Politecnico di Milano
Facoltà di Ingegneria Industriale e dell'Informazione
LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Workload-Aware Power Optimization Strategy For Heterogeneous Systems

Master of Science thesis of:
Emanuele Del Sozzo
Matricola 795650

Advisor:
Ing. Marco D. Santambrogio

Co-advisors:
Dott. Ing. Gianluca C. Durelli
Prof. Massimo Violante

Abstract

Because of the reaching of the power wall and the consequent end of the Moore's law, over the past decade the trend in microprocessor design has shifted towards parallel architectures, such as multi/many-cores and Heterogeneous System Architecture (HSA) solutions to boost performance and reduce power consumption. Heterogeneity can drastically help in improving energy efficiency although it has the drawback of increasing system management complexity. Different processing units, like Central Processing Units (CPUs), Graphic Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), combined together may provide different solutions and trade-offs at both performance and power/energy consumption ends. On the other hand, actual applications workloads have become more flexible and dynamic. As well, they present different requirements to be satisfied, like throughput, power and energy consumption and so on. This scenario requires the usage of an architecture that is capable of respecting all the application requirements simultaneously. HSAs result to be the most suitable systems for this task, since they are able to both provide high performance and, at the same time, reducing power/energy consumption.

One of the most interesting kind of HSA are *asymmetric multiprocessor systems*. Such architecture has the advantage of providing different throughput or power performance according the running task. Indeed, they are composed of different computing clusters that share the same Instruction Set Architecture (ISA) but have different micro-architectures; hence different clusters target different goals. Therefore a combination of such computing units may be employed to accomplish various requirements.

In this thesis we proposed a workload-aware run-time resource management policy, designed for asymmetric systems, that as the double goal of ensuring the desired Quality of Service (QoS) of running applications while optimizing the system power consumption. Such policy combines Dynamic Voltage and Frequency Scaling (DVFS) (one of the most explored approaches for energy efficiency, both in embedded and large scale systems) and task allocation techniques. The results proved that the proposed policy is able to achieve higher performance, in terms of throughput and power efficiency, than the state of the art heterogeneous scheduler designed for such architecture.

Sommario

Dato il raggiungimento del *power wall* e la conseguente fine della legge di Moore, nell'ultima decade la tendenza nella progettazione dei microprocessori si è spostata verso le architetture parallele, come soluzioni *multi/many-cores* e architetture di sistemi eterogenei (HSA) per incrementare le prestazioni e ridurre i consumi di potenza. L'eterogeneità può drasticamente aiutare a migliorare l'efficienza energetica nonostante abbia lo svantaggio di una maggiore complessità nella gestione del sistema. Differenti unità di processamento, come *Central Processing Unit (CPU)*, *Graphic Processing Unit (GPU)* e *Field Programmable Gate Array (FPGA)*, combinate insieme possono fornire differenti soluzioni e compromessi ai fini sia delle prestazioni che del consumo di potenza/energia. D'altro canto, i carichi delle attuali applicazioni sono diventati più flessibili e dinamici. Allo stesso modo, questi presentano differenti requisiti da soddisfare, come *throughput*, consumo di potenza ed energia e così via. Questo scenario richiede l'utilizzo di una architettura che sia capace di rispettare tutti i requisiti dell'applicazione contemporaneamente. HSA risultano essere i sistemi più adatti a questo compito, dato che sono in capaci sia di fornire alte prestazioni e, allo stesso tempo, di ridurre il consumo di potenza/energia.
Una delle più interessanti tipologie di HSA sono i sistemi di multiprocessori asincroni. Tale architettura ha il vantaggio di fornire differenti prestazioni, in termini di *throughput* o potenza, in base al *task* in esecuzione. Infatti, questi sono composti da differenti *cluster* computazionali che condividono lo stesso *Instruction Set Architecture (ISA)* ma hanno differenti micro-architetture; quindi *cluster* differenti si concentrano su obiettivi differenti. Di conseguenza, una combinazione di tali unità di computazione può essere impiegata per soddisfare vari requisiti.
In questa tesi proponiamo una politica di gestione delle risorse a *run-time* conscia del carico delle applicazioni, progettata per sistemi asincroni, che ha il doppio scopo di garantire la desiderata *Quality of Service (QoS)* delle applicazioni in esecuzione mentre ottimizza il consumo di potenza del sistema. Tale politica combina tecniche di *Dynamic Voltage and Frequency Scaling (DVFS)* (uno degli approcci più esplorati per l'efficienza energetica, sia in sistemi embedded che sistemi su larga scala) e di allocazione di *task*. I risultati dimostrano che la politica proposta è in grado di

ottenere prestazioni più alte, in termini di *throughput* e consumo di potenza, dell'attuale *scheduler* eterogeneo progettato per questa architettura.
Questo lavoro è organizzato come segue:

- il Capitolo 1 dà una visione di insieme del contesto di questo lavoro e brevemente introduce la soluzione proposta;
- il Capitolo 2 fornisce le conoscenze necessarie, descrivendo le caratteristiche della nostra architettura di riferimento e gli strumenti principali che useremo in questo lavoro;
- il Capitolo 3 analizza le soluzioni dello stato dell'arte per l'ottimizzazione di potenza/energia e il soddisfacimento dell'obiettivo di prestazioni nelle HSA; inoltre espone un'analisi dello *scheduler* eterogeneo disponibile sulla nostra architettura di riferimento;
- il Capitolo 4 descrive il problema che vogliamo affrontare e presenta la nostra soluzione;
- il Capitolo 5 fornisce i dettagli dell'implementazione della nostra politica;
- il Capitolo 6 riporta i risultati dei test effettuati sulla nostra politica;
- il Capitolo 7 discute i risultati e le limitazioni del nostro lavoro, e descrive i possibili lavori futuri.

Contents

List of Tables

List of Figures

Introduction 1

This chapter provides an introduction to our thesis. In particular, we describe the main component of the context of this work, and then we briefly present our proposal.
In Section 1.1 we describe the context of this work, while Section 1.2 introduces the architecture we are going to target. Section 1.3 gives an overview of our proposed solution, and, finally, Section 1.4 outlines the structure of this thesis.

1.1 Context Definition

Thanks to the advancements in technology and methodologies, we are now living in an age where computing performance plays a key role in many fields, ranging from finance to cutting-edge research. For decades performance improvements have been achieved by increasing the operating frequency of the used processing units, and users benefited from it without impact on the complexity of their programs. Such enhancements in computing systems were enabled by improvements in transistor technologies according to the Moore's law [1] and the Dennard's scaling law [2]; indeed, every year reduction of MOS transistors size implied more transistors to fit in the same area of the integrated circuit, whereas the power density remained roughly constant. Figure 1.1 shows the transistor count trend in Intel microprocessors from 1971 to 2015 [3]; we can notice that, in almost 45 years, the number of transistors moved from 2300 (Intel 4004 processor [4], 1971) to 5.56 billion (18-core Xeon Haswell-E5 [5], 2014). However, this trend is not true anymore; indeed,

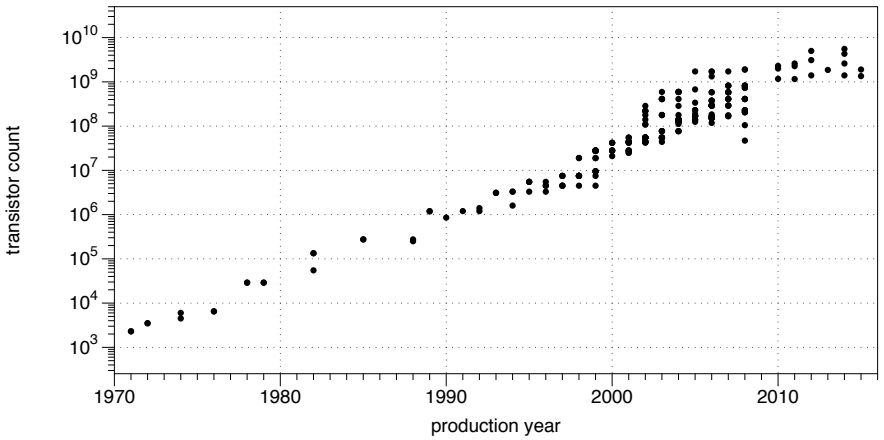


Figure 1.1: Transistor count in Intel microprocessors

(logarithmic scale)

the density of the transistors, in conjunction with higher frequencies, makes it unfeasible to dissipate the high thermal power generated from such small surfaces [6, 7]. Hence, academy and companies started to look for new approaches to deal with power wall limit. As a result, to cope with the performance demand, companies adopted multi-cores and parallel solutions [8, 9, 10, 11]. This choice shifted the burden of performance improvement to programmers, who have to dive into complexity to achieve the performance they need.

Nowadays, Multi-cores systems are available in different forms according to costumers' needs. Indeed, it is easy to find quad-core processors on consumer electronics (like Intel i5 [12] and i7 [13] series or AMD A-Series [14]) and 16-cores on enterprise class server nodes (for instance, Intel Xeon family [15], IBM Power Systems [16], and Oracle SPARC Systems [17]), and, given the number of devices spread around the world, it is straightforward to estimate the impact on the global energy consumption scale. To mitigate this problem, various techniques have been developed, such as clock gating, Dynamic Voltage and Frequency Scaling (DVFS) and power state switching. More recently, a promising approach to pursue performance while keeping energy consumption under control is the exploitation of a Heterogeneous System Architecture (HSA) [18]. An

HSA combines different kinds of processing units offering different performance/power consumption trade-offs, like Central Processing Units (CPUs), Graphic Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), in a single system, to enable the execution of the applications on the most appropriate/convenient kind of resource. CPUs can efficiently run generic tasks, GPUs are optimal for massively parallel repetitive tasks, and FPGAs can be (dynamically) configured to provide a hardware implementation of a software description for an efficient execution. Hence, a system aware of its underlying architecture and the tasks characteristics and needs can exploit the predominant feature of its specific resources. Nowadays available techniques can also be leveraged to improve the efficiency of every single used computational resource and by doing so the overall system benefits of an even better energy efficiency.
In the context of High Performance Computing (HPC) systems, the application workloads have been changing through the years [19, 20, 21]; indeed, while workload used to be static and high-performance oriented, the actual scenario is typically composed of various and flexible on-demand computing workloads. In addiction, such new workloads have different requirements as well, in terms of throughput, power and energy consumption and so on. This situation requires a different approach to face actual workloads. Again, HSAs are to the most convenient solution to tackle such problem; indeed, thanks to their nature, HSAs may be used to satisfy different workload goals, in terms of both performance and power/energy consumption. For these reasons, HSAs currently in use also in HPC systems; indeed, the top supercomputer in July 2015, according TOP500 list [22], is **Tianhe-2**, a system that features both Intel Xeon E5-2692 and Intel Xeon Phi 31S1P coprocessor [23], while other supercomputers, like **Titan** and **Piz Daint**, are accelerated by NVIDIA GPUs. On the other hand, HSAs appear also in July 2015 Green500 list [24], the ranking of the most energy-efficient supercomputers in the world. HSAs dominate the top places of the Green 500 list; indeed, the most energy-efficient supercomputer is **Shoubu**, a heterogeneous system composed of Intel Haswell CPUs [25] and PEZY-SC [26] many-core accelerators, which also feature second and third ranked supercomputers in this list.

1.2 Asymmetric Multiprocessing

An interesting kind of HSA are the *asymmetric multiprocessor systems*, i.e. systems composed of processors with different features. The ARM **big.LITTLE** technology [27] is an example of asymmetric multiprocessor. This solution is mainly employed in embedded systems and mobile devices, thanks to its power and energy efficient nature. Indeed, many mobile devices are powered by ARM big.LITTLE technology; for instance, Samsung implemented such solution inside their Exynos System on Chip (SoC) for Samsung Galaxy S4 [28], S5 [29], and S6 [30] device, as well as Qualcomm for Snapdragon SoC (LG G4 [31], HTC One M9 [32]).
ARM big.LITTLE technology is composed of two clusters of processors: a *big* cluster and a *LITTLE* cluster. Although both clusters share the same Instruction Set Architecture (ISA), their pipelines are different in terms of length and complexity, which means different power and performance levels. As result, big cluster is able of providing high performance, in exchange for higher power consumption, whereas LITTLE cluster is designed to be power efficient, delivering lower performance. Therefore, ARM big.LITTLE design allows it to satisfy two fundamental requirements: high compute capability and very low power consumption. In our vision, we want to tackle the problem of managing and optimizing the power consumption of the ARM big.LITTLE architecture.

1.3 Thesis goal

In a system where one or more applications are running, like in the HPC context, it is crucial to guarantee the Quality of Service (QoS) of each application, in particular when their workloads are various and dynamic rather than static. On the ARM big.LITTLE, as well as on other HSAs, there are no procedures that control and ensure such requirements. For this reason, our thesis aims at introducing a software policy for ARM big.LITTLE architecture, in order to both monitor and satisfy QoS requirements, and, at the same time, optimize the system power consumption, exploiting ARM big.LITTLE features.
To this end, we propose a workload-aware run-time resource manage-

ment policy that exploits DVFS (combined with dynamic core assignment) and task allocation (at thread level) that:

- manages power consumption;
- guarantees performance constraints of the running applications;
- is able to adapt itself to varying system load by exploiting online performance measurements;
- balances the workload among the available heterogeneous cores.

At first, the proposed policy has been developed in high-level simulator for fast resource management policy prototyping. Such simulator, developed in the context of EU Self-Adaptive Virtualisation-Aware High-Performance/Low-Energy Heterogeneous System Architectures (SAVE) Project [33], is available online at [34]. Once the policy development and testing on SAVE Virtual Platform was completed, it has been implemented and evaluated on a real development board, the Odroid XU3 [35], which is powered by the ARM big.LITTLE architecture. Finally, the policy results were compared with the ones provided by Heterogeneous Multi-Processing (HMP), the state of the art heterogeneous scheduler available on the Odroid XU3 development board.

1.4 Thesis Organization

The work presented in our thesis is organized as follows:

- Chapter 2 explains the necessary background knowledge to understand this work; in particular it introduces the HSAs, focusing on the ARM big.LITTLE, our target architecture, and EU SAVE Project;
- Chapter 3 presents an overview of the state of the art, showing different solutions oriented to both reduce power/energy consumption and guarantee QoS in HSAs; then, it provides an analysis of HMP, the heterogeneous scheduler the ARM big.LITTLE platform features;

- Chapter 4 defines the problem this work wants to tackle and explains the solution we propose, i.e. a workload-aware run-time resource management policy for the ARM big.LITTLE architecture;
- Chapter 5 explains the details of the proposed policy by exposing, step by step, its features and structure;
- Chapter 6 evaluates the results of our policy on both SAVE Virtual Platform and on the Odroid XU3 development board, and compare such results with HMP;
- Chapter 7 presents a general overview of the results of this thesis, analyzes the limitations of our work, and describes possible future works.

Background 2

This chapter exposes the background of this thesis work, and reviews the main tools we are going to employ for such purpose. The chapter, at first, introduces the HSA (Section 2.1), then analyzes the heterogeneous architecture we are targeting (Section 2.2). The two main tools we are going to use in this work are presented in Section 2.3 and Section 2.4. Finally, Section 2.5 sums up the chapter and gives some hints about how this thesis work will be developed.

