigurable instruction sets.

3.5 Monitoring and control environment at the hardware layer

Techniques and tools employing monitoring probes are already commercially available in providing observability during system development. For instance, Xilinx's Chipscope [56] and ARM's Coresight technology [57], provide real-time system visibility and on-chip debug for reconfigurable or ARM based platforms, respectively.

Researchers also propose similar debugging environments for visualizing a SoC's state at the logical communication level [58].

Ciordas in [59] shows how designers can integrate the monitoring functionality in a Network on Chip (NoC) through a new system design flow, thus offering an integrated solution for system-wide dynamic monitoring covering performance, energy and fault tolerance objectives, while providing a new programming model to communicate efficiently with the monitoring units. Important technological achievements from industry (e.g. EZChip 72-core [60]) and academy (e.g. the multi-core SoC from University of California [61]) embrace dynamic circuit techniques to deal with power consumption and thermal hotspots. However, at present, a variety of dynamic monitoring operations are needed to observe, to reconfigure and optimize software and hardware components at various levels, limiting the opportunity of exploiting the available information.

In conclusion, to achieve the self-adaptive vision, systems must be able to monitor themselves, their environment and make educated decisions about how to alter their behavior.

CHAPTER 4

PROBLEM DEFINITION AND PROPOSED SOLUTION

Considering what has been discussed so far, it is clear that the focus of our work is self-adaptivity for heterogeneous systems. We want to be able to observe the application performances and the hardware status, react to the dynamism of the workloads while optimizing the performance energy trade off. Reacting means making our decision based on the information we have collected and adapt the system accordingly. We have seen in Chapter 3 that many approaches tackle some parts of the problems we are addressing but methods for the optimal use and management of HSAs are not yet available. We will define the problem we are addressing in Section 4.1. Then we will present the basic concepts behind self-adaptivity in Section 4.2. Our self-adaptive component, the Orchestrator, is introduced in Section 4.3. The hardware abstraction that seats aside our component is described in Section 4.4. Finally, our self-adaptive proposed solution is wrapped up in Section 4.5.

4.1 Problem Definition

Workloads of the last decades have been characterized by static workloads and this is reflected by the multitude of static optimization techniques available in the computer science literature, e.g. compiler transformation methodologies. Today the advent of cloud technologies dictated for a dynamic class of workloads that have been initially approached by aggregating multiple resources in clusters of computing systems.

The fallacy of this first solution is that it is impossible to predict the beginning and the end of the tasks, i.e. optimize the resource assignments and configuration to satisfy both the end-user interests (performance) and the system administrators ones (resource usage). Power performance efficiency is also a major concern given the non-negligible impact that computing systems have since their wide-scale deployment in every part of our modern services. Heterogeneous System architectures are increasingly common, think of how spread GPU, MPSoC, Many-core architectures and FPGAs are becoming. The idea of better exploiting the resources available in a HSA means taking advantage of power performances trades off which leads to the optimization we are looking for. By embracing HSA solutions, although, the management complexity of such systems becomes nontrivial.

A solution to this problem is represented by self-adaptivity, the concept of letting a system itself deal with its complexity,the burden of the underlying computational resources and the unpredictability of the dynamic workloads. To do so our system has to be aware of the performances of the task it is executing (self-aware),has to deal with run-time changes, has to leverage methodologies for its decision making and has to have a variety of configurable hardware knobs that those decisions exploit.

4.2 Self-Adaptivity

To leverage the new opportunities presented by the presence of various accelerators in heterogeneous systems,to deal with their complexity and to let our solution cope with dynamic workloads, we embraced and fully relied on self-adaptivity. The idea behind self-adaptivity is straightforward once the context is clear and the right tools to exploit it are provided. We have

developed a component that takes care of information gathering by using a monitoring API, a decision mechanism that behaves accordingly to its selected optimization policy and resource managers to abstract the underlying hardware by providing a simple interface. This component is the Orchestrator, the core of our self-adaptive solution.

4.3 Orchestrator

The Orchestrator is an entity that is in charge of receiving requests from applications to be executed and to choose the optimal and available hardware trying to match constraints issued from each application with system-level constraints. To perform its tedious task the Orchestrator relies on a concept widely used in control systems, the feedback loop. Through the feedback loop, the Orchestrator monitors the status of the system and minimizes the difference between the desired goals and the current system status. The Orchestrator hence has to be made of different parts that work together to make the system evolve towards the optimal desired state. The concept of Observe-Decide-Act loop introduced by IBM with its "Architectural blueprint for autonomic computing" [62] is widely used by the Orchestrator.

4.3.1 ODA Loop

The ODA loop is at the core of our Orchestrator component. Each single phase of the loop is an essential part towards the self-adaptivity of our system. Observation of the applications and their performances is a key task. Based on these findings, the Orchestrator then decides what would lead the system closer to the performances demanded by the applications. Finally the actuation of these decisions takes place and the self-adaptivity through a first iteration of

the ODA loops concretizes. The loop infinitely repeats so as the self-adaptation of our system. Each subsection will detail each of those three phases.

4.3.1.1 Observe

The Observation phase consists in gathering and collecting all the information coming from the system resources and the applications exposing their performances through the monitoring API. This phase is what makes our system self-aware. Our vision of the observable statuses is reported in Table I.

TABLE I: OBSERVABLE STATUSES

Application Status	Throughput Latency Deadline Performance per Watt Energy Consumption
Resource Status	Power Energy Temperature Utilization Frequency & Voltage

All the data contributes to the optimization lead by the feedback loop. The power consumption information coming from the chips can be leveraged for optimizing energy savings while the current performances of the applications are essential for improving Quality of Service (QoS).

The Orchestrator, through observation, has a global vision of the system it is adaptively optimizing.

4.3.1.2 Decide

The Decision phase is what comes after the observation one. It involves the Orchestrator, a set of goals and constraints and a set of run-time management policies. In Table II we report the goals and constraints that our solution, through self-adaptivity, tries to satisfy at each iteration of the ODA loop.

TABLE II: GOALS AND CONSTRAINTS

Application	Goals	Throughput Latency Deadline Performance per Watt Energy Consumption
	Constraints	Throughput Latency Deadline Performance per Watt Energy Consumption
System	Goals	Power Energy Consumption
	Constraints	Power Energy Consumption

Goals and constraints contain the information regarding the goals that the system have to fulfill and the constraints that must be met at both application and system level. A goal

identifies the objective of the system management while a constraint is a form of hard limit that have to be enforced. A run-time management policy is instead an algorithm aimed at optimizing the declared goals by leveraging the information collected during the observation phase and the configurable hardware knobs exposed by the underlying architecture.

4.3.1.3 Act

The last phase carried during the ODA loop is the acting phase. Once the Orchestrator has decided how the system has to adapt to satisfy the running applications, it has to configure the hardware accordingly. This happens through mechanisms like DVFS, that impact heavily the performances and energy consumption of the system, task mapping and resource migration. Acting consists in forwarding the right information containing the configuration to the resource managers, the components responsible for hardware abstraction.