2.1 Heterogeneous System Architectures

HSAs are becoming increasingly adopted in several scenarios [18, 36, 37]; in particular, they may be remarkably useful in reaching performance efficiency and controlling energy consumption, although it means a greater system management complexity as well. For instance, it is crucial to find the best way to allocate tasks on system resources, based on task features.
An HSA is a single system combining different processing units (CPUs, GPUs and FPGAs), which provide different solutions and trade-offs in terms of performance and power consumption. For instance, FPGAs may dynamically supply a hardware implementation of a software description, while CPUs may execute generic tasks, and, finally, GPUs are more convenient for parallel repetitive tasks. Since the system is conscious of underlying architecture, it can allocate a task, according to its characteristics, to the most suitable resource, and so exploit the resource

nature and features. This results in a more efficient execution at both performance and energy consumption ends.

2.2 ARM big.LITTLE

The HSA we focus on is proposed by ARM [38] and it is known as **big.LITTLE** technology [27, 39, 40, 41]. Such solution was introduced in 2011, and is being used in embedded systems and mobile devices, thanks to the possibility to grant power and energy benefits to the final device. ARM big.LITTLE was designed to address two fundamental requirements:

- high compute capability within thermal bounds (high performance end),
- very low power consumption (low performance end).

Following this vision, ARM big.LITTLE technology is powered by two types of processors, resulting in a heterogeneous processing architecture. In this section, we will describe ARM big.LITTLE architecture and main features.

2.2.1 Architecture

ARM big.LITTLE architecture (illustrated in Figure 2.1) may have different processors configurations. The one we present here is the configuration available on Samsung [42] Exynos 5422 SoC [43], composed by the following subsystems:

- a cluster of ARM Cortex-A15 cores [44], with a shared Level 2 cache,
- a cluster of ARM Cortex-A7 cores [45], with a shared Level 2 cache,
- a Cache Coherent Interconnect (CCI), the ARM CoreLink CCI-400 interconnect Intellectual Property (IP) [46],
- a I/O coherent master,

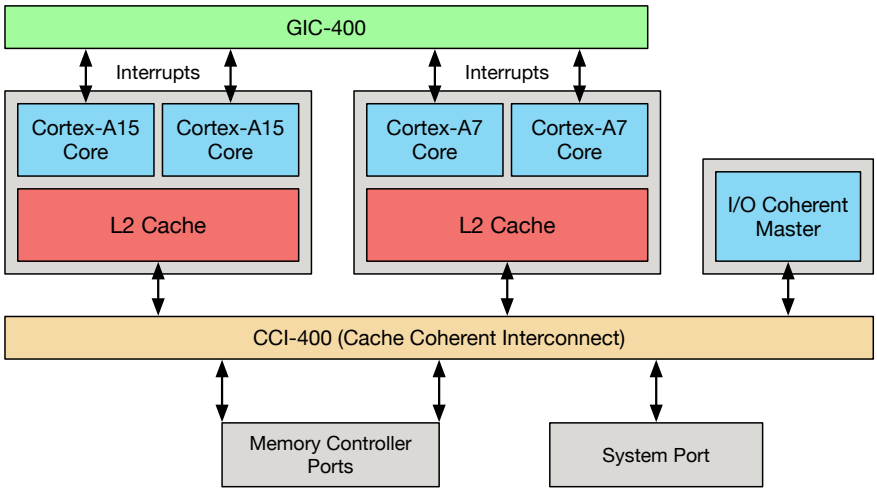


Figure 2.1: big.LITTLE system

- a Generic Interrupt Controller (GIC), the CoreLink GIC-400 [47], which dynamically distributes interrupts to all the cores.

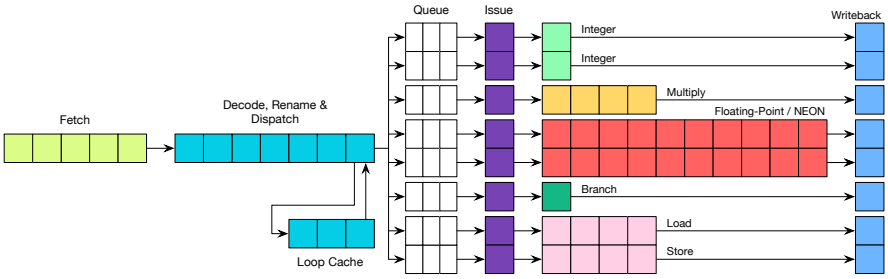
In Section 2.2.1.1 and Section 2.2.1.2 we will describe more in detail the processing clusters architecture and CCI.

2.2.1.1 Cortex-A15 and Cortex-A7 architecture

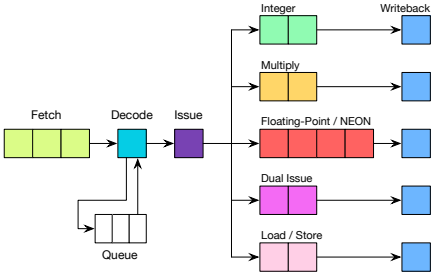
As stated previously, the ARM big.LITTLE architecture we are considering is composed by two processing clusters:

- a *big* cluster consisting of high performance cores (the ARM Cortex-A15),
- a *LITTLE* cluster consisting of low powered cores (the ARM Cortex-A7).

Since both processors support the same ISA, i.e. the ARMv7-A, they are therefore able to handle the same instructions and the same higher-level software applications. On the other hand, their internal micro-architectures are different. In this way, the big.LITTLE design is able to provide different levels of power and performance, and so satisfy the



(a) Cortex-A15 pipeline



(b) Cortex-A7 pipeline

Figure 2.2: big.LITTLE Cortex cores pipelines

requirements previously listed.
Figure 2.2 shows the pipelines of both Cortex-A15 and Cortex-A7 processors. In particular, Cortex-A15 leverages an out-of-order, triple issue processor, with a 15 to 24 stages pipeline, as displayed in Figure 2.2(a). Cortex-A7 is an in-order, non-symmetric dual-issue processor, with a 8 to 10 stages pipeline, as represented in Figure 2.2(b). Different pipeline length and complexity result in different power and performance levels. Indeed, on one hand, the Cortex-A15 delivers high performance, but requires a high power consumption, on the other, the Cortex-A7 is designed to be power efficient, hence its performance are lower.

2.2.1.2 Cache Coherency Interface

The idea behind the ARM big.LITTLE solution is to dynamically instantiate the right task to the right processor. Since different tasks mean

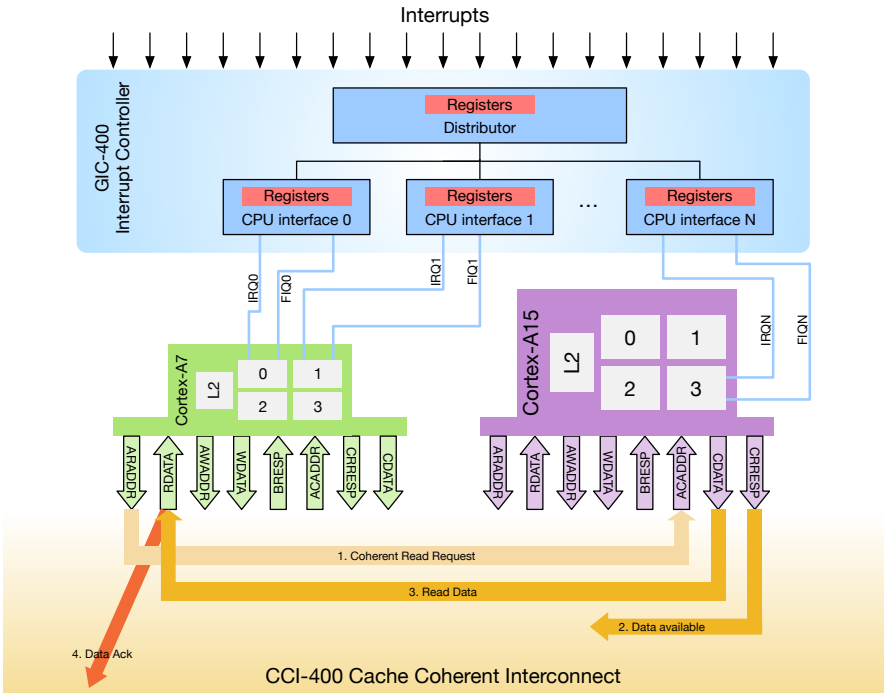


Figure 2.3: Cache Coherency Interface system

different performance and power requirements, it is crucial to allocate them to the most suitable core family. Usually, most of the tasks can be properly executed by one or more Cortex-A7 cores. For instance, in a smartphone context, most of the normal telephony-related functions can be handled by the Cortex-A7 cluster. However, in case of performance hungry tasks, if Cortex-A7 cores cannot satisfy the requirement, then Cortex-A15 cores are turned on. In this way, the task is migrated from the Cortex-A7 to the Cortex-A15 cores, and so it may leverage the big cluster and respect the requirement. Tasks may be re-allocated back to the Cortex-A7 cores, when high performance are no longer needed. This implies that one or more Cortex-A15 cores may be switched off, and, as consequence, that the power consumption is reduced.
It is clear that one of the critical points of the ARM big.LITTLE architecture is the migration time, i.e. the time needed to migrate a task from

one processing cluster to another. If the time required to switch context was too long, the system performance would be noticeable affected. For this reason, an ad-hoc interconnection bus, the CCI, was introduced to transfer data among clusters. This ensures that the task execution is not affected. Indeed, on a device at 1 GHz, the context switching is completed in less than 20,000 clock cycles [39].
CCI relies on the AMBA AXI Coherency Extensions (ACE) protocol [48], which extends AMBA Advanced eXtensible Interface (AXI) protocol and supplies a coherent data transfer at bus level [49]. AMBA ACE protocol requires three more coherency channel, in addiction to AMBI AXI five channels. Figure 2.3 illustrates how CCI system works in a coherent data read from Cortex-A7 cluster to Cortex-A15 cluster. In particular:

1. Cortex-A7 cluster issues a *Coherent Read Request* through its own RADDR channel. CCI is in charge of taking such request to Cortex-A15 cluster ACADDR to snoop into its cache.

2. When Cortex-A15 cluster receives the request, it checks the data availability and returns such information through CRRESP channel. If the data is really available, it is placed on the Cortex-A15 cluster CDATA channel.

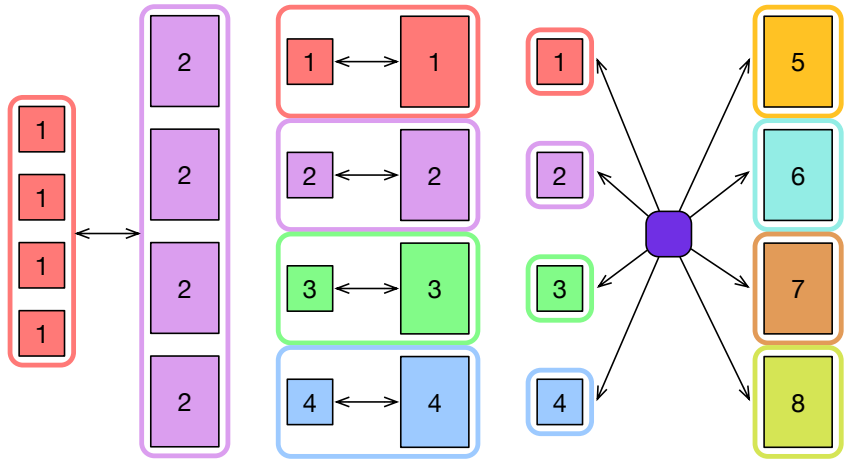
3. Then, CCI transfers the data from Cortex-A15 cluster CDATA channel to Cortex-A7 cluster RDATA channel. This means a cache linefill in Cortex-A7 cluster.

4. Finally, Cortex-A7 cluster returns a *Data Ack* to CCI to state the correct data transfer.

Thanks to AMBA ACE protocol, full coherency between Cortex-A15 and Cortex-A7 cluster is ensured, without external memory transactions.

2.2.2 Schedulers

The three main schedulers used on ARM big.LITTLE architecture are: *Cluster Migration* (Section 2.2.2.1), *CPU Migration* (Section 2.2.2.2) and *Global Task Scheduling (GTS)* (Section 2.2.2.3), as shown in Figure 2.4.



(a) Cluster Migration (b) CPU Migration (c) Global Task Scheduling

Figure 2.4: big.LITTLE schedulers

2.2.2.1 Cluster Migration

Cluster Migration scheduler groups the cores according to their type; therefore, there are two clusters of cores: the big cluster and the LITTLE cluster. This implies that the Operating System (OS) can see one of two processor cluster, instead of all the cores actually available on the big.LITTLE architecture.
Cluster Migration scheduler allows that only one cluster can be active at the time, while the other is powered off. Hence, tasks are allocated on either big cores cluster or LITTLE cores cluster. It is clear that this approach does not scale properly, since, if a CPU intensive task and a light one are running, both must be allocated on the same cluster, while each task should be allocated to the most suitable core, according to the task characteristics.

2.2.2.2 CPU Migration

CPU Migration scheduler requires that the number of big cores is equal to the number of LITTLE cores. The idea is similar to Cluster Migration, but, instead of clustering cores by type (a big and a LITTLE

cluster), here each big core is paired to a LITTLE core, as illustrated in Figure 2.4(b). In this way, the OS does not see the single cores, but the pairs as logical processing units.
In each pair, there is only one active core at the time, whereas the other is switched off. This means that each pair may be either a big or a LITTLE core, and, according to the workload, the most suitable core is dynamically activated by DVFS.

2.2.2.3 Heterogeneous Multi-Processing

ARM GTS implementation, also known as ARM big.LITTLE HMP, does not require an equal number of big and LITTLE cores. Indeed, as displayed in Figure 2.4(c), the cores are no longer grouped in pairs, hence the OS task scheduler sees all the available cores and understands their different computing and power features (big or LITTLE). The scheduler analyzes the performance required by each thread, the current workload on each processor, and, thanks to statistical data and heuristics, is able to allocate the thread to the most suitable core and balance threads between big and LITTLE cores. Like CPU Migration, the unused cores, or a whole cluster, are turned off. However, differently from CPU Migration, the system can deploy all cores, instead of half of them. Moreover, ARM big.LITTLE HMP can isolate intensive threads on big cores and light threads on LITTLE cores. Finally, ARM big.LITTLE HMP targets interrupts to individual cores, while CPU Migration migrates all the context, including interrupts, between big and LITTLE cores.
In order to properly migrate threads from one core to another, ARM big.LITTLE HMP uses a tracked load metric system based on two configurable thresholds: the *up migration threshold* and the *down migration threshold*. For instance, if the average tracked load of a thread running on a LITTLE core surpasses the up migration threshold, then ARM big.LITTLE HMP may decide to move such thread to a big core. On the other hand, when the average tracked load of a thread allocated on a big core falls under the down migration threshold, the thread is may be migrated to a LITTLE core. Therefore, at cluster level, ARM big.LITTLE HMP is responsible for properly allocating and, in case, migrating thread between big and LITTLE clusters. Within clusters, standard Linux scheduler Symmetric Multi-Processing (SMP) balances the

load across the cluster cores. Besides, SMP has been recently updated in Linux 4.3, to make its metric more precise and more representative [50].
The tracked load metric system collects information according to the processor frequency. This means that ARM big.LITTLE HMP and DVFS mechanisms can easily cooperate without problems. Moreover, ARM big.LITTLE HMP task migration is supported by a set of software thread affinity management techniques. In particular:

fork migration: when a new thread is created, there is no tracked load history, hence it is allocated on a big core, so that it can be easily migrated to a LITTLE cores in case of a light workload thread;

wake migration: when a task moves from idle to run state, its tracked load history is analyzed and, usually, the task is assigned to the cluster it used to run on;

forced migration: In case of threads that do not sleep, or not so often, their tracked load history is periodically checked and they are migrated according to the configurable thresholds system;

idle-pull migration: when no task is allocated to a big core, load metrics of tasks running on LITTLE cluster are analyzed and, if a task exceeds the up migration threshold, it is migrated to the idle big core, otherwise it is switched off;

offload migration: when LITTLE cores are idle or under-utilized, threads on big cores are periodically migrated downwards to exploit unused compute capacity, while they still remain eligible for up migration.

2.3 Odroid XU3

The development board used in this thesis work is the **Odroid XU3** [35]. Such board, powered by the ARM big.LITTLE technology, is a new generation of computing device, which includes powerful and energy-efficient hardware and smaller form factor.
The main specifications of Odroid XU3 board are listed in Table 2.1.

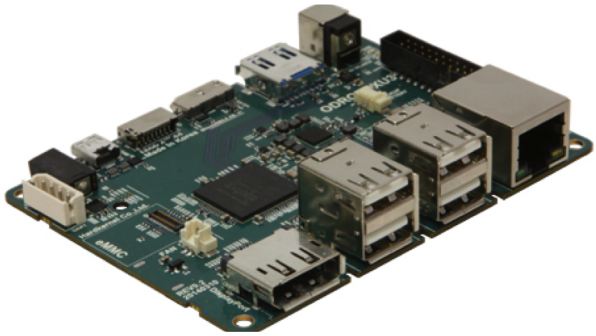


Figure 2.5: Odroid XU3 development board

architecture	Samsung Exynos 5422 SoC
CPUs	ARM Cortex-A15 1.9GHz quad core and ARM Cortex-A7 1.3GHz quad core
GPU	ARM Mali-T628 MP6 [60]
RAM	2Gbyte LPDDR3 at 933MHz (14.9GB/s memory bandwidth), PoP stacked
storage	eMMC5.0 HS400 Flash Storage
interfaces	1 x USB 3.0 Host, 1 x USB 3.0 OTG, 4 x USB 2.0 Host, HDMI 1.4a and DisplayPort 1.1
other	integrated power consumption monitoring tool

Table 2.1: Odroid XU3 specifications

The Odroid XU3 board can run various distributions of Linux, including the Ubuntu OS [51] and the Android OS [52]. In particular, the Linux distribution we used is Lubuntu 14.04 [53].
The Odroid XU3 feature we are interested in the most is HMP scheduler. Indeed, in September 2013, Samsung announced that HMP solution [54, 55, 56] would be implemented for their architectures [57], starting from Exynos 5420 [58]. In previous architectures, like Exynos 5410 [59], Samsung used Cluster Migration scheduler, but it was not the right choice to fully maximize the benefits of ARM big.LITTLE solution. In **??**, we will analyzed HMP behavior when it deals with multi-threaded CPU intensive tasks.

2.4 EU SAVE Project

SAVE is a European collaborative research project [33, 61, 62]. Such project is funded by the Seventh Framework Programme [63], and it aims at developing software/hardware technologies able to efficiently leverage HSAs.
The goal of SAVE project is to develop a system able to autonomously choose the most convenient computing resource (e.g. CPU, GPU or FPGA) where a task has to be executed. This is done by exploiting self-adaptivity and hardware-assisted virtualization. Such system aims at ensuring requirements of actual HPC scenario characterized by various and flexible on-demand computing workloads, rather than static and high-performance oriented as they used to be. SAVE project developers decided to focus on HSAs since they are the most suitable architecture capable of satisfying user-defined optimization goals (like performance, energy, reliability and so on). The developers' vision results in an architecture that is more dynamic and adaptable to workload features, while heads for energy consumption minimization.

2.4.1 SAVE Virtual Platform

The other tool we are going to employ in this thesis work is SAVE Virtual Platform [34, 64], a HSA simulator.
Since heterogeneity implies a high cost in both design and system management complexity, it is critical to define novel and innovative run-time resource management policies capable of, autonomously, distribute workloads onto convenient resources, in order to satisfy tasks Service Level Agreement (SLA), like throughput, power/energy consumption. However, implementation and, especially, validation of such policies is not an easy task. For instance, it requires the availability of the real target platform for the implementation, although this may lead to a policy limited to that particular target platform. Moreover, once the run-time management policy has been implemented, it is necessary to port and test it on other different architectures. Summing up, it is clear that a fast design and prototyping tool for run-time resource management policies on HSAs may be useful. In this way, the designers can concentrate on the policy implementation, without considering secondary issues, and

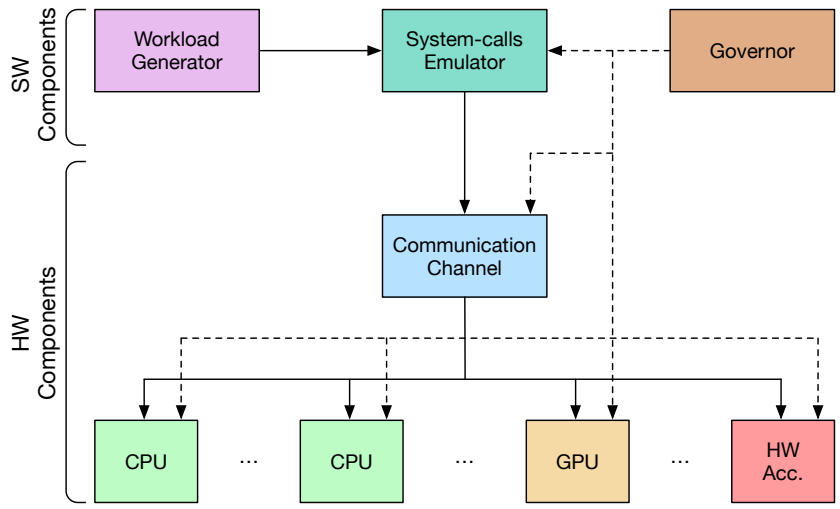


Figure 2.6: Structure of SAVE Virtual Platform

so easily achieve a general and consistent validation of the policy.
SAVE Virtual Platform is a system-level simulation framework implemented in SystemC and Transaction Level Modeling (TLM) [65] (Figure 2.6). Such framework is composed by a set of functional models for the HSA. More in detail, applications are represented as task graphs, where the computational kernel nodes may have different implementations based on the available resources. On the other hand, SystemC modules model the pool of generic resources (e.g. Cortex-A15 CPU, Cortex-A7 CPU, Mali GPU), and, based on cycle-accurate simulators and application execution profiling, their performance and power consumption. Thus, SAVE Virtual Platform allows to configure and simulate a complex homogeneous or heterogeneous system architecture and execute different workloads on the available processing units. Finally, the designer can implement and test run-time resource management policies thanks to the *governor* module, which is in charge of allocating the proper computational units to the applications.

2.5 Summary

In this chapter we introduced the main tools we are going to exploits for this thesis work. In particular, we presented the Odroid XU3 development board and the SAVE Virtual Platform. In the vision of this work, we are going to implement a workload-aware run-time resource management policy for ARM big.LITTLE architecture. Such policy is designed for multi-threaded high CPU intensive tasks, and will take advantage of DVFS techniques in order to both guarantee QoS and, since we are using a heterogeneous architecture, the most power-efficient configuration, where a configuration is a combination of cores (big and LITTLE) number and frequency. Moreover, we will leverage HMP weak points in dealing with multi-threaded applications (as we will discuss in Chapter 3), so to improve our run-time resource management policy performance, like throughput, power and energy consumption. Therefore, our policy is a combination of DVFS techniques and workload analysis in order to properly schedule one or more tasks on the available resources. The policy will be implemented and tested on the SAVE Virtual Platform and then validated on the Odroid XU3 development board.

Related works 3

This chapter both presents the works that contributes to the state of the art in the field, with an emphasis on works about DVFS management on HSA. and provides an analysis of HMP scheduler. The proposal of this thesis is a workload-aware run-time resource management policy designed for HSA, like the ARM big.LITTLE architecture. In particular, our policy aims at satisfying QoS requirements of multi-threaded computational-intensive applications, while reducing power consumption. In order to achieve such result, DVFS technique is employed to set the system in a power efficient configuration suitable for respecting the applications goals. The knowledge of the behavior and scalability of multi-threaded applications may be used to enhance performance, in terms of throughput, power and energy consumption of such applications. For this reason, we ran several tests on the Odroid XU3 in order to profile the behavior of HMP scheduler in case of multi-threaded applications, i.e. the applications we want to target. In this way, we improved our policy and designed it to properly allocate and distribute workload among the available resources. In this way, we avoid to use HMP, the state of the art scheduler for ARM big.LITTLE architecture, since we can achieve better performance at both throughput and power consumption ends.
In Section 3.1 we present the main technique we are going to exploit for this work (DVFS), and we report some works that prove its benefits. In Section 3.2 we analyze works that use DVFS techniques on HSA, in particular on the ARM big.LITTLE, out target architecture. Section 3.3 reports an interesting work that may be similar and related to

the one we propose, but it is still under development. Section 3.4 summarizes the content of this chapter and compare the most interesting works with our policy. Section 3.5 presents the throughput analysis of a multi-threaded application. Section 3.6 is about the first prototype of our solution we found to both balance the workload among the cores and improve performance. Section 3.7 shows two attempts of enhancement for our solution. Finally, Section 3.8 reviews the analysis we performed on HMP scheduler.

3.1 Dynamic Voltage and Frequency Scaling

One of the main techniques for improving energy efficiency in embedded systems is DVFS [66]. DVFS is a technique used to switch the voltage and/or frequency of a system based on performance requirements, like throughput, power or energy consumption, and so on. In CMOS circuits, power scales proportionally to $V^2 f$ (being V the voltage and f the frequency), hence, if we lower frequency, the voltage required to by the circuit can be lowered too, leading to energy saving. For this reason, such technique is used along with power-aware scheduling techniques.
In literature, there exist many works that exploit DVFS techniques, either by simply analyzing system behavior as the frequency and voltage change, or by implementing algorithms that take advantage of such tuning. Moreover, DVFS may be applied to different resources, like CPUs and GPUs. Abe et al. [67] provided a power and performance analysis of systems accelerated with GPUs. The authors demonstrated that the knowledge of the workload characteristics mat be used to proper tune both GPU core and memory clock frequencies in order to improve energy efficiency. The authors reported a 28% reduction of the system energy on a matrix multiplication benchmark by scaling down the memory frequency with a NVIDIA GeForce GTX 480 [68].
In the context of HPC systems, Ge et al. [69] analyzed and compared how DVFS can impact, in terms of performance and energy consumption, on CPU and GPU applications. The authors relied on a power aware heterogeneous system including dual Intel Sandy Bridge CPUs [70] and NVIDIA K20c Kepler GPU [71]. Their work showed that DVFS in CPU and GPU computing behaves similarly in terms of performance

(higher states imply higher performance) and differently in terms of energy efficiency. Moreover, for computational intensive applications with large data sizes, GPUs turned out to require the same amount of energy in almost every DVFS state, hence the highest DVFS state is optimal in terms of performance and energy consumption.
Mei et al. [72] explored the effects of DVFS on GPUs energy consumption with 37 benchmark applications. The authors achieved an average of 19.28% energy reduction, with respect to the default setting, while losing less than 4% in performance. Moreover, the authors proved that core and memory frequency scaling strictly depends on GPU application characteristics.
Let us now focus on works that directly implemented DVFS techniques in their proposed solutions. Spiliopoulos et al. [73] extended *gem5* simulator [74] in order to turn it into a complete hardware-software framework suitable for full-system DVFS. In particular, the authors added the notion of clock and voltage domains, along with their managers, to gem5 simulator, on both hardware and software sides. Moreover, gem5 was extended with a power-estimation framework to evaluate the efficiency of various DVFS policies. Finally, the authors showed that Linux and Android *cpufreq* governors may be easily integrated in their framework.
Deng et al. [75] presented *MultiScale*, a system that, relying on software policies and hardware mechanisms, coordinates DVFS among multiple memory controllers in multi-core server processors. At first, *MultiScale* monitors application bandwidth requirements across memory controllers, then, it applies heuristics to find a frequency combination able to reduce to minimum the overall system energy consumed by memory system. The authors demonstrated that *MultiScale* is particularly efficient in systems with traffic skewing scheduling and allocation policies.
Choi et al. [76] propose a intra-process DVFS policy for non real-time applications executing on embedded systems. Such DVFS policy exploits data about external memory access statistics in order to build a regression model that helps the CPU to choose the proper energy efficient combination of voltage and frequency able to respect soft timing constraints of upcoming workloads. The authors implemented this policy on an XScale-based embedded system and evaluated it with actual hardware measurements on a set of applications. The policy resulted to

save more than 70% of CPU energy (with 12% of performance degradation) in case of memory-bound applications, while 15-60% CPU energy was saved for CPU-bound applications (5-20% of performance penalty). Saewong and Rajkumar [77] proposed four different DVFS schemes aimed at energy saving for embedded systems. Such schemes are designed for different hardware configurations according. For instance, one scheme chooses one single frequency able to complete the task within its deadline and to minimize power consumption. Another scheme works like the previous by it is priority based, hence it may create extra slack by running the other tasks at a frequency higher than the minimum they need.