4.4 Resource Managers

Leaving all the tasks to the Orchestrator would have been a burden and would have make our system too complex. While the observation and decisions mechanisms happen within the ODA Loops, the act phase happens on its borderlines as the action has to be carried by the resource managers. A resource manager is the component responsible of the hardware resources. It exposes all the information about its hardware resources (e.g. power consumption, availability), as well as the information about the applications currently running on them. The managers hence represent the actuators of the Orchestrator ODA-Loop.

4.5 Proposed Solution

The goal of this work is to propose an efficient self-adaptivity abstraction for HSAs that focuses on efficiency while lowering the complexity of usage for its end users. This is done by the use of the Orchestrator component, the heart of our self-adaptive approach. The workloads are continuously monitored, tackling down the problem of dynamicity we are addressing. The monitoring information are an essential part for the constraints and goals enforcement that our decision mechanism has to satisfy. Doing so leads to a better exploitation of the underlying hardware architectures which leads our solution closer to the efficiency demanded by our problem context. Hardware vendor specific implementations are not a concern to the end-user thanks to our resources managers that abstract the hardware. This drastically lowers the complexity of dealing with heterogeneous systems. The implementation of each single component of the aggregate that constitutes the Orchestrator is discussed in Chapter 5.

CHAPTER 5

IMPLEMENTATION

The solution we propose is to use self-adaptivity to manage and exploit HSAs. Therefore our purpose is to create a component that makes our idea doable : the Orchestrator. The Orchestrator is the software component responsible of distributing the current workload and assign it to the hardware resources. To be able to do so it needs a monitoring infrastructure, decision strategies, a hardware abstraction layer and a set of possible hardware configuration settings to set. In Section 5.1 we frame the context of the Orchestrator. Section 5.2 is dedicated to the description of the implementation of the monitoring API we used. Section 5.3 covers the decision policies that we have developed. In Section 5.4 we show the mechanisms of the hardware abstraction needed by the Orchestrator. Section 5.5 focuses on the hardware knobs exposed by the hardware abstractions. Object of Section 5.6 is the system manager. In Section 5.7 we describe why our solution is easily expandable for future hardware resources and decision policies. Finally Section 5.8 wraps up all the implemented components to describe the Orchestrator as one component.

5.1 Context Background

Multicore processors, GPUs used for accelerating parallel computations and the exploitation of custom components have set the complexity bar of using computing systems higher and higher. Heterogeneous systems are composed of an increasing number of possibly configurations

furthermore new systems are required to satisfy performance and power efficiency requirements. Expecting an application developer to have a wide system knowledge to leverage the peculiarities of HSAs and produce an application that performs well on a pool of various machines is no longer practical. Programming, operating and managing those systems hence should be properly abstracted. If we provide to the application developers an infrastructure that reliefs them from thinking about the hardware their applications are going to run on, handle ourselves the management of those resources and meet the application performance requirements adaptively, the burden of the problems posed by HSAs would be a detail obscure to them. In this context our solution is going to provide a set of tools and interfaces that are going to mask the underlying complexity of the used computing systems.

5.2 Monitoring

Observing is at the base of any rational decision. If our system knows how it is performing, based on the goals it has to satisfy, it can adapt and better respond to the dynamic workloads it is taking care of. Given that the goal of our solution is to reach an optimal configuration for a HSA, we are going to monitor every application running in our system and all the resources our system is managing. Hardware already expose information about its current usage and power consumption, applications do not. Applications are not usually designed with performance monitoring in mind, they are designed around the task they are required to fulfill. Starting from the assumption that applications have to be treated as black-boxes, we need to address their monitoring problem with a generic solution that can fit this scenario. If we could simply have the application itself exposing some generic information coming out of their black-box

behavior, we could grab that information and use it for our decision mechanisms. This is a nontrivial task that needs a well designed monitoring API that is going to be included in the applications. We chose to use the Heartbeats API, a generic interface for specifying program performances and goals.

5.2.1 Heartbeats API

The Heartbeat Framework [7][63] is what the applications are going to use to communicate their current performances and expose the goals that our system is required to satisfy. Heartbeat can be easily used by programmers given its simple standardized interface. The framework measures the application performance by using its basic measurement unit : the heartbeat. Every time an application reaches a significant point of its execution flow it makes an API call. . Counting these calls, over time, provides the information of how many heartbeats per time frame are being signaled. We can then leverage the intrinsic information of this heartbeat rate and be able to tell how the application is performing. The heartbeats API allows the application to declare its heartbeat rate goal. Our self-adaptive solution will leverage the monitored performance information to meet the goal declared by the application itself. What the programmer has to do to make the application Orchestrator enabled is to include the above mentioned API call. The way that the applications and the Orchestrator are going to share this information is through shared memory.

5.3 Policies

In our Orchestrator component approach, policies define the decision mechanisms that the Orchestrator is going to utilize to adapt the system and meet the application and system targets.

A policy, which is a decision making strategy, can embrace one of the many control solution approaches. The most common ones include heuristic solutions, standard and advanced control based solutions, model-based and model-free machine learning solutions. Heuristics are designed for simplicity and computational performances. They start from a guess about the application's hardware requirements and continuously adjust this estimate. Standard control-based solutions are based on canonical models and they rely usually on Proportional Integral and Derivative (PID) controllers or Petri nets. Advanced control-based solutions require models that are more complex. Some of the most used strategies of this category include Adaptive Control (AC) and Model Predictive Control (MPC). Adaptive Control exploits the idea of having some of its model variables unknown, by doing so, having defined a mechanisms to identify those unknown parameters, the controller is going to adjust those parameters on the fly. In Model Predictive Control instead the controller uses future system reactions predictions to select the next actions that are going to be performed. Model Based Machine learning techniques requires a definition of a context in which the system learns its characteristic behaviors then adjusts its configuration at run-time based on the provided model. Artificial Neural Networks can be used to build a model for our model based controller. Another model-based technique that could be used goes under the name of Genetic Algorithm. Model-free machine learning solutions instead do not require any definition of any model to function properly. Famous examples of this family of solutions are represented by Reinforcement Learning algorithms like SARSA and Q-Learning. We have implemented various policies for our solution that are based on some of the techniques introduced in this section. We are now going to describe in detail our decision mechanisms.

5.3.1 Thread Scaling

A decision mechanism implemented and available in the pool of strategies that the Orchestrator can choose to leverage is called Thread Scaling. This policy can also be usefully exploited to reduce the contention on resources for parallel workloads. This strategy is based on the idea that the performances of applications running in the system can be influenced and controlled by changing the number of thread the application is allowed to use. It decides at run-time the number of threads that an application has to use to execute a given computational kernel. Open Multi Processing (OpenMP) standard approach consists in sharing fairly the available resources among all the running applications. For example, if we have three instances of the same task, those are all going to perform the same. If those applications instead declare each one a different goal, with the standard OpenMP approach, the system cannot satisfy that request. In this situation our solution is able to identify the computational power needed by each application, assigning them dynamically the right number of threads in order to fulfill the set goals.