3.2 DVFS on HSA

In this section, we present works on DVFS that have been developed for HSA.
Kianzad et al. [78] proposed a framework called *CASPER* based on genetic algorithms. Such framework is designed for both homogeneous and heterogeneous architectures and integrates task scheduling and DVFS in order to generate a schedule capable of respecting deadline and minimizing power consumption. Experimental results showed that the proposed framework is able to save averagely about 8% more energy with respect to non-integrated solutions that exploit the same power management techniques. Yang et al. [79] proposed an energy-efficient scheduler to map real-time applications on HSA. Such approach is composed by two phases: the former one is a design time exploration based on Pareto curves that produces a set of schedules, the latter exploits a low complexity scheduler that selects the optimal combination of working points in terms of performance and energy consumption.
Other works focused on our target architecture, the ARM big.LITTLE, and either characterized its performance and power consumption, or developed DVFS and scheduling solutions for such architecture.
Pricopi et al. [80] developed models to estimate both performance and power consumption of workloads on a ARM big.LITTLE architecture. In particular, their models predict the behavior of an application on a target core, given its execution profile on the current core. Moreover,

the proposed model evaluates cache miss and branch misprediction rates on the target core.
Yoo [81] characterized a model for power-performance scaling in ARM big.LITTLE architecture, and empirically validated such model using Dhrystone benchmark.
Imes and Hoffmann [82] investigated energy oriented resource allocation strategies for different embedded platforms where performance constraints have to be met and energy consumption minimized. The authors focused on both homogeneous (Intel Haswell [25]) and single-ISA heterogeneous multi-core systems (ARM big.LITTLE), and found out that different platforms require different resource allocation strategies. In particular, the former requires Race-to-Idle heuristics to achieve energy efficiency, while the latter requires Never-Idle heuristics. Moreover, a wrong strategy could double to energy consumption with respect to the optimal solution. This work demonstrates that, in case of an available QoS, it is a good procedure to provide the minimum throughput required, instead of a full speed solution; hence, a system able to adapt itself to the application goal, as the one we propose, is needed.
Lukefahr et al. [83] proposed *Composite Cores*, an architecture to reduce switching overheads between different cores, and so reducing energy consumption due to core migration. Such architecture pairs LITTLE and big compute μEngines in order to increase both energy efficiency and performance, while a controller handles the μEngine switching in order to both reduce energy consumption, and constrain performance loss to a configurable bound. Differently from our proposed solution, this work does not exploit DVFS techniques and, most important, employs only one μEngine at the time, instead of leveraging all the available resources simultaneously.
Gaspar et al. [84] presented a framework for HSA that performs a real-time control of multi-threaded applications execution to reach their performance goals. Such framework models the underlying architecture, as well as the running applications, in order to achieve better results in terms of both performance and power consumption by scaling the system resource allocation and frequency. This was developed on Odroid XU+E [85] development board featuring ARM big.LITTLE processor, hence the system resource allocation is done relying on Odroid XU+E

cluster migration scheduler. Finally, the authors reported a 49% energy saving, while the relative performance error decreased from 2.801 to 0.168. Like our proposed solution, this work guarantees the QoS, takes advantage of DVFS techniques and distributes the workload in a more convenient and efficient way. However, this work exploits the computing resource at cluster level only, since it relies on cluster migration scheduler.

Muthukaruppan et al. [86] developed a hierarchical power management framework for asymmetric multi-core systems. The work was developed on a Versatile Express development platform [87], powered by ARM big.LITTLE architecture. Such framework, based on control theory, is built as an extension of Linux completely-fair scheduler, and is designed to handle multiple controllers in order to satisfy QoS constraints and minimize power consumption. Also this work is similar to the one we proposed. Nonetheless, this work, like the previous one, is based on cluster migration scheduler, and so it exploits either A15 cluster or A7 cluster, but not both of them.

Kim et al. [88] presented a work on ARM big.LITTLE architecture, and, in order to take advantage of its features, improved Linux kernel load balancing algorithm by including information about processor utilization. In this way, the new scheduler results more energy efficient than the standard one (up to 11.35% more efficient), while the performance are almost not affected. Differently from our work, this one does not implement DVFS techniques or aims at satisfying a performance goal, but focuses on reducing energy consumption. Although it is not based on cluster migration scheduler, it relies on CPU migration scheduler, hence the system couples each big core with a LITTLE core, composing a set of virtual CPUs where only one core of the couple can be used at the time.

Holmback et al. [89] presented a performance monitor based power manager, designed for cluster switched ARM big.LITTLE architectures. In particular, such manager allocation policy is based on application performance instead of workload levels, since workload is not a sufficient metric for big.LITTLE systems. In this way, the system is not forced to Race-to-Idle anymore and can run on the lowest possible clock frequency, without performance reduction. The authors showed that their

power manager is capable of obtaining an impressive energy reduction with a small performance degradation. This work is similar to [84, 86], in terms of content, but does not propose a goal oriented policy, and still exploits cluster migration scheduler.

3.3 Energy-Aware Scheduling

The idea of an Energy-Aware Scheduling (EAS), i.e. a scheduler that takes into account energy information, to be the future way to go [90, 91, 92, 93]. In this way, energy-aware Linux kernel scheduler may be aware of the energy cost and make more intelligent decisions, also thanks to the knowledge of the underlying hardware platform.
Following this vision, the main idea proposed by ARM is a new design including `CPUfreq` [94] decisions, `CPUidle` [95] awareness and ARM big.LITTLE task placement in a completely generic way. The task scheduler goal is to optimize the energy cost corresponding to the different task allocation decisions by taking all possible processor clock frequencies and sleep states into account. In this way, the scheduler may analyze if it should migrate a task from a CPU to another, or leave the task on that CPU and scale frequency instead. In order to achieve such result, it is necessary to build a table containing relative processor computing capability and energy consumption for every possible configuration of different systems.
The proposed solution of course requires important changes in Linux kernel parts, hence its implementation may not be so straight-forward. For this reason, *Linaro* proposed two different tools. The first tool is a workload simulator that allows the users the reproduce application behaviors interacting with the task scheduler. The synthetic workloads generated on such simulator may be tested, measured and shared with other users to guarantee testing uniformity. On the other hand, the second tool is designed to collect statistical information about various processor sleep states. Moreover, the tool logs the changes in processor clock frequency. In this way, if a power model is paired with the scheduler, the tool can provide valuations of the energy consumed by the system in that particular configuration. The idea proposed for the EAS is very similar to the solution proposed in this thesis, therefore, it

work	goal oriented	DVFS	task allocation	resource usage
[83]	✓		✓	virtual CPU
[84]	✓	✓	✓	cluster
[86]	✓	✓	✓	cluster
[88]			✓	virtual CPU
[89]		✓	✓	cluster
our proposal	✓	✓	✓	all

Table 3.1: Relevant works summary

may state (in a theoretical way, at least) the goodness of our approach.

3.4 State Of The Art Summary

In this chapter we have presented some of the most interesting works related to the DVFS in both homogeneous and heterogeneous systems. In particular, after an introduction to DVFS and the benefits of such technique ([67, 69, 72, 73, 75, 76, 77]), we focused on works that are more related to the purpose of this thesis. We analyzed works about DVFS on HSA ([78, 79]), and, specifically, on ARM big.LITTLE architecture. In Chapter 4, we will characterize ARM big.LITTLE system in terms of both performance and power consumption, like [80, 81] already did, and we will use such analysis to build our policy.
It is clear that the novelty of our work with respect to the one we presented is the fact that our policy is able to employ all the available resource on the ARM big.LITTLE architecture. This is a consequence of the HMP scheduler included in our system, which is, as we stated in Chapter 2, the most suitable scheduler for this architecture. Therefore, we will compare the results of our workload-aware run-time resource management policy with HMP scheduler, since it is the only state of the art work that takes advantage of all the available resources on the ARM big.LITTLE platform.

In this chapter, we provide an analysis of HMP scheduler. The knowledge of the behavior and scalability of multi-threaded applications may be used to enhance performance, in terms of throughput, power and

energy consumption of such applications. For this reason, we ran several tests on the Odroid XU3 in order to profile the behavior of HMP scheduler in case of multi-threaded applications, i.e. the applications we want to target. Section 3.5 presents the throughput analysis of a multi-threaded application. Section 3.6 is about the first prototype of our solution we found to both balance the workload among the cores and improve performance. Section 3.7 shows two attempts of enhancement for our solution. Finally, Section 3.8 reviews the analysis we performed on HMP scheduler.

3.5 Throughput Analysis

For these tests, we will focus on Black Scholes (BS) [96] benchmark, i.e. an implementation of Black and Scholes model, a mathematical model of financial market, widely used in global financial markets to calculate the theoretical price of European options (a type of financial security). We profiled BS benchmark in different configurations. We define a configuration as a tuple $< N_B, f_B, N_L, f_L >$, where:

1) N_B is the number of used big cores,

2) f_B is the frequency of big cores,

3) N_L is the number of used LITTLE cores,

4) f_L is the frequency of LITTLE cores.

We tested how BS scales as the number of cores and frequency change. In particular, we focused only on configurations that use only either big cores or LITTLE cores. In this way, we wanted to find a ratio between big and LITTLE throughput performance. For each subset of big/LITTLE cores, we tested all possible frequencies, starting from 800MHz, to big/LITTLE maximum frequency (1900MHz/1300MHz), with a step of 100MHz. Hence, the dimension of the configuration space we analyzed is:

$$|N_B| \times |f_B| + |N_L| \times |f_L| = 4 \times 12 + 4 \times 6 = 72$$

The command `cpufreq-set` [97] allows to change the frequency of either big or LITTLE cores. On the other hand, the command `taskset` [98] allows to set or retrieve the CPU affinity of a new command or a running one. Hence, we can decide on which core we want to run an application. BS kernel is performed 300 times, while each kernel instance executes 4000000 options. BS benchmark is parallelized by OpenMP Application Programming Interfaces (APIs); in particular, OpenMP uses 8 threads by default for each kernel execution. As we stated in Chapter 4, the results of our tests showed that BS is a well-balanced benchmark, which scales almost ideally with the number of used cores (when we employ only one cluster at the time). In Figure 3.1 we reported again BS performance chart.
We can notice that 1 big core at 1900MHz, ideally, performs as 2 LITTLE cores at 1200MHz. As well, 2 big cores at 1900MHz ideally perform as 4 LITTLE at 1100MHz. Therefore we can use this information in order to have a better mapping of threads on big and LITTLE cores. In particular, for each thread we map on a LITTLE core, we should map 2 threads on a big core.
Finally, when we tested BS on all the other configurations, we noticed weird behavior of HMP in all the configurations of this form: $< 1, f_B, N_L, f_L >$, with $N_L \geq 1$. In particular, although a number from 1 to 4 of LITTLE cores was available, HMP scheduled the task on the big core only. On the other hand, in all the other configurations different from $< 1, f_B, N_L, f_L >$, HMP performs properly. Such final analysis was carried out using process and resource viewers, like **htop** [99]. This unexpected behavior suggested us that HMP has difficulties in dealing with multi-threaded applications, in particular when they are parallelized on big and LITTLE cores at different frequencies. Our idea is to enhance our run-time resource management policy in order to balance workload by properly spread the instantiated threads among the cores according to a mapping ratio.

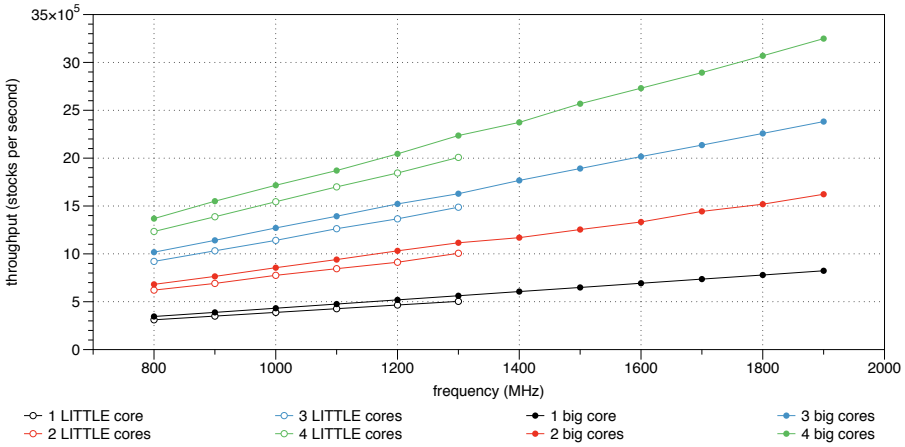


Figure 3.1: Black Scholes throughput as frequency and cores change

3.6 Thread Mapping

We have seen so far that it is possible to find a ratio between big and LITTLE cores throughput performance. In BS case, this ratio is close to 2. Hence, we want to use such value and manually map threads to big/LITTLE cores. Specifically, when we map 1 thread on a LITTLE core, we map 2 threads on a big core. The default number of threads available for our architecture is 8. When a perfect mapping is not available, we tried different mappings in order to find the most suitable for that configuration. Linux provides APIs able to set the affinity of a thread to a certain core. The command is `sched_setaffinity(tid, sizeof(set), &set)` [100], where:

- `tid` is the thread ID,
- `set` is a variable of type `cpu_set_t` and represents the core the thread has to be mapped to.

For our tests, we set core frequency to 1900MHz for big cores and 1300MHz for LITTLE cores. This because the frequencies of LITTLE cores we observed having similar behavior, in terms of throughput, to big cores vary. Hence, in first approximation, we decided to set LITTLE cores frequency to their maximum, as well as big cores frequency.

The configurations we are going to consider in these tests are the ones of this kind: $< N_B, 1900, N_L, 1300 >$. In particular, we split them in two subsets:

- *Critical Configurations*, defined by the tuple: $< 1, 1900, N_L, 1300 >$, with $N_L \geq 1$,
- *Generic Configurations*, defined by the tuple: $< N_B, 1900, N_L, 1300 >$, with $N_B > 1$ and $N_L \geq 1$.

The former are the configurations where we noticed that HMP does not spread the workload over the cores available, the latter are the ones where HMP works properly. For each subset of configuration, we test HMP versus our solution, which means manually mapping threads to big/LITTLE cores, with respect to the ratio we previously found. Finally, we compare the results of our tests in terms of throughput, measured as stocks per second (sps), execution time (second), power consumption (watt), energy consumption (joule), and, finally, throughput over energy consumption (sps / joule) as a cumulative measure of all the previous metrics. Power and energy consumption measures represent the consumption of the whole system.

3.6.1 Critical Configurations Tests

Here we present the tests run on Critical Configurations, i.e. configurations whose form is $< 1, 1900, N_L, 1300 >$, where $N_L \geq 1$. In such configurations, HMP does not perform in a proper way. In other words, HMP does not parallelize the workload over the cores available, but it runs the task on the big core only. As consequence, the application throughput is lower than what it could actually be. For this reason, we want to see if, manually allocating threads on cores, we can improve the application throughput.

3.6.1.1 1 big core - 3 LITTLE cores

In configuration $< 1, 1900, 3, 1300 >$, since a perfect mapping was not possible, we tried two mappings:

1) 2 threads on 1 big core, 2 threads on each LITTLE core,

2) 4 threads on 1 big core, 2 threads on 1 LITTLE core, and 1 thread on each of the remaining 2 LITTLE cores.

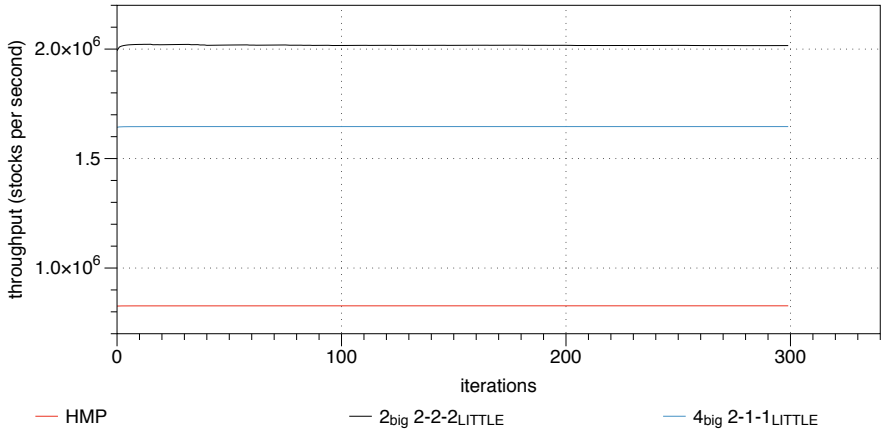
The results of the tests are the following:

throughput: both our solutions surpass HMP scheduler in terms of throughput (Figure 3.2(a)). Specifically, HMP throughput, on average, is 8.28×10^5, while our mappings throughput are, respectively, 20.17×10^5 and 16.46×10^5. This means that, in this Critical Configuration, one of our mappings can achieve a 2.44X speedup.