5.3.2 Heterogeneous Mapping

This strategy covers applications that include multiple computational kernels that can be executed on different hardware resource. Our HSA solution exposes a user-level API that the application designers use to provide some information about the available computational kernels of the application. The programmer specifies the available software implementation for each targeted hardware (e.g. CPU, GPU, FPGA) by registering a function handler for each implementation. By following this simple procedure the Orchestrator will be aware of all the

applications that want to exploit heterogeneous resources and will autonomously manage the resource assignment among them in order to achieve their goals. Upon each execution of the kernel, the system automatically decides on which of the available units (e.g. CPU or GPU) to execute in order to respect the desired throughput. This policy allows the use of the minimum amount of resources needed to execute an application, leading to the possibility to consolidate multiple user applications on the same machine to increase its utilization.

5.3.3 Asymmetric Power

This policy has been developed to deal with asymmetric processors which are resources that expose the same kind of computational unit implemented in two different ways both accessible and exploitable at the same time. An example of this kind of architecture is the ARM big.LITTLE technology [64]. The idea behind this approach is to integrate in a single multicore architecture two families of processors that have different characteristic. Considering a Samsung Exynos 5422 SoC [65], the big.LITTLE approach has been implemented by combining, on a single die, four cores based on the ARM Cortex-A15 [66] architecture with four ARM Cortex-A7 [67] cores for a total of eight heterogeneous cores. The Cortex-A15 cores have been designed to achieve high performance and they are built upon an architecture that consists of a multi-issue out-of-order superscalar pipeline. Cortex-A7 cores instead have been designed around power efficiency and this is reflected by a simpler in-order-issue 8 stage pipeline. Power efficient cores are then labeled as LITTLE, performance oriented cores are instead referred as big cores. Our asymmetric power decision strategy for the Orchestrator takes advantage of those structural differences by leveraging a simple heuristic solution. Based on the possible configuration of the

systems, i.e. the number of cores and the possible frequencies, we build at runtime a table. For the Exynos 5422 SoC on a ODroid XU3 [5] platform , the number of configurations are 1224. This number is given by all the possible configurations of big cores, big cores frequency, little cores, little cores frequency. Each row of this table hence contains the basic information of the number of cores to use and the frequency to be used for each family of computing resources. By calling n_b and n_L respectively the number of big and LITTLE cores that are going to be used, f_b and f_L the selected frequency for big and LITTLE cores, a configuration c can be specified as:

$$c = n_b, f_b, n_L, f_L$$

Additional information are the configuration speedup value and its total power consumption. The expected speedup is constructed by following this procedure. The application is first launched by using the slowest configuration available f_{REF_L}, i.e. the configuration that uses only one LITTLE core at the slowest frequency. Then the application is run at on a single big core at its slowest frequency f_{REF_B}. Performances are then linked by a gain factor K. Each configuration speed up of the table is then generated following this formula :

$$speedup(c) = K \cdot n_b \cdot \frac{f_b}{f_{REF_B}} + n_L \cdot \frac{f_L}{f_{REF_L}}$$

The maximum power consumption $P(c)$ of a given configuration c is instead gong to be computed as

$$
\begin{aligned}
P(c) = (&P_{B\ idle}(N_b - n_b, f_b) \\
&+ P_{B\ utilized}(n_b, f_b)) \\
&+ (P_{L\ idle}(N_L - n_L, f_L) \\
&+ P_{L\ utilized}(n_L, f_L))
\end{aligned}
$$

where N_b is the the total number of big cores,N_L is the the total number of LITTLE cores, n_b and n_L the number of utilized cores, their frequencies f_b and f_L, $P_{\{B|L\}\ idle}$ and $P_{\{B|L\}\ utilized}$ are two functions that return the maximum power consumption under usage or idle states based on those frequencies and number of cores.

Once every row of the table has been generated, they are ordered by speedup. The purpose of this asymmetric power policy is then to find the most power efficient configuration $\hat{c}$ that satisfies the application goal

$$
\begin{aligned}
&\hat{c} =< n_b, f_b, n_L, f_L > \text{such that} \\
&\hat{c} \in \bar{C} = \{c | T_a > G_a, \forall a \in A\} \text{ and} \\
&P(\hat{c}) = \min\ P(\bar{c}), \forall \bar{c} \in \bar{C}
\end{aligned}
$$

where T_a is the current application throughput and G_a is the declared throughput goal.

5.4 Hardware abstraction

A Processing Element manager is the component responsible of the hardware resources. It exposes all the information about its hardware resources, as well as the information about the applications currently running. the managers represent the actuators of the Orchestrator's ODA-loop. These entities are needed as they encapsulate the specific hardware information, and have the knowledge to make the best run-time decision. Without them, this task would be fulfilled by the Orchestrator, resulting in huge complexity and possibly slowing down the decision time. The managers are the components that know and have full access to the HW they control. The mapping between managers and hw resources is not one to one; each manager is in fact in charge of orchestrating a pool of homogeneous resources. In our vision, it is possible to identify three kind of managers (CPU,GPU,FPGA) which can be present in multiple instances depending on the number of clusters of components available on the specific architecture. Due to the different nature of the hardware resources, each manager will be implemented differently and will have access to the specific knobs exposed by the computational cluster. All the managers will internally implement an ODA loop, which is in charge of managing only the applications the Orchestrator will have assigned to them, monitoring their performance and the status of the HW they control with the possibility to tune specific HW knobs. This two layer ODA loop will allow to derive simpler policies for the managers and the Orchestrator and will permit an easy extension and introduction of new managers since the control of HW mechanisms is decoupled from the Orchestrator activity.

5.5 Knobs

In this section we are going to describe the actuators that are used by our policies and that are exposed by the hardware abstraction mechanisms that are implemented through the resource managers. Each decision mechanism we previously described is fed in the information coming from the observation phase and is designed around some of the knobs made available by the managers. Examples are our asymmetric power policy that makes, in its heuristic approach, extensive use of task mapping and DVFS techniques.

5.5.1 Task Mapping

Task Mapping is a technique that assigns an application to a specific CPU core. This capability is very important as it permits to balance the load on multiple CPU systems we are targeting, and it can also be used to modify and control the power drawn by the CPUs. Task mapping functionality is usually provided by the operating system scheduler. For example, the Linux kernel allows to pin a task (process or thread) to a particular set of CPUs through the affinity mask, a per-process structure the scheduler uses to determine on which CPU a task can run. The *sched_setaffinity()* and *sched_getaffinity()* system calls allow respectively to set and get the affinity bitmask of a given process. An higher level utility to change the affinity of a task is *taskset()*.

5.5.2 DVFS

Dynamic voltage and frequency scaling is a frequently utilized power management technique, where the operating clock frequency and voltage of a processor are decreased to allow a substantial reduction in the power consumption. As the power consumption in a CMOS circuit

is proportionally related to fV^2 (where f is the circuit frequency and V is its voltage), a great reduction of power consumption is achieved. Particularly interesting is the case of memory-bound workloads where DVFS can lead to significant reduction in the energy the computation requires. Most operating systems already exploit DVFS as part of their power management architecture. Furthermore, most of them provide interfaces for users or applications to manually control the DVFS operating point of the system processors : for example, under Linux, DVFS related settings are accessible from the user space through the *sysfs* file system and the *cpufrequtils* package provides higher-lever utilities to manage them.

5.6 System Manager

The System Manager acts as an interface in between the system monitors and knobs and the Orchestrator. System monitors (e.g., energy and thermal registers, workload distributions and computational units loads) are used to monitor the performance at the system level providing an overall/general view on the system status. System knobs are elements that can be tuned by specific actions that are not related to just one computational unit, e.g. the possibility of moving a workload or part of it from one processing element to a different one.

5.7 Expandability

What makes our approach scalable and expandable is the possibility of adding new policies hence decision mechanisms as well as new resource managers that are going to provide new knobs to the policies. This means that if a novel architecture has to be integrated in our solution, it can be easily added with no added complexity management cost. Considering how our two layer ODA approach has been implemented, one in the Orchestrator and one in the resource manager,

this will allow to derive simpler policies for the managers and the Orchestrator and will permit an easy extension and introduction of new managers since the control of HW mechanisms is decoupled from the Orchestrator activity.

5.8 Orchestrator Wrap-up

Our orchestrated solution towards a self-adaptive HSA has been described in this section. Observation, which in is done both at the application level and system level, is achieved thanks to the use of the Heartbeats API and the hardware information coming from the resource managers. The decision mechanisms of our Orchestrator then utilize all the collected data. A selected policy is then going to optimize the system resource assignments and its configurations to meet the goals exposed by the application. Our policies include thread scaling, heterogeneous mapping and asymmetric power. The acting phase is enforced by the configuration knobs forwarded by the decision policies to the resource managers. Once the iteration of our ODA loop completes, the cycle repeats and the system adapts continuously to the workload it is handling.

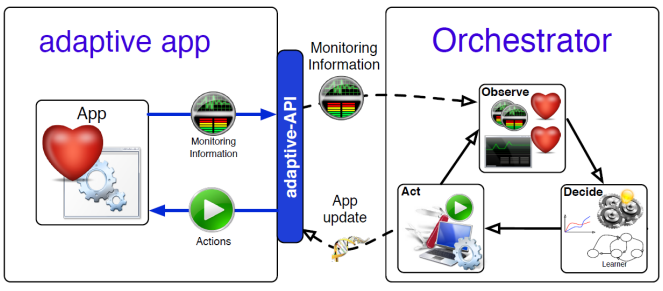


Figure 1: Orchestrator and application interaction abstraction

CHAPTER 6

RESULTS

This chapter goal is to present the experimental results of our work. For doing so we focus on the comparison between the behavior of a normal Linux environment and our self-adaptive approach. First of all we present our configuration settings and testing platforms in Section 6.1. The following sections discuss about the results obtained by using our decision mechanisms built on top of self-awareness and hardware abstractions. In particular, in Section 6.2 we go through the performance impact our solution has on the system. Section 6.3 gives the outcomes of our heterogeneous resource oriented decision mechanism. Section 6.4 shows our thread scaling efficacy for OpenMP applications. Finally Section 6.5 draws the comparison of our power oriented heuristic for heterogeneous CPUs with current available implementations.

6.1 Testing Environments

For our solution testing and results evaluations we considered two different platforms. Both of the platforms run the Linux operating system. The peculiarities and the components that build up our heterogeneous configurations are reported on Table III. The feature that all of our platforms share is DVFS, a technique explicitly exploited by our Asymmetric power policy. The peculiar characteristic of our second platform, the ODroid XU3 [5] development board, is the ARM big.LITTLE [64] heterogeneous CPU architecture which is based on the idea of leveraging different designed processors for better energy efficiency. All of our test case are

based on the Black and Scholes formula, an option pricing application. Being computational intensive, they do not consider bottlenecks due to memory access or bandwidth.

TABLE III: TESTING PLATFORMS

	Platform 1	ODroid XU3
CPU	Intel i5 750	Samsung Exynos 5422
CPU Cores	Dual-Core	Octa-Core
CPU Features	DVFS, HT	DVFS, big.LITTLE
GPU	Nvidia GT240	Mali-T628 MP6
GPU Features	CUDA, OpenCL	OpenCL

6.2 Inter Process Communication Overhead Analysis

One of the key points in developing the Orchestrator consists in realizing a component that does not impact the performances of the system while performing the monitoring, decision and actuation tasks. Having a low impact on the system will give the Orchestator the ability to collect the information needed with a small delay during the observation phase and take decisions in the appropriate moment during the decision phase. The prototype implemented for evaluation the IPC between the different actors in the system makes use of multiple IPC primitives. In particular, we need a way to exchange information between:

- the applications and the Orchestrator during the applications setup phase;
- the applications and the Orchestrator to provide live performance information;

- between the active policies and the mechanism that actuates the policy decision.

These last two situations are the most relevant in term of overall system performances since information between these are exchanged during the execution of the application kernel and a slowdown here will greatly impact the application. The first one, instead, is executed only once per application, thus its overhead is not critical for the overall performance. The prototype uses shared memory as an interprocess communication mechanism to exchange information among the monitors, the Orchestrator, and the actuation layer. The shared memory guarantees the possibility to collect information in short time and to make it seamlessly available to a set of actors in the system. This solution permits to avoid any kind of overhead involved in measurement and actuation that will directly impact the performance of the application. The setup phase is based instead on a message passing interface which is generally slower than shared memory. To check the performance impact of our self-adaptive solution we have measured the overhead due to the monitoring infrastructure we used and the overhead of our decision mechanism. Given that the Orchestrator is involved only in the initial phase that consists of binding the application to the resources, its overhead is negligible. The overhead we have measured is mostly given by the heartbeat monitoring infrastructure and our decision mechanism loop, the ODA loop. In our readings, the monitoring overhead is given by the call to the system that returns the time-stamp needed to compute the heartbeats rate. We computed that *gettimeofday* call average time is $32ns$. The overall time to emit an heartbeat is on average $40ns$. The function that performs the application profiling and chooses which application implementation has to be used also contributes to our system's overheads. Finally

$72ns$ on average is the overhead due to the two function calls that are needed to profile each application. These numbers are almost negligible if we take in account that in our context the computational kernels take at least some milliseconds to execute.

6.3 Heterogeneous Mapping

This section presents a first implementation of the heterogeneous mapping strategy, one of the tasks the Orchestrator takes care of. The system, as explained in the previous chapters, exposes a user-level API that the application designer uses to provide some information about the computational kernels of the application. Using this interface, the designer specifies the available implementations (e.g. CPU, GPU, FPGA) of each computational kernel in the application by registering a function handler for the implementation. The application designer is in charge of registering the different implementations for a given kernel through an API and set the desired goal. Upon each execution of the kernel, the system automatically decides on which of the available units (CPU or GPU in this example) to execute in order to respect the desired throughput. The example of Figure 2, running on test platform 1 of Table III, compares the performances of the application when executing on one resource only, i.e on the CPU (drawn in red) and only on the GPU (drawn in green), with our heterogeneous solution.