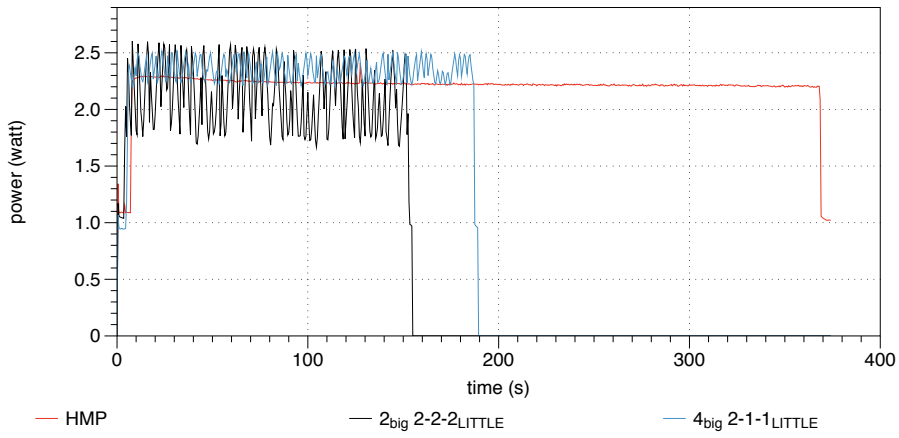
power consumption: HMP scheduler is not the most power efficient solution (Figure 3.2(b)); indeed, our first mapping consumes, on average, `2.07W`, our second mapping `2.29W`, and, finally, HMP `2.19W`. If we consider the execution time, HMP is the slowest solution (`374s`), whereas our mappings take `154s` and `189s`.

energy consumption: thanks to their execution times and, for the first mapping, also its power consumption, the energy required by our mappings is quite smaller than HMP (Figure 3.2(c)). In particular, HMP energy consumption is `1638.30J`, instead our mappings consume, respectively, `642.47J` and `869.38J`.

throughput-energy ratio: again, the ratio shows that our solutions are overall more efficient than HMP (Figure 3.2(d)). The ratio values are, for our mappings, 3139.87 and 1893.39, while, for HMP, 505.04.

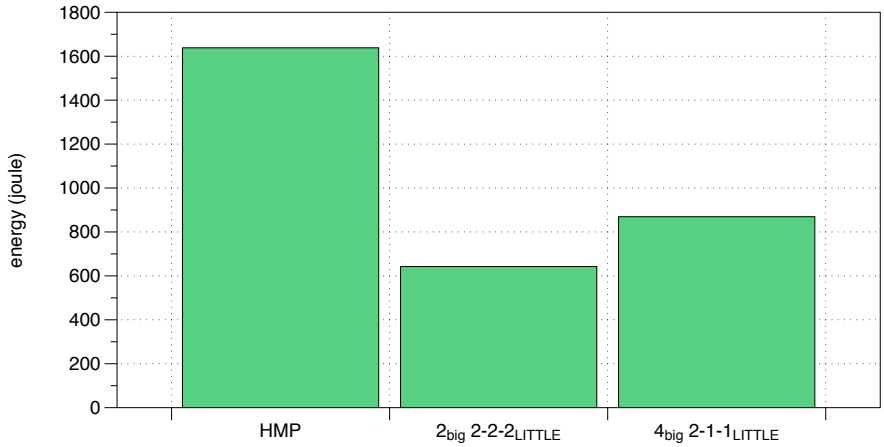


(a) 1 big - 3 LITTLE throughput

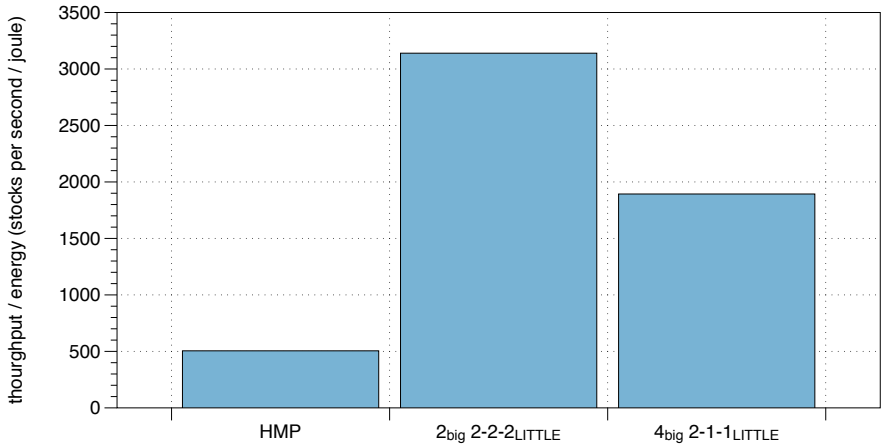


(b) 1 big - 3 LITTLE power consumption

Figure 3.2: 1 big - 3 LITTLE test



(c) 1 big - 3 LITTLE energy consumption

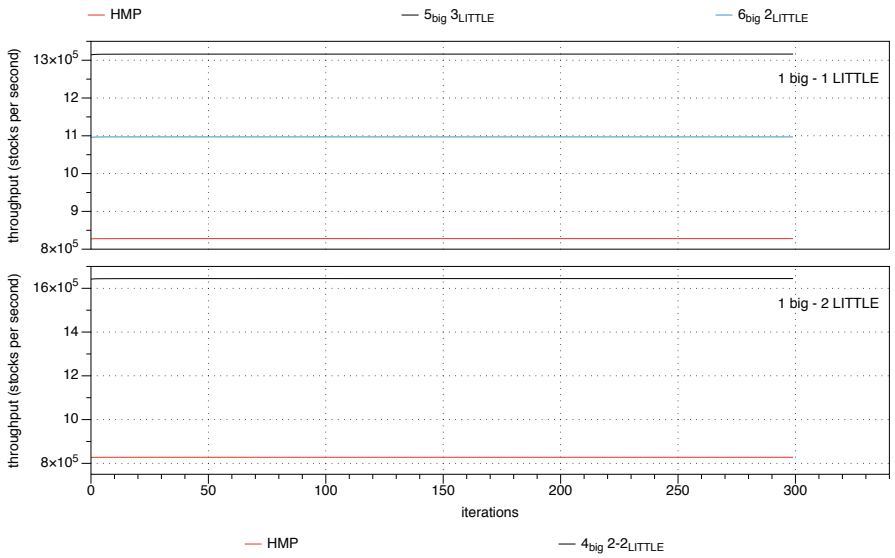


(d) 1 big - 3 LITTLE throughput-energy ratio

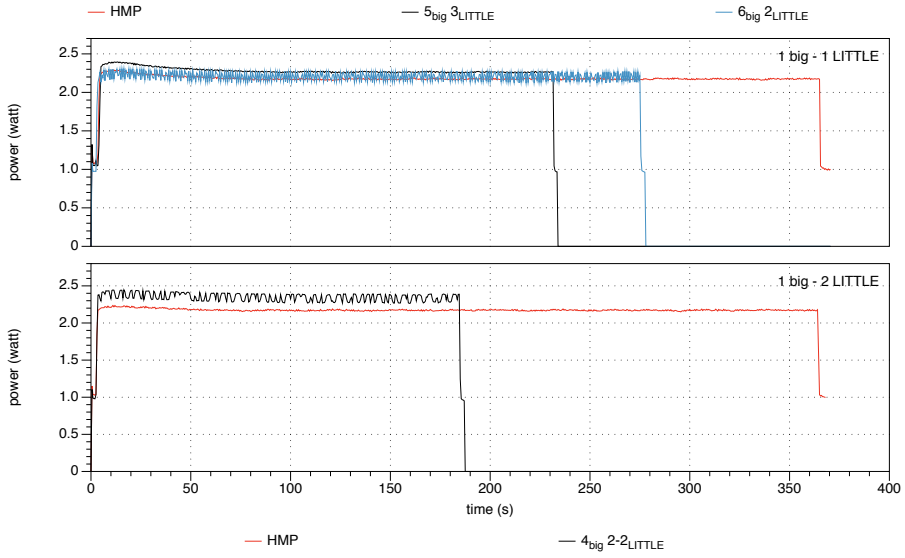
Figure 3.2: 1 big - 3 LITTLE test

3.6.1.2 Critical Configurations Summary

We have seen that HMP scheduler has a strange behavior in the Critical Configuration we tested. The same behavior can be observed in Critical Configurations $< 1, 1900, 1, 1300 >$, $< 1, 1900, 2, 1300 >$ and $< 1, 1900, 4, 1300 >$. Figure 3.3 shows the same tests performed on Critical Configurations $< 1, 1900, 1, 1300 >$, $< 1, 1900, 2, 1300 >$, while the results of tests on Critical Configuration $< 1, 1900, 4, 1300 >$ are reported at the end of the chapter in Table 3.2, as well as all the other tests results.
In all the cases, our thread mappings result in better throughput (around 2X speedup in most of the cases). Although some mappings consume few hundred `mWatts` more than HMP, this does not impact the overall energy consumption since execution time of our mappings is impressively shorter than HMP. As consequence, our solutions energy consumption is remarkably smaller than HMP. As well, throughput-energy ratio, which indicates the goodness of one solution with respect to another, always promotes our thread mapping against HMP.
It is interesting to highlight that HMP throughput, execution time, power consumption, energy consumption, and throughput-energy ratio almost do not change in all the 4 configurations we tested. Just configuration $< 1, 1900, 4, 1300 >$ has slightly better throughput. This means that having 1 to 4 LITTLE cores available, in addiction to 1 big core, makes quite no difference to HMP scheduler.

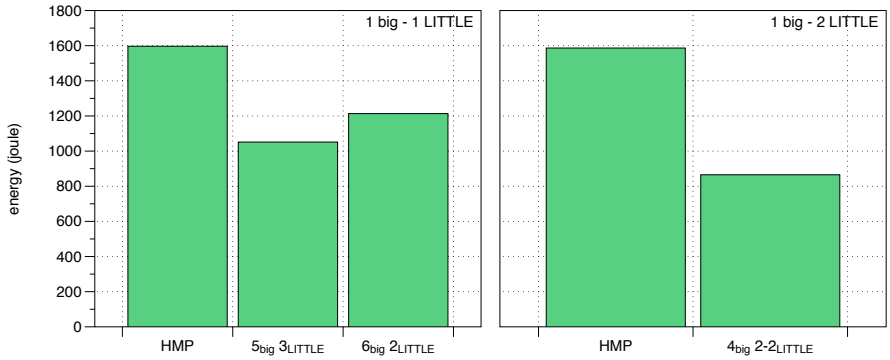


(a) 1 big - 1 LITTLE and 1 big - 2 LITTLE throughput

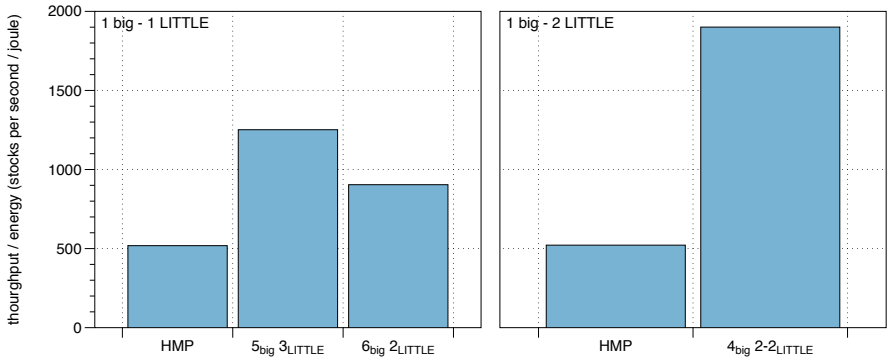


(b) 1 big - 1 LITTLE and 1 big - 2 LITTLE power consumption

Figure 3.3: 1 big - 1 LITTLE and 1 big - 2 LITTLE tests



(c) 1 big - 1 LITTLE and 1 big - 2 LITTLE energy consumption



(d) 1 big - 1 LITTLE and 1 big - 2 LITTLE throughput-energy ratio

Figure 3.3: 1 big - 1 LITTLE and 1 big - 2 LITTLE tests

3.7 Enhancing Thread Mapping

We performed tests on Generic Configurations $< 2, 1900, 2, 1300 >$ and $< 4, 1900, 4, 1300 >$ (the results are reported in Table 3.2). From these tests, it is evident that, when we are dealing with Generic Configurations, HMP performs better than our thread mapping solution, in terms of both throughput and execution time, although our mappings power consumption is smaller than HMP, as well as energy consumption (for some mappings). Hence, we need to improve our mapping solution if we want to deal with Generic Configurations.
In this section, we present the enhancements we introduced in our thread mapping solution in order to improve our performance, particularly in terms of throughput. In Section 3.7.1, we tested two Generic Configurations: $< 2, 1900, 2, 1300 >$ and $< 3, 1900, 2, 1300 >$. The goal was to have a perfect mapping according to our ratio. Therefore, when it was necessary, we changed the number of usable threads.
Finally, we also tried to exploit OpenMP dynamic scheduler in order to check if we could improve the throughput of our perfect mappings. While OpenMP static scheduler divides the loop into equal-sized chunks, or, at least, as equal as possible, OpenMP dynamic scheduler exploits internal work queue to assign each thread a chunk-sized block of the loop iterations. Besides, at the end of a thread execution, the thread pops the next block from the top of the work queue. Since this solution did not provide an improvement in terms of the metrics we are considering, the results of such tests are directly reported in Table 3.2.

3.7.1 Perfect Mapping

Here, we present the perfect mapping tests run on Generic Configurations $< 2, 1900, 2, 1300 >$ and $< 3, 1900, 2, 1300 >$, where we used a perfect mapping of threads on the available cores, according to the ratio we found for BS benchmark. While in the former configuration we changed the number of usable threads, the latter configuration was already suitable for this goal.

3.7.1.1 2 big cores - 2 LITTLE cores

Since for configuration $< 2, 1900, 2, 1300 >$ a perfect mapping with 8 threads was not available, we set the number of usable threads to 6. Hence, we ran the tests with the following schedules:

1) HMP with 8 and 6 usable threads,

2) 2 threads on each big core, 1 threads on each LITTLE core.

The results of the perfect mapping test for the Generic Configuration $< 2, 1900, 2, 1300 >$ are the following:

throughput: 6-threads HMP and our perfect mapping have, on average, throughput almost identical to 8-threads HMP (Figure 3.4(a)). In particular, while 8-threads HMP throughput is 24.73×10^5, 6-threads HMP throughput is 24.50×10^5, and our perfect mapping throughput is 24.69×10^5. In terms of slowdown, for 6-threads HMP it is 0.991X, for our perfect mapping it is 0.999X.

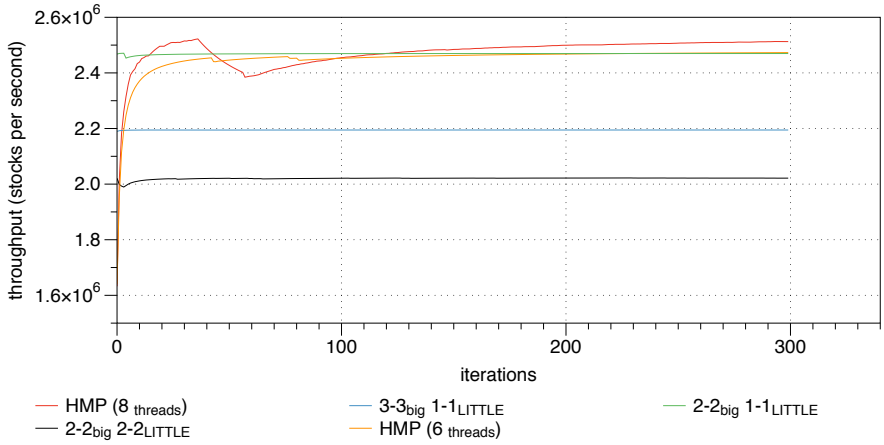
power consumption: both 6-threads HMP and our perfect mapping power consumption are slightly smaller than 8-threads HMP (Figure 3.4(b)); indeed, while 8-threads HMP power consumption is `3.48W`, 6-threads HMP requires, on average, `3.46W`, and our perfect mapping averagely consumes `3.45W`, which is higher than the power consumed by the previous mappings for this Generic Configuration. If we consider the execution time, 8-threads HMP is still the best solution (`122s`), while the time needed by 6-threads HMP and our perfect mapping to complete the benchmark is nearly the same (respectively, `127s` and `126s`). The execution time of our perfect mapping is definitely shorter than the one required by our previous mappings.

energy consumption: since the power consumption our the three solutions we are considering in this test are very similar, the energy consumption mainly depends on the execution time (Figure 3.4(c)). As consequence, 8-threads HMP is still the most energy efficient solution (`856.06J`), instead 6-threads HMP is the most energy-hungry solution (`886.28J`). Finally our perfect mapping consumes `872.39J`,

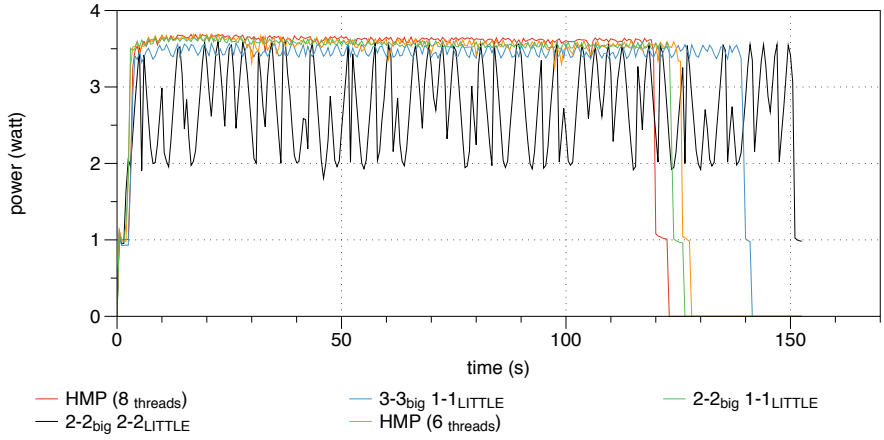
and, due to the higher power consumption, such value is higher than the energy consumed by the previous mappings.

throughput-energy ratio: again, 8-threads HMP results to be the most efficient schedule, according to this metric (Figure 3.4(d)). On one hand, 8-threads HMP throughput-energy ratio is 2888.39, on the other, 6-threads HMP ratio value is 2763.97, and our perfect mapping ratio value is 2830.06, an enhancement with respect to the previous mappings.

This test shows that, when we can exploit a perfect thread mapping, our solution is capable of reaching and overtaking HMP performance. Indeed, our perfect mapping solution is superior to 6-threads HMP solution in each metric we considered, even though some values are very close. On the other hand, 8-threads HMP has still better performance than our solution, although our perfect mapping performance are similar to 8-threads HMP than our previous mappings for this Generic Configuration.

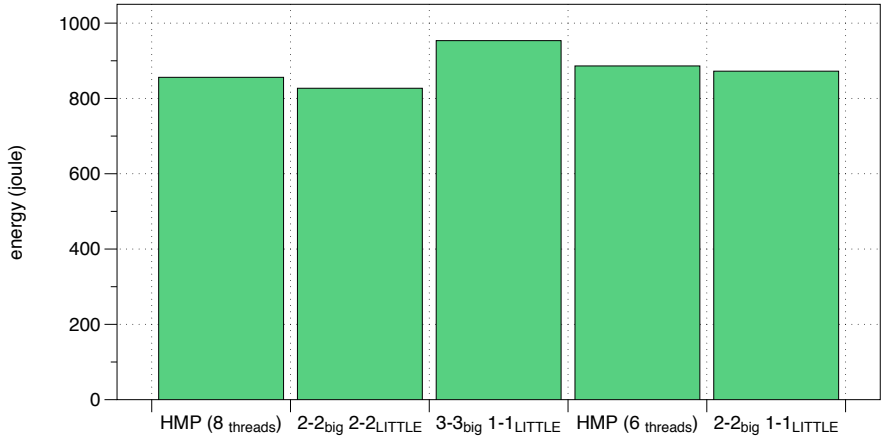


(a) Perfect mapping 2 big - 2 LITTLE throughput

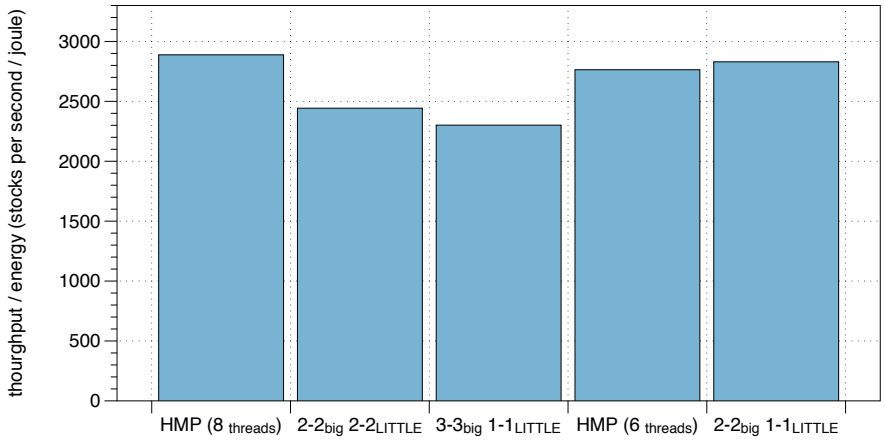


(b) Perfect mapping 2 big - 2 LITTLE power consumption

Figure 3.4: Perfect mapping 2 big - 2 LITTLE test



(c) Perfect mapping 2 big - 2 LITTLE energy consumption



(d) Perfect mapping 2 big - 2 LITTLE throughput-energy ratio

Figure 3.4: Perfect mapping 2 big - 2 LITTLE test

3.7.1.2 3 big cores - 2 LITTLE cores

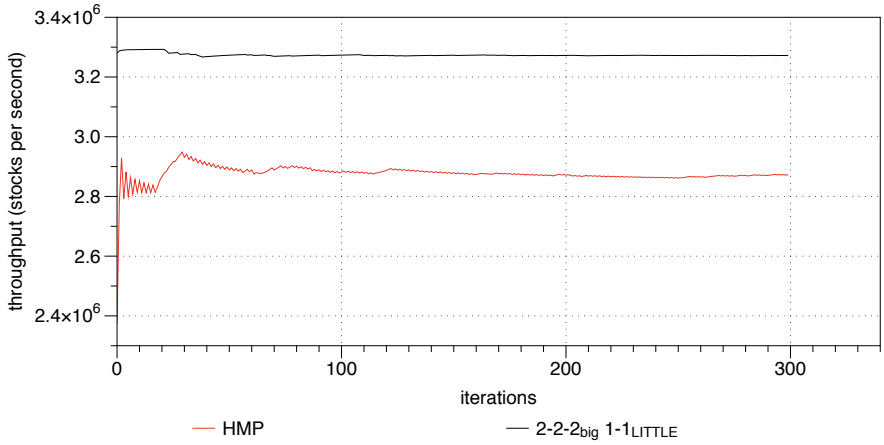
Configuration $< 3, 1900, 2, 1300 >$ provides a perfect thread mapping, which is: 2 threads on each big core, 1 thread on each LITTLE core. Here, we introduce the results of the tests for Generic Configuration $< 3, 1900, 2, 1300 >$:

throughput: differently from previous Generic Configuration tests, our perfect mapping is able to surpass HMP performance in terms of throughput (Figure 3.5(a)). Specifically, HMP throughput is, on average, 28.75×10^5, whereas our perfect mapping average throughput is 32.74×10^5.

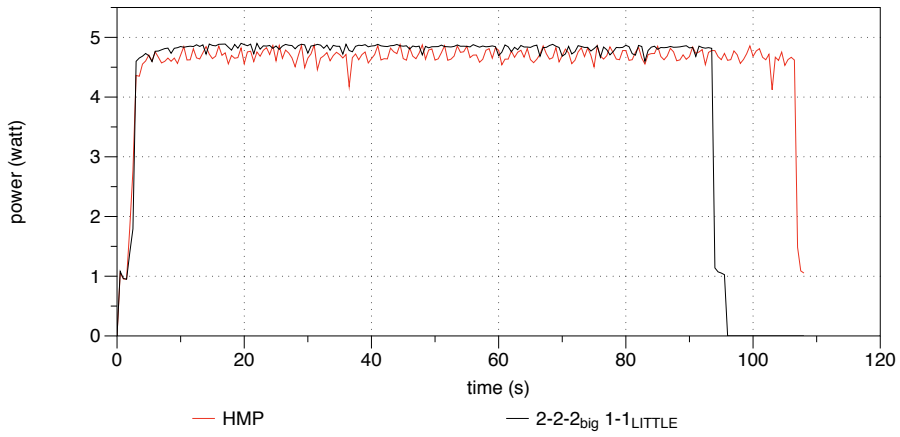
power consumption: HMP results to be more power efficient than our perfect mapping solutions (Figure 3.5(b)); indeed, HMP power consumption is averagely `4.55W`, while our perfect mapping consumes little more power (`4.63W`). In terms of execution time, our perfect mapping results to be faster than HMP. (`95s` versus `108s`).

energy consumption: Our perfect mapping results to be more energy efficient than HMP solution (Figure 3.5(c)). In particular, HMP consumes `986.98J`, while the energy consumption of our perfect mapping is `888.39J`. This is due to the difference in the execution times, since the power consumption are quite close.

throughput-energy ratio: this metric suggests that the most efficient schedule is our perfect mapping solution (Figure 3.5(d)). The throughput-energy ratio value of our perfect mapping is 3685.14, while the value of HMP scheduler is 2913.27.

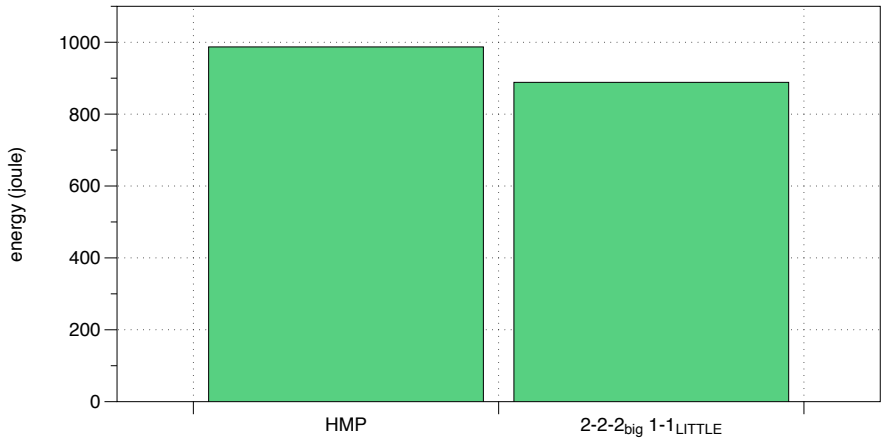


(a) Perfect mapping 3 big - 2 LITTLE throughput

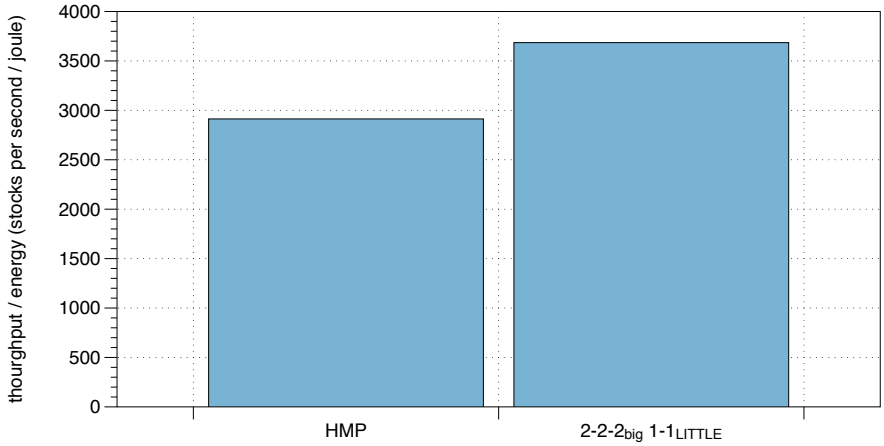


(b) Perfect mapping 3 big - 2 LITTLE power consumption

Figure 3.5: Perfect mapping 3 big - 2 LITTLE test



(c) Perfect mapping 3 big - 2 LITTLE energy consumption



(d) Perfect mapping 3 big - 2 LITTLE throughput-energy ratio

Figure 3.5: Perfect mapping 3 big - 2 LITTLE test

3.7.1.3 Thread Mapping Summary

These tests proved that, when a perfect mapping is available (thanks to the configuration or setting the number of usable threads), it is possible to achieve at least the same throughput of 8-threads HMP scheduler.
In Generic Configuration $< 2, 1900, 2, 1300 >$, our perfect mapping reached, on average, performance very similar to 8-threads HMP, in terms of throughput and power consumption. On the other hand, execution time and energy consumption of 8-threads HMP are slightly better than our perfect mapping ones. If we compare our solution with 6-threads HMP, it results to have better performance in all metrics. Finally, in throughput-energy ratio metric, our perfect mapping performance is very close to 8-threads HMP, whereas it surpasses 6-threads HMP.
General Configuration $< 3, 1900, 2, 1300 >$ did not require a change in the number of available threads, since it already provided a perfect mapping. Our solution performs certainly better than HMP in terms of throughput, execution time and energy consumption, while just the power consumption is marginally higher than HMP. Therefore, also throughput-energy ratio of our perfect mapping is superior to HMP ratio.