If the user sets a goal (indicated by the blue line) then the Orchestrator does its best to meet the specified goal by dynamically changing utilized resource, from CPU to GPU and viceversa. This policy allows the use of the minimum amount of resources needed to execute an applications, leading to the possibility to consolidate multiple user applications on the same machine to increase its utilization. Figure 3 shows two applications being executed on the same

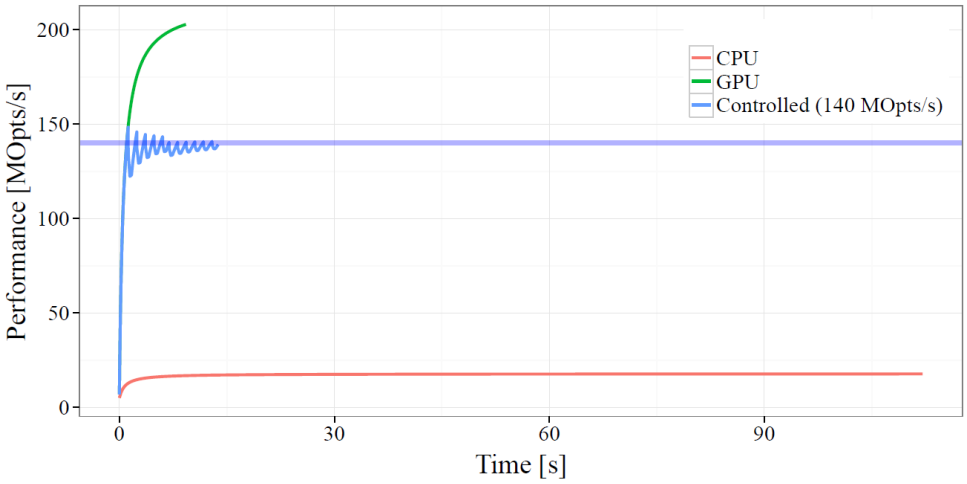


Figure 2: Heterogenous Mapping on a single application

machine, belonging to different users who set different goals. At run-time the Orchestrator allocates the CPU and GPU to the applications in order to meet their goals. The applications set two different performance goals of 40 and 120 (colored bands in the figure). The measured performances, averaged, are represented by the continuous strokes while the objective functions are the goals expressed by each application. As the Figure 3 shows both applications end meeting their goals. In this case the Orchestrator reduces the contention on the GPU allowing an applications to execute on it only when it needs to increase its throughput. Note that as opposed to Figure 2 the performance of the controlled applications are not smooth when they are co-located. This is due both the fact that with the performance goal set they both need to access the GPU and also the fact that when executing on the CPU they both use OpenMP to parallelize the execution on the available cores.

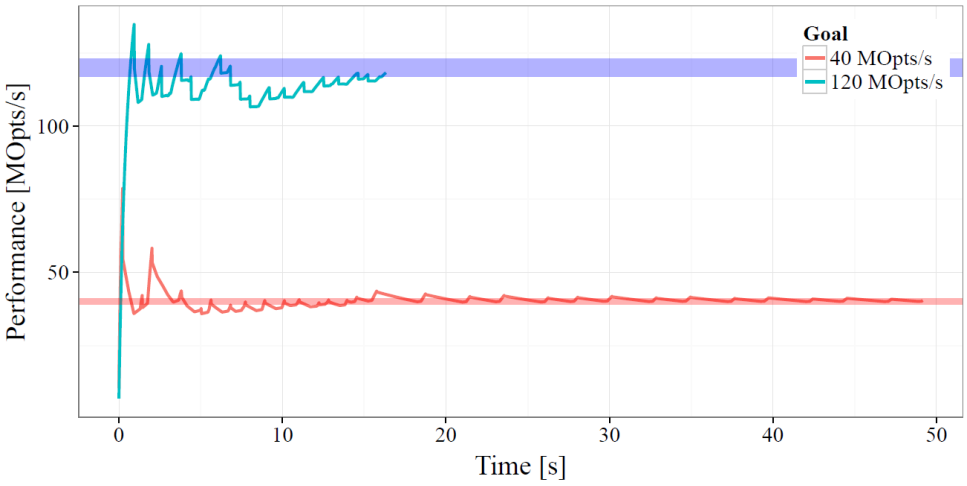


Figure 3: Heterogeneous Mapping on two applications with two different goals

If used unconstrained, OpenMP supposes that only one multi-threaded application is running, generating a number of threads equal to the number of available cores, leading to an high contention when multiple applications are co-located. The developed allocation mechanism is then not only able to allocate CPU and GPU to meet the performance, but is able to do that in situations where there is a high contention on CPU resources.

6.4 Thread Scaling

This section illustrates a prototype of policy that is in charge of dynamically change the number of threads the applications use to execute. This example is a simple prototype which collects information about application performance and takes actions to allocate the available resources to meet applications requirements. At the moment no user or kernel space actuators are available for changing the number of threads an application can use. For this reason we

developed a user space library that the application can link against, and that provides the application the possibility to be instructed with the number of threads to use; the application will then spawn the correct number of threads.

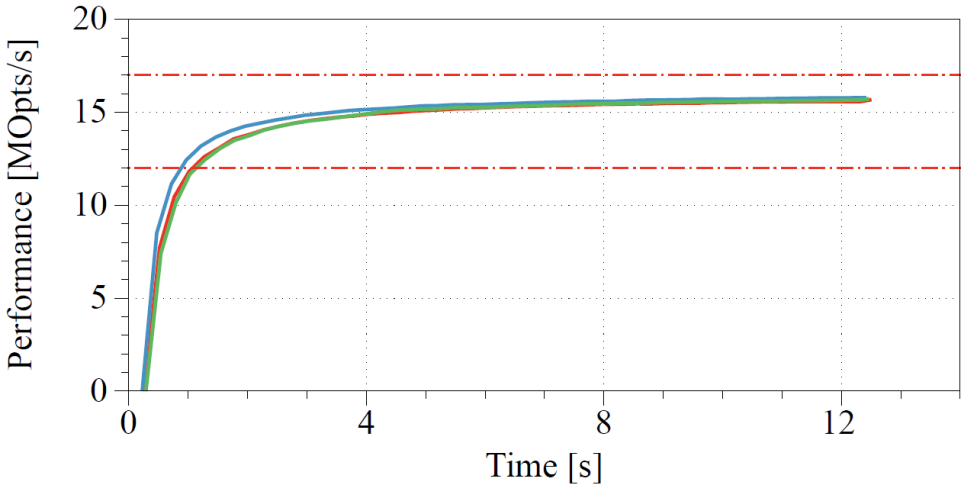


Figure 4: OpenMP standard behavior

Figure 4 shows the behavior of a workload composed of three instances of the Black and Scholes benchmark parallelized using OpenMP when executed on the same machine at the same time. The users' goals are 18 MOpts/s for two of them and 12MOpts/s for the third one. The OpenMP default behavior consists in executing the applications fairly sharing the available resources, thus all the three instances end up with an average throughput of 15 MOpts/s. This means that two applications missed their goals.