3.8 Conclusions

We have seen so far that it is possible to improve the performance of HMP, the Odroid XU3 heterogeneous scheduler. In particular, our solution is manually mapping threads to cores according to a ratio between big and LITTLE cores we found analyzing how BS benchmark scales as the number of cores and frequency change.
We tested two kinds of configurations: Critical and Generic Configurations. In the former case, we took advantage of HMP improper behavior, and we always achieved great results. In the latter case, at first, HMP performance was definitely better than our thread mapping solution. Here HMP properly spreads the workload over the available cores. Besides, the Generic Configurations we tested did not provide a perfect mapping. Hence, we either changed the number of usable threads or

chose a suitable configuration in order to achieve a perfect mapping. In this way, we improved our solution throughput performance and reached, at least, 8-threads HMP throughput. Finally, we exploited OpenMP features to further progress our solution performance, but we did not achieve the desired results. Table 3.2 collects the results of all the tests we ran. The speedup is computed with respect to the 8-threads not-dynamic HMP schedule of each configuration.
The limitations of HMP scheduler we presented in this chapter will be used as starting point for the development of our proposed solution, i.e. a workload-aware run-time resource management policy able to find a convenient configuration for the application goal, automatically compute the mapping ratio, and finally distribute the workload.

Configuration	Scheduler/Mapping	Threads	Speedup	Throughput (Msps)	Time (s)	Power (W)	Energy (J)	$\frac{\text{Throughput}}{\text{Energy}}$ ($\frac{\text{sps}}{\text{J}}$)
$< 1, 1900, 1, 1300 >$	HMP	8	1X	0.828	370	2.15	1596.80	518.51
$< 1, 1900, 1, 1300 >$	5_{big} 3_{LITTLE}	8	1.59X	1.316	233	2.25	1051.57	1251.63
$< 1, 1900, 1, 1300 >$	6_{big} 2_{LITTLE}	8	1.33X	1.097	277	2.18	1213.54	904.01
$< 1, 1900, 2, 1300 >$	HMP	8	1X	0.828	368	2.15	1586.58	521.74
$< 1, 1900, 2, 1300 >$	4_{big} 2-2_{LITTLE}	8	1.97X	1.644	187	2.31	865.23	1900.55
$< 1, 1900, 3, 1300 >$	HMP	8	1X	0.828	374	2.19	1638.30	505.04
$< 1, 1900, 3, 1300 >$	2_{big} 2-2-2_{LITTLE}	8	2.44X	2.017	154	2.07	642.47	3139.87
$< 1, 1900, 3, 1300 >$	4_{big} 2-1-1_{LITTLE}	8	1.99X	1.646	189	2.29	869.38	1893.39
$< 1, 1900, 4, 1300 >$	HMP	8	1X	0.918	368	2.15	1587.09	578.62
$< 1, 1900, 4, 1300 >$	4_{big} 1-1-1-1_{LITTLE}	8	1.77X	1.641	186	2.31	864.17	1898.39
$< 2, 1900, 2, 1300 >$	HMP	8	1X	2.473	122	3.48	856.06	2888.39
$< 2, 1900, 2, 1300 >$	2-2_{big} 2-2_{LITTLE}	8	0.82X	2.020	152	2.70	827.02	2442.88
$< 2, 1900, 2, 1300 >$	3-3_{big} 1-1_{LITTLE}	8	0.89X	2.195	141	3.37	953.52	2301.52
$< 2, 1900, 2, 1300 >$	HMP	6	0.991X	2.450	127	3.46	886.28	2763.97
$< 2, 1900, 2, 1300 >$	2-2_{big} 1-1_{LITTLE}	6	0.999X	2.469	126	3.45	872.39	2830.06
$< 2, 1900, 2, 1300 >$	dynamic HMP	8	0.56X	1.376	222	3.21	1430.30	962.54
$< 2, 1900, 2, 1300 >$	dynamic HMP	6	0.55X	1.367	223	3.20	1429.48	956.06
$< 2, 1900, 2, 1300 >$	dynamic 2-2_{big} 1-1_{LITTLE}	6	0.80X	1.980	154	3.38	1043.21	1898.10
$< 3, 1900, 2, 1300 >$	HMP	8	1X	2.875	108	4.55	986.98	2913.27
$< 3, 1900, 2, 1300 >$	2-2-2_{big} 1-1_{LITTLE}	8	1.14X	3.274	95	4.63	888.39	3685.14
$< 3, 1900, 2, 1300 >$	dynamic HMP	8	0.72X	2.071	149	4.37	1310.16	1580.90
$< 3, 1900, 2, 1300 >$	dynamic 2-2-2_{big} 1-1_{LITTLE}	8	0.90X	2.597	119	4.59	1096.14	2369.11
$< 4, 1900, 4, 1300 >$	HMP	8	1X	4.422	72	5.98	873.69	5061.40
$< 4, 1900, 4, 1300 >$	1-1-1-1_{big} 1-1-1-1_{LITTLE}	8	0.91X	4.015	82	4.18	693.23	5791.47

Table 3.2: Tests summary

Problem Definition and Proposed Solution 4

This chapter defines the problem we want to tackle with our work and introduces our solution, which is a workload-aware run-time resource management policy. Our proposal aims at both satisfying QoS requirement (i.e. minimum throughput) and reducing power consumption by proper allocating tasks on a combination of cores (big and LITTLE) number and frequency, and balancing their threads on available resources. Section 4.1 presents the rationale of this thesis work. Section 4.2 outlines the details of our proposed solution.

4.1 Problem Definition

The actual scenario is characterized by various and flexible on-demand computing workloads, with different requirements, like throughput, power consumption and so on. HSAs result to be the most suitable architectures for such a scenario, since their nature allows them to satisfy different goals (e.g. throughput, power and energy consumption, etc.). Therefore, a self-adaptive management of HSA resources capable of dealing with various and dynamic workloads may be the right approach to tackle this problem.
In the following subsections we will define more in detail the problem we are focusing on. At first, we introduce the kind of applications that this work aims at satisfying (Section 4.1.1). Then, we profile such applications and our target device in terms of both power consumption (Section 4.1.2) and performance (Section 4.1.3), in order to describe the properties they are supposed to feature. Indeed, we expect the power

consumption of big and LITTLE clusters to scale as cV^2f (being c a constant typical of each cluster, V the voltage, and f frequency levels), while the applications throughput is assumed to linearly scale as the frequency increases and, since the applications are multi-threaded, as the number of cores enlarges.

4.1.1 Application Models

The approach we propose is designed for applications with computational-intensive loops, where the user can specify some minimum throughput constraints on the performance of the loop. Our reference application model is the one reported in Figure 4.1, which is a cyclic task graph where the kernel node, the computationally intensive part of the code, might be potentially parallelized with the fork-join paradigm, by using state-of-the-art libraries, such as OpenMP [101].
There are many applications that follow such pattern and have been implemented in a parallel way on CPUs, GPUs or FPGAs; for instance, financial algorithms, like BS benchmark introduced in Chapter 3, or video processing ones. Numerical analysis algorithms belong to this category too. An example is *Jacobi iterative method* [102], which is used to determine the solutions of a diagonally dominant system of linear equations. Another application is *Stochastic Simulation algorithm (SSA)* [103, 104], an exact procedure for numerically simulating the time evolution of a well-stirred chemically reacting system. Then, *Barnes-Hut algorithm* [105] is a physics method for directly computing the force on N bodies in the gravitational N-body problem. *Reverse Time Migration (RTM)* [106] is a geophysics algorithm for seismic imaging that aims at constructing an image of the subsurface from recordings of seismic reflections. Finally, *Brain Network* image analysis applications exploit the computation of *Pearson Correlation Coefficient (PCC)* [107], a statistic coefficient that measures the degree of linear correlation between two variables, to infer interconnections between neurons.
This kind of applications may have some throughput requirements that the system has to meet, and they can be expressed as performance requirements over the iterative part of the program. Starting from the model represented in Figure 4.1, the user can then express a constraint

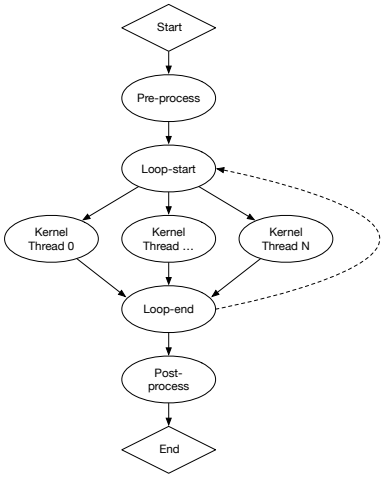


Figure 4.1: Application model

on the performance of the program included in the *Loop-start* - *Loop-end* nodes of the description. In this context, we assume that the *Pre-process* and *Post-process* tasks are not computational-intensive and do not impact on application performance.

In order to collect the performance information for all the running applications, we assume that high-level information is available and exported by the application. This means that, instead on relying on low-level metrics such as Instructions Per Cycle (IPC), it is preferable to collect information at an higher abstraction level that is closer to the final user; for instance, we want to express the performance of a video processing application in terms of *frames/s*. To obtain such a metric, we need to instrument the applications so that they communicate their progress at the end of each iteration (i.e. in the *Loop-end* node).

Even though the instrumentation of an application is a invasive task, current libraries such as Application Heartbeats and similar [108, 109, 110] allow for a minimal modification of the application code (a couple of lines of code). The availability of this high-level information is a benefit not only for the decision mechanism, which can know exactly when an application progresses, but also for the final user who can specify requirements using a meaningful metric.

4.1.2 DVFS Management

A first analysis consisted in profiling the power consumption with different DVFS levels available for the considered architecture. The power consumption of both big and LITTLE clusters is supposed to scale proportionally to cV^2f expression, where c is a constant typical of each cluster, V the voltage, and f frequency levels.
Based on the specific architecture, DVFS can be available at the core level, or cluster-level; in the adopted reference platform, the Odroid XU3 development board, this feature is supported at cluster level, by means of DVFS actuators and power sensors. This means that, as we set a frequency value for a type of cores (big or LITTLE), this is set for all the cores belonging to that cluster. On the Odroid XU3, Big and LITTLE cores have different frequency ranges: big cores frequency goes from 800MHz to 1900MHz, whereas LITTLE cores frequency from 800MHz to 1300MHz.
To collect per-core full power consumption, we used used the Linux `stress` utility [111], which forces a full load on the system and we computed per-core power consumption by dividing the sensed values by the number of used cores. Table 4.1 reports, per cluster, the power consumed in idle and full power for all the possible DVFS configurations; Figure 4.2 illustrates the power curves when only a subset of big or LITTLE cluster is used (the cluster may be distinguish by looking at the reached frequencies).
The collected data (which confirm the above power scaling hypothesis) show that the Cortex-A7 cluster used at the highest frequency is generally less power hungry than two Cortex-A15 cores used at the minimum frequency. Therefore, when running a parallel application, if the LITTLE cores achieve an acceptable performance, it is not necessary to use big cores even if the running applications do compute-intensive operations. Furthermore, according to the accurate experimental analysis, we can also assume that the power profile of the processor does not significantly vary depending on the application running on the processor, but it depends on the load the application causes on the resource. As a consequence, the power metrics derived from the previous experiment can be used for any application, provided that a scaling factor given by

Cluster	Voltage (V)	Freq. (MHz)	Idle Power (W)	Full Power (W)
A7	1	800	0.041	0.19
A7	1	900	0.059	0.232
A7	1	1000	0.07	0.273
A7	1.1	1100	0.083	0.319
A7	1.1	1200	0.096	0.369
A7	1.2	1300	0.116	0.437
A15	0.9	800	0.187	0.864
A15	0.9	900	0.21	0.955
A15	0.9	1000	0.242	1.116
A15	0.9	1100	0.278	1.284
A15	1	1200	0.318	1.469
A15	1	1300	0.362	1.731
A15	1	1400	0.395	1.836
A15	1	1500	0.43	2.065
A15	1	1600	0.5	2.4
A15	1.1	1700	0.581	2.779
A15	1.1	1800	0.666	3.197
A15	1.2	1900	0.8	3.985

Table 4.1: Samsung Exynos 5422 power measurements table

.

the processor utilization is adopted. These measurements will thus be used to characterize the power consumption of the cores in the SAVE Virtual Platform, which registers the system utilization and can provide at each simulation step a power measurement depending on the current workload.

4.1.3 Performance Considerations

Our methodology does not focus on power only, but aims at meeting also performance requirements. We expect the performance to scale linearly as frequency increases. Moreover, since our target applications are multi-

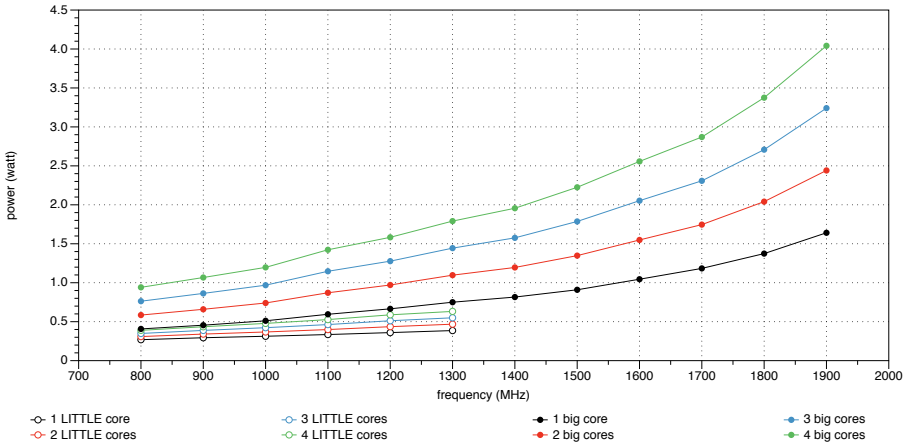
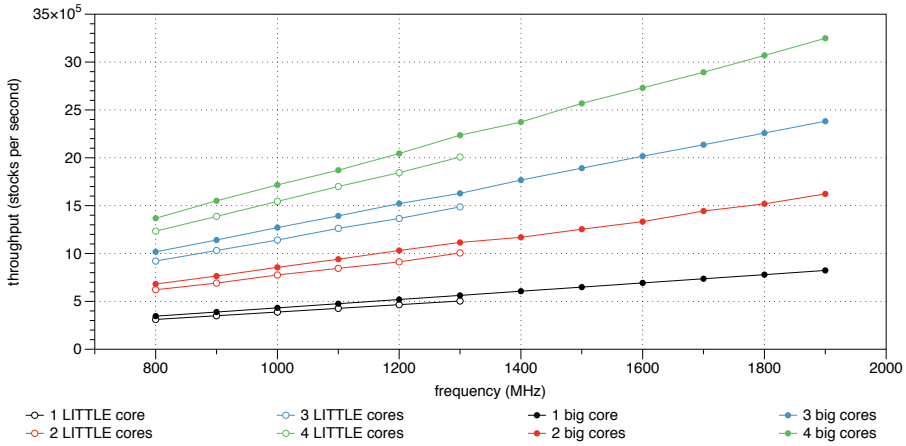
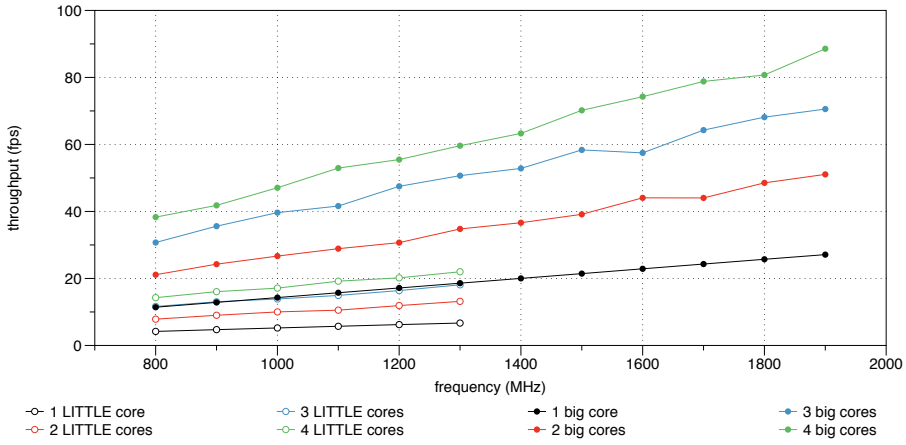