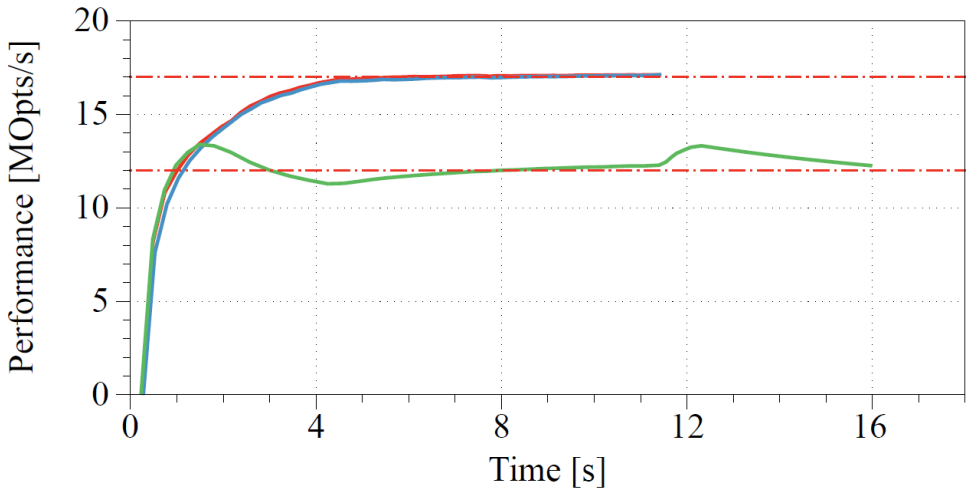
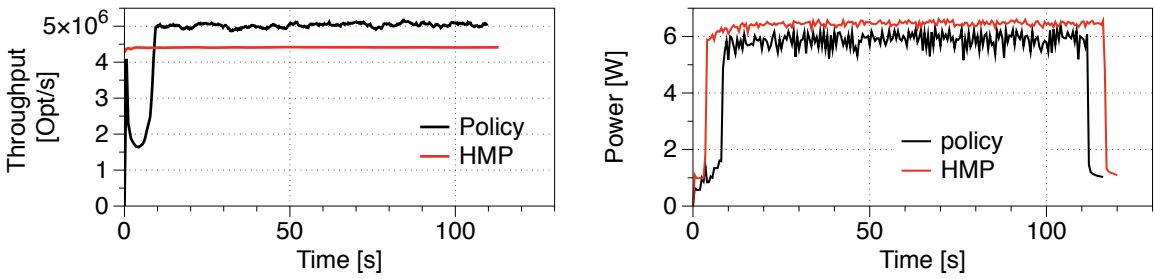


Figure 5: Thread Scaling controlled behavior

Figure 5 shows the effect of the policy implemented to manage the thread scaling of OpenMP applications. In this situation, the system during the execution of the test is able to identify the computational power needed by each application, and consequently instructs them to execute with the correct number of threads in order to fulfill the desired goals. This simple example is also meant to validate the structure of the Orchestrator and its ability to monitor the system while taking informed decisions to optimize application performance.

6.5 Asymmetric Power

This section illustrates our power oriented prototype of policy that addresses heterogeneous CPUs. Our heuristic has been tested on the ODroid platform reported on Table III. Our solution has been compared to the Heterogeneous Multi-Processing (HMP [68]) scheduler that represents the state-of-the-art solution for big.LITTLE architectures. The HMP scheduler is a



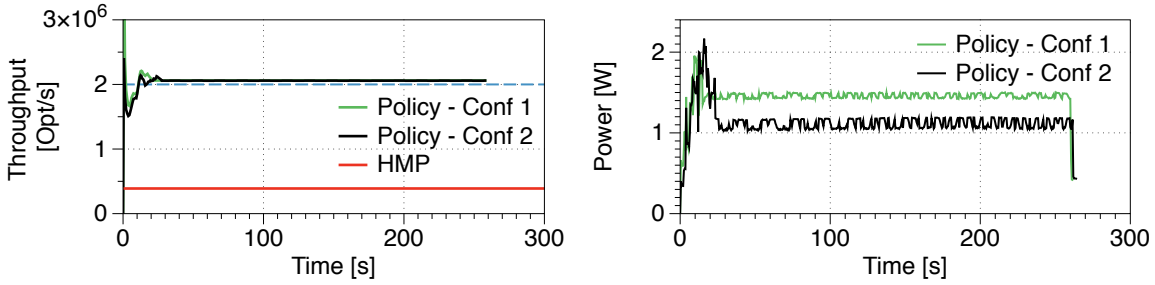
(a) Throughput

(b) Power

Figure 6: Comparison of our heuristic against HMP

variation of the standard Linux scheduler that allocates heavier and computational intensive tasks to the big cores of the processor and lower priority or less intensive tasks to the LITTLE cores. In our first test case we aim at showing the improvements of our strategy over HMP. For doing so we focused on a situation in which the throughput has to be maximized so we disabled DVFS.

Figure 6 shows the results achieved by our policy. We obtained a speedup of about 11% due to the different workload distribution among the different CPU cores and a reduction of peak power consumption of 8% thanks to a better exploitation of the more power efficient design of the Cortex-A7s. Our second test case instead focused on power optimization. We considered two different configurations obtained at run-time by our policy. The first configuration (Conf 1) aimed at satisfying the throughput goal requested by the application, the second configuration (Conf 2) added the power optimization to our decision mechanism by selecting the configuration



(a) Throughput (b) Power

Figure 7: Efficiency of the Asymmetric Power policy

that is estimated to be the most power efficient, as explained in Chapter 5. HMP has been statically set to Conf 2.

Figure 7 shows the results we obtained. The plot of the power consumption underlines the effects of choosing a more power efficient system configuration. Conf 2 lead to a 36% power consumption reduction compared to Conf 1. Additionally, the plot of the throughput shows the beneficial effects thread balancing on the resource used. Due to HMP lack of this ability, the standard approach led to a performance reduction in the order of 5 times, as HMP execution completed at 1400s.

CHAPTER 7

CONCLUSIONS AND FUTURE WORKS

This final chapter of our work draws the conclusions and the considerations of this first implementation of our self-adaptive solution for HSAs. Our Orchestrator successfully addressed the issue of managing heterogeneous systems by lowering the complexity developers had to face. Simple and easily integrable interfaces played a major role for the relevant results of our proposed solution. In Section 7.1 we discuss the contribution of this work as well as its limits. Finally in Section 7.2 we propose some future works that can use our solution as a starting point.

7.1 Contributions and Limits

The main contribution of this work is that we have developed and laid down an approach to decouple hardware from software implementation and let the cumbersome management of heterogeneous resources be part of the past. The behavior of the system in controlling diversified applications running in a multi-programmed environments shows that our Orchestrator and its policies are capable of a straightforward management. What makes our solution effective is its self-awareness. A key point of the research is indeed the use of high level metrics in the monitoring infrastructure: this allows the user to easily express goals on the applications, and the Orchestrator to clearly understand whether those goals are met or not. By exploiting the information coming from the application Heartbeats and the hardware resources our Orches-

trator knows how the system is performing over time and through its decision mechanisms it reconfigures the system and reassigns the tasks to the most suitable computational resource to achieve, with a simplicity known to no other similar solution, tangible computational efficiency to satisfy each application target. The system is well suited for managing parallel workloads and can be deployed on desktop workstations and nodes of a computer cluster to control the execution of applications consolidated on a single node. One of the limits of our solution is the need for the applications to declare their goals. If no goal is declared, our system does not have a way to know if it is performing good or bad with respect to each single application. This will lead to a classic execution of the application, where resources can easily be over-allocated to a single task leading to hardware under-utilization and low efficiency. Another limit is the availability of different implementations for each applications. Considering the case in which only a CPU implementation is available, the efficiency exploitable from HSA would be left dormant. Our solution has abstracted the hardware on which each implementation is going to execute through OpenCL, but actual implementations have to be provided.

7.2 Future Works

Future works considering this work as their starting point should focus on improving and implementing new resource allocation policies By performing a better resource allocation higher power efficiency and resource utilization can be reached. Our approach made use of a-priori applications profiling information. Those same information could be estimated at run time and by doing so better adaptivity and higher performances could be reached. Another relevant research topic is the introduction of run-time power measurements in self-adaptive strategies.

We have implemented a starting approach to this problem with our Asymmetric Power strategy for heterogeneous CPU that utilizes worst case power consumption profiles. Adding real-time power consumption measurements instead of worst case estimations would lead to better results. That specific policy could also be enlarged by adding other families of computational resources, e.g. GPU. The lack of different implementations could be addressed with run-time and just in time compilation systems, as shown to be feasible by different publications (e.g. [69],[70]). Overall the expandibility advertised in this work can be leveraged for integrating easily other emerging architectures, like FPGAs and custom ASIC accelerators.

CITED LITERATURE

1. McMartin, C. and Bohacek, R. S.: Qxp: powerful, rapid computer algorithms for structure-based drug design. Journal of computer-aided molecular design, 11(4):333–344, 1997.

2. Levitt, M. and Warshel, A.: Computer simulation of protein folding. Nature, 253(5494):694–698, 1975.

3. Ko, C.: Computer-aided decision support system for disaster prevention of hillside residents. National Taiwan University of Science and Technology, Taipei, Taiwan, 1999.

4. HSA Foundation. `http://www.hsafoundation.com/`.

5. ODroid XU3. `http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127`.

6. Evans, D.: The internet of things: How the next evolution of the internet is changing everything. CISCO white paper, 1:14, 2011.

7. Hoffmann, H., Eastep, J., Santambrogio, M. D., Miller, J. E., and Agarwal, A.: Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In Proceedings of the 7th international conference on Autonomic computing, pages 79–88. ACM, 2010.

8. IBM Cell Processor. `http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/cellengine/`.

9. Sony PlayStation 3. `https://web.archive.org/web/20080224185451/http://www.us.playstation.com/ps3/about/specs`.

10. Flynn, M. J.: Some computer organizations and their effectiveness. Computers, IEEE Transactions on, 100(9):948–960, 1972.

11. Tianhe-2 Supercomputer. `http://www.top500.org/system/177999`.

12. Nvidia CUDA. `http://www.nvidia.com/object/cuda_home_new.html`.

CITED LITERATURE (continued)

13. Open Computing Language. https://www.khronos.org/opencl/.

14. Raspberry. https://www.raspberrypi.org/.

15. Cox, S. J., Cox, J. T., Boardman, R. P., Johnston, S. J., Scott, M., and Obrien, N. S.: Iridis-pi: a low-cost, compact demonstration cluster. Cluster Computing, 17(2):349–358, 2014.

16. Raspberry Pi Cluster. http://www.southampton.ac.uk/~sjc/raspberrypi/.

17. EPiCS, Engineering Proprioception in Computing Sytems. http://www.epics-project.eu.

18. FlexTiles, Self adaptive heterogeneous manycore based on FlexibleTiles. http://flextiles.eu.

19. AETHER, Self-Adaptive Computing Technologies for Future Embedded and Pervasive Applications. http://www.aether-ist.org.

20. Masters, M. M.: Exploring usability in mobile autonomic networks. In Proceedings of the 10th international conference on Human computer interaction with mobile devices and services, pages 549–550. ACM, 2008.

21. Reinecke, P. and Wolter, K.: Adaptivity metric and performance for restart strategies in web services reliable messaging. In Proceedings of the 7th international workshop on Software and performance, pages 201–212. ACM, 2008.

22. Kumar, R., Tullsen, D. M., Ranganathan, P., Jouppi, N. P., and Farkas, K. I.: Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In ACM SIGARCH Computer Architecture News, volume 32, page 64. IEEE Computer Society, 2004.

23. Breitgand, D., Goldstein, M., Henis, E., Shehory, O., and Weinsberg, Y.: Panacea towards a self-healing development framework. In Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on, pages 169–178. IEEE, 2007.

24. Strassner, J., Kim, S.-S., and Hong, J. W.-K.: The design of an autonomic communication element to manage future internet services. In Management Enabling the Future Internet for Changing Business and New Computing Services, pages 122–132. Springer, 2009.

CITED LITERATURE (continued)

25. Quiroz, A., Gnanasambandam, N., Parashar, M., and Sharma, N.: Robust clustering analysis for the management of self-monitoring distributed systems. Cluster Computing, 12(1):73–85, 2009.

26. Cascaval, C., Duesterwald, E., Sweeney, P. F., and Wisniewski, R. W.: Performance and environment monitoring for continuous program optimization. IBM Journal of Research and Development, 50(2.3):239–248, 2006.

27. Ansel, J., Chan, C., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A., and Amarasinghe, S.: PetaBricks: a language and compiler for algorithmic choice, volume 44. ACM, 2009.

28. Hoffmann, H., Maggio, M., Santambrogio, M. D., Leva, A., and Agarwal, A.: Seec: A framework for self-aware computing. Technical Report MIT-CSAIL-TR-2011-046, MIT CSAIL, November 2011.

29. Sironi, F., Bartolini, D. B., Campanoni, S., Cancare, F., Hoffmann, H., Sciuto, D., and Santambrogio, M. D.: Metronome: operating system level performance management via self-adaptive computing. In Proc. Annual Design Automation Conference, pages 856–865, 2012.

30. Bellasi, P., Massari, G., and Fornaciari, W.: A rtrm proposal for multi/many-core platforms and reconfigurable applications. In Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on, pages 1–8. IEEE, 2012.

31. Rossbach, C. J., Currey, J., Silberstein, M., Ray, B., and Witchel, E.: PTask: operating system abstractions to manage GPUs as compute devices. In Proc. ACM Symposium on Operating Systems Principles, pages 233–248, 2011.

32. Inta, R., Bowman, D. J., and Scott, S. M.: The "Chimera": An Off-The-Shelf CPU/GPGPU/FPGA Hybrid Computing Platform. Int. J. Reconfig. Comp., 2012.

33. So, H. K.-H. and Brodersen, R.: A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph. ACM Trans. Embed. Comput. Syst., 7(2):14:1–14:28, 2008.

34. Chang, C., Wawrzynek, J., and Brodersen, R. W.: BEE2: A High-End Reconfigurable Computing System. IEEE Design and Test of Computers, 22(2):114–125, 2005.

CITED LITERATURE (continued)

35. Krieger, O., Auslander, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Da Silva, D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A., and Uhlig, V.: K42: building a complete operating system. In Proc. ACM SIGOPS/EuroSys European Conference on Computer Systems, pages 133–145, 2006.

36. Ramirez, A. J., Knoester, D. B., Cheng, B. H., and McKinley, P. K.: Applying genetic algorithms to decision making in autonomic computing systems. In Proceedings of the 6th international conference on Autonomic computing, pages 97–106. ACM, 2009.

37. Bitirgen, R., Ipek, E., and Martinez, J. F.: Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture, pages 318–329. IEEE Computer Society, 2008.

38. Palopoli, L., Cucinotta, T., Marzario, L., and Lipari, G.: Aquosaadaptive quality of service architecture. Software: Practice and Experience, 39(1):1–31, 2009.

39. Wen-Mei, W. H.: GPU Computing Gems Emerald Edition. Elsevier, 2011.

40. Mathworks HDL Coder. `http://mathworks.com/products/hdl-coder/`.

41. Xilinx System Generator. `http://www.xilinx.com/products/design-tools/vivado/integration/sysgen.html`.

42. Synopsis Synplify. `http://www.synopsys.com/Tools/Implementation/FPGAImplementation/FPGASynthesis/Pages/SynplifyPro.aspx`.

43. Lindtjorn, O., Clapp, R., Pell, O., Fu, H., Flynn, M., and Mencer, O.: Beyond traditional microprocessors for geoscience high-performance computing applications. Ieee Micro, (2):41–49, 2011.

44. Menotti, R., Cardoso, J. M., Fernandes, M. M., and Marques, E.: Automatic generation of fpga hardware accelerators using a domain specific language. In Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on, pages 457–461. IEEE, 2009.

45. Xilinx Vivado High-Level Synthesis. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

CITED LITERATURE (continued)

46. Altera SDK for OpenCL. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

47. Edwards, S. et al.: The challenges of synthesizing hardware from c-like languages. Design & Test of Computers, IEEE, 23(5):375–386, 2006.

48. Mittal, G., Zaretsky, D., Tang, X., and Banerjee, P.: An overview of a compiler for mapping software binaries to hardware. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 15(11):1177–1190, 2007.

49. Stitt, G. and Vahid, F.: Binary synthesis. ACM Transactions on Design Automation of Electronic Systems (TODAES), 12(3):34, 2007.

50. Vahid, F., Stitt, G., and Lysecky, R.: Warp processing: Dynamic translation of binaries to fpga circuits. Computer, (7):40–46, 2008.

51. Lysecky, R. and Vahid, F.: Design and implementation of a microblaze-based warp processor. ACM Transactions on Embedded Computing Systems (TECS), 8(3):22, 2009.

52. Xilinx MicroBlaze Soft Processor Core. http://www.xilinx.com/products/design-tools/microblaze.html.

53. Bispo, J. and Cardoso, J. M.: On identifying and optimizing instruction sequences for dynamic compilation. In Field-Programmable Technology (FPT), 2010 International Conference on, pages 437–440. IEEE, 2010.

54. Clark, N., Blome, J., Chu, M., Mahlke, S., Biles, S., and Flautner, K.: An architecture framework for transparent instruction set customization in embedded processors. In ACM SIGARCH Computer Architecture News, volume 33, pages 272–283. IEEE Computer Society, 2005.

55. Clark, N., Hormati, A., and Mahlke, S.: Veal: Virtualized execution accelerator for loops. In Computer Architecture, 2008. ISCA'08. 35th International Symposium on, pages 389–400. IEEE, 2008.

56. Xilinx ChipScope Pro. http://www.xilinx.com/tools/cspro.htm.

57. ARM CoreSight. http://www.arm.com/products/system-ip/debug-trace/.

CITED LITERATURE (continued)

58. Goossens, K., Vermeulen, B., and Nejad, A. B.: A high-level debug environment for communication-centric debug. In Proceedings of the Conference on Design, Automation and Test in Europe, pages 202–207. European Design and Automation Association, 2009.

59. Ciordas, C., Hansson, A., Goossens, K., and Basten, T.: A monitoring-aware network-on-chip design flow. Journal of Systems Architecture, 54(3):397–410, 2008.

60. EZChip TILE-Gx. http://www.tilera.com.

61. Truong, D. N., Cheng, W. H., Mohsenin, T., Yu, Z., Jacobson, A. T., Landge, G., Meeuwsen, M. J., Watnik, C., Tran, A. T., Xiao, Z., et al.: A 167-processor computational platform in 65 nm cmos. Solid-State Circuits, IEEE Journal of, 44(4):1130–1144, 2009.

62. Kephart, J., Kephart, J., Chess, D., Boutilier, C., Das, R., Kephart, J. O., and Walsh, W. E.: An architectural blueprint for autonomic computing. IEEE internet computing, 18(21), 2007.

63. Maggio, M., Hoffmann, H., Santambrogio, M. D., Agarwal, A., and Leva, A.: Controlling software applications via resource allocation within the heartbeats framework. In Decision and Control (CDC), 2010 49th IEEE Conference on, pages 3736–3741. IEEE, 2010.

64. ARM big.LITTLE technology. https://www.arm.com/products/processors/technologies/biglittleprocessing.php.

65. Samsung Exynos 5422. http://www.samsung.com/global/business/semiconductor/minisite/Exynos/w/solution.html#?v=octa_5422.

66. ARM Cortex-A15. http://www.arm.com/products/processors/cortex-a/cortex-a15.php.

67. ARM Cortex-A7. http://www.arm.com/products/processors/cortex-a/cortex-a7.php.

68. Chung, H., Kang, M., and Cho, H.-D.: Heterogeneous multi-processing solution of exynos 5 octa with arm® big. little technology.

CITED LITERATURE (continued)

69. Vaz, G., Riebler, H., Kenter, T., and Plessl, C.: Deferring accelerator offloading decisions to application runtime. In ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on, pages 1–8. IEEE, 2014.

70. Damschen, M. and Plessl, C.: Easy-to-use on-the-fly binary program acceleration on many-cores. arXiv preprint arXiv:1412.3906, 2014.

VITA

NAME	Ettore Maria Giuseppe Trainiti
EDUCATION	
	Bachelor of Science in Computer Engineering, September 2013
	Politecnico di Milano, Milan, Italy
EXCHANGE EXPERIENCES	
2015	Visiting Research Student
	Universitat Paderborn, Paderborn, Germany
2014	Master Student
	University of Illinois at Chicago, Chicago, IL, USA
2013	Athens Programme
	Technische Universitat Munchen, Munich, Germany
2009-2010	PoliTong Sino-Italian Campus
	Tongji University, Shanghai, China
LANGUAGE SKILLS	
Italian	Native speaker
English	Full working proficiency
	2011 - TOEFL examination (101/120)