Figure 4.2: Samsung Exynos 5422 power measurements chart

threaded, the performance are supposed to scale also with the number of cores. However, while the power does not depend on the specific application but rather on the load, performance, intended as throughput, depends not only on the specific application, but also on the interaction between different applications concurrently running on the system. To design an optimization policy it is thus necessary to put performance and power into relation, and rapidly investigate the different possible working points. Figure 4.3 reports the performance profile of two of the benchmarks we used on the reference platform. The benchmarks are: *PCC* and *BS*. Both benchmarks are parallel versions implemented with OpenMP starting from the code available in CUDA [112] examples.
The measurements have been collected by running the two applications in isolation and we collected the average throughput of each application during its whole execution. For each subset of big/LITTLE cores, we tested all possible frequencies, starting from 800MHz, to big/LITTLE maximum frequency (1900MHz/1300MHz), with a step of 100MHz. For sake of simplicity we report only the curves that use exclusively one cluster. Such tests confirm the above performance hypothesis, and it illustrates how we can achieve different performance in using the big or the LITTLE cores depending on the application running in the system. In fact, the BS benchmark can benefit from parallelization using the



(a) Black Scholes benchmark



(b) Pearson Correlation Coefficient benchmark

Figure 4.3: Benchmarks throughput as frequency and cores change

Cortex-A7 cores. This benchmark scales almost ideally with the number of used cores and, for this reason, it has good performance also on the Cortex-A7 cores. On the other hand, the PCC application distributes the computation unfairly among the available cores and therefore does not obtain an ideal speedup, not performing well on Cortex-A7 cores.
Since the proposed policy is tailored for parallel workloads, we can conclude that independently of the actual scalability of the single applications, we will obtain linear dependency on the frequency, and on the number of cores used and their combination. However, this consideration holds as long as we use either big or LITTLE cluster separately. In fact, when we parallelize an application on both clusters, throughput does not scale linearly, but begins to be affected by side effects due to cores synchronizations, in particular when we use different clusters at different frequencies. In such configuration, the ARM big.LITTLE turns into an Asynchronous Clock Architecture (ACA), i.e. an architecture where each core (or cluster in this case) is operated at different voltage levels and clock frequencies. ACA may suffer of performance degradation due to the different frequencies of its cores [41]. This means that, paradoxically, a higher frequency may not imply higher throughput. Figure 4.4 shows how BS throughput scales when we use 2 big cores at 1600MHz and 1 to 4 LITTLE cores from 800MHz to 1300MHz. As results, given the number of cores and frequencies, throughput of either big or LITTLE cluster may be estimated, but not when both clusters are employed (or, at least, an estimate will not be 100% correct). Finally, we noticed that HMP has difficulties in dealing with multi-threaded applications and balancing workload, specially when they are executed on different clusters at different frequencies. An analysis of HMP behavior and how to overcome this problem will be presented in Chapter 3.
In a multiple applications scenario, the Linux scheduler [113] tries to assign a fair amount of CPU time to applications co-located on the same core, leading to a scalability profile that scales by some factor when two or more applications are co-located, but their profile will not drastically change. In case of parallelization by OpenMP API, the workload is fairly distributed among the instantiated threads, even though one application creates more threads than another. For instance, let us consider a benchmark that is run on a Cortex-A15 core. Whether we use one or more

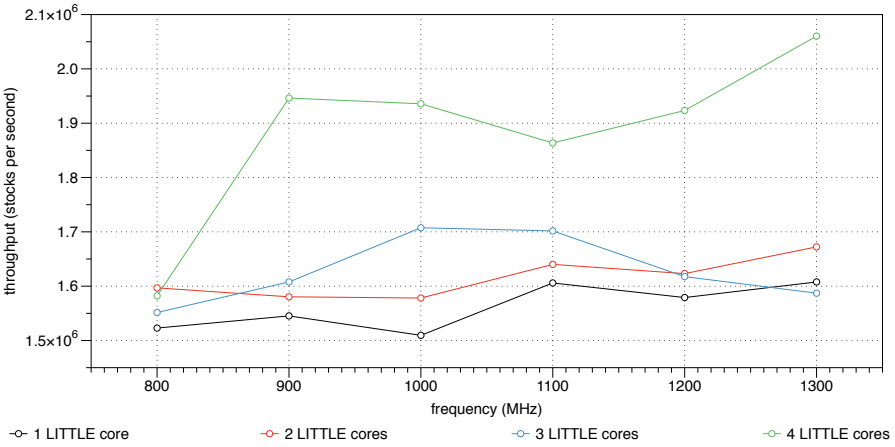


Figure 4.4: Black Scholes thoughput using 2 big cores at 1600MHz

threads, the average benchmark throughput will still be the same, let us call it X. Now, if we run two instances of the benchmark on the same Cortex-A15 core (the first one creates one thread, the second two threads), the final throughput will be, respectively, $\frac{1}{3}X$ and $\frac{2}{3}X$.

4.2 Proposed Solution

The overall idea of the run-time resource management policy we propose is to combine the information regarding power and performance scaling, reported above, to derive a method to explore quickly and effectively the search space at run-time. Moreover, the policy distributes the workload by mapping threads to the available cores according to a mapping ratio. Such value is derived as the ratio between the big cores throughput and LITTLE cores throughput at their given frequencies. Thus, a proper workload distribution ensures higher application performance. Such mapping strategy will be defined in Chapter 3. Finally, our policy can also leverage the availability of online performance estimation obtained by means of an *Application Heartbeat API*.

4.2.1 Configuration Space

The policy we propose is aimed at managing both DVFS and the number of cores currently used by the system exploiting mechanisms to enable/disable cores as the ones exposed by the Linux *sysfs* file system [114]. In this context the possible configurations of our system can be represented by the tuple $< N_B, f_B, N_L, f_L >$, as we did in Chapter 3. The dimension of the configuration space is:

$$conf_B = |N_B| \times |f_B| = 4 \times 12 = 48$$

$$conf_L = |N_L| \times |f_L| = 4 \times 6 = 24$$

$$conf_{BL} = |N_B| \times |f_B| \times |N_L| \times |f_L| = 4 \times 12 \times 4 \times 6 = 1152$$

$$conf_B + conf_L + conf_{BL} = 48 + 24 + 1152 = 1224$$

Note that, in this computation, we assumed that the DVFS controller sets the frequency for the whole cluster and not for the single core, according to the current ARM big.LITTLE architecture, and that there is no difference in the specific combination of selected active cores, which is generally true. The configuration space so derived is large (1224 different configurations), but not cumbersome; these information can be stored in the system memory, but we still need some smart method to navigate this configuration space in search for the right one to use to guarantee both throughput requirements and power efficiency.

4.2.2 Power Characterization

A first aspect to consider is that each one of these configurations is characterized by its power consumption, which is given by the sum of the power consumption of the used cores. The power consumption of given configuration $< N_B, f_B, N_L, f_L >$ is:

$$idle_B = 4 - N_B$$

$$idle_L = 4 - N_L$$

$$power_B = fullPower_B@f_B \times N_B + idlePower_B@f_B \times idle_B$$

$$power_L = fullPower_L@f_L \times N_L + idlePower_L@f_L \times idle_L$$

$$totalPower = power_B + power_L$$

The power information exploited in these formulas come from the previous power profiling.

4.2.3 Performance Characterization

Regarding the performance, we can also suppose that each one of these configurations is characterized by a given performance and that the following assumptions holds:

- application performance benefits from a fairly balanced distribution among used cores;
- application performance generally improves with the number of cores on which computations are parallelized;
- given two different processor types, it is possible to measure the performance gain when switching from one to the other.

Under this considerations we can characterize each configuration with a performance value, in terms of a speedup with respect to a base configuration. Let us suppose we know how the performance of the system scales as we add a Cortex-A15 (Cortex-A7) core or we switch core frequency. We define such factors as $coreScale_B$ ($coreScale_L$) and $freqScale_B$ ($freqScale_L$). Such factors are approximated values extracted during performance analysis tests. Assuming linear scaling and no influence we can compute the speedup with simple formulas. For instance, the speedup expected for the configuration $< N_B, f_B, N_L, f_L >$

can be computed as:

$$speedup_B = N_B \times coreScale_B + \frac{f_B - minf_B}{100} \times freqScale_B \times N_B$$

$$speedup_L = N_L \times coreScale_L + \frac{f_L - minf_L}{100} \times freqScale_L \times N_L$$

$$totalSpeedup = speedup_B + speedup_L$$

Obviously this is an ideal speedup model, which is not 100% accurate, in particular due to the not monotonic behavior of throughput in configurations that employ both big and LITTLE cores. A more accurate model would be too tailored to the benchmark used as reference, and so useless in case of different benchmarks. Moreover, this speedup value is characteristic of every application (when running alone), and it depends on how the applications influence each others when they are co-located. Therefore, it would be useless to profile offline the applications to determine the scale factors to use to setup the policy. For this reason, in our methodology we adapt the speedups relying on online performance measurements and we constantly update those values depending on the current running conditions (i.e. the application mix); Chapter 5 details the update process. However, even if we change online the speedups, our speedup model is monotone; hence configurations can consequently be ordered on the basis of the speedup or the power. Although this may be correct for power consumption, it does not hold for speedup, as we previously stated. Therefore, our policy must be aware of the fact that, given a configuration, a following one, in terms of speedup, might not entail higher throughput.

4.2.4 Policy Overview

In this section we provide an overview of our proposed solution. Our policy is targeted for asymmetric processors and introduces the advantage not only to control DVFS, but, taking care of heterogeneity, also to control the number of currently active cores (big or LITTLE) to achieve

better performance at power and throughput ends.
Figure 4.5 shows the workflow of our policy. Now we briefly describe each Step, while a more detailed analysis is performed in Chapter 5.

Step 1: the policy starts by building a table with all the possible available configurations, their power consumption values and speedup values.

Step 2: the policy profiles the applications in two reference configurations, in order to generate a reference mapping ratio. Such value will then be used to generate the mapping ratios for other configurations.

Step 3: the policy chooses a default operation point to be enforced as initial starting configuration and begins to monitor the applications execution. At each monitoring iteration, the policy computes the needed speedup with respect to the applications performance, and updates the baseline.

Step 4: the policy scans the configuration table to find the optimal configuration, i.e. the one that delivers the needed speedup. The scan is implemented as a variant of binary search.

Step 5: the policy actuates the configuration by changing the affinity mask of all the applications in order to use the same group of cores and changing the operating frequency of the used computational resources.

Step 6: when the policy converges, it scans all the configurations that guarantee the QoS (i.e. all the configurations following the one the policy has converged to) to find the most power efficient configuration.

Step 7: when the application set changes, a reset routine is used to restore some parameters to their default values so that the policy can converge again to a convenient configuration for the new application set.

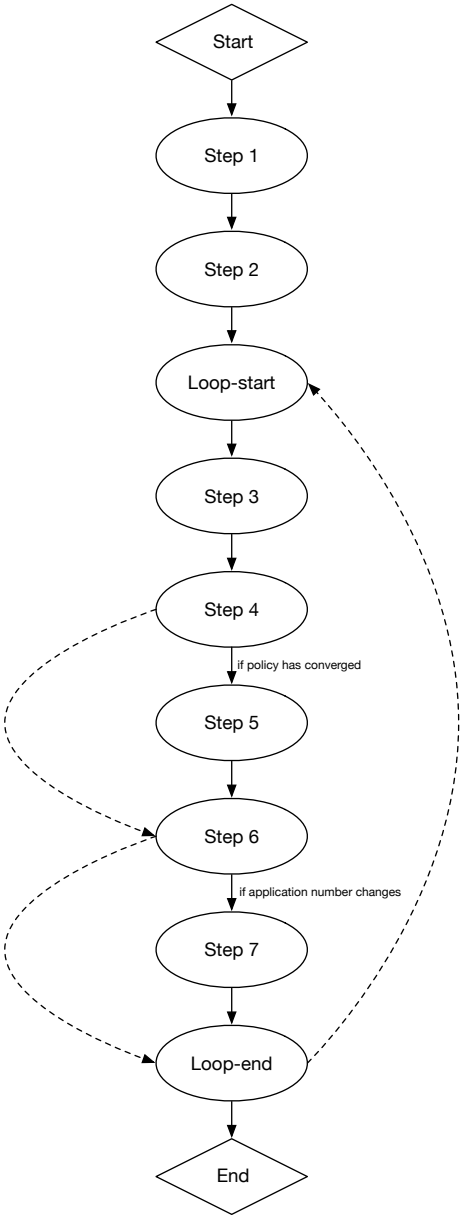


Figure 4.5: Policy workflow

Implementation 5

In this chapter we illustrate the implementation details of the workload-aware run-time resource management policy we propose in this thesis work. As stated in the previous chapters, our policy aims at guaranteeing the QoS of the application set (at throughput end), and, at the same time, reducing their power consumption by enforcing the most convenient configuration.
In Section 5.1 we show how we computed the mapping ratios for all the different combinations of big and LITTLE cores frequencies. In Section 5.2, we explain, step by step, how the policy works.

5.1 Mapping Ratio

In **??**, we proved that a proper allocation of threads on the available resources may produce improvements in terms of the application performance. The allocation was done according to the mapping ratio we found by looking at big and LITTLE cores throughput at different frequencies. In the tests we performed, we approximated such mapping ratio to 2, hence, for each thread we mapped on a LITTLE core, we mapped 2 threads on a big core. However, this approximation holds only if big frequency is set to `1900MHz` and LITTLE frequency to `1300MHz`, and, most important, it is application dependent. Therefore, we need to generalize this method in order to address all combinations of big and LITTLE cores frequency. We define the mapping ratio as the ratio between big and LITTLE cores throughput at their given frequencies. Since this ratio is a real number, it is necessary to approximate it to a

rational number. Thus, we can consider the numerator as the number of threads to allocate on each big core and the denominator as the number of threads to allocate on each LITTLE core, as the following formula suggests:

$$approximate(p_{ratio}) = \frac{numerator}{denominator} = \frac{threads_B}{threads_L}$$

When the policy starts or the application set changes, it would be necessary to compute the mapping ratio for each combination of big and LITTLE cores frequencies. However, this may be a long and intensive task, and, besides, it may steal time to the policy itself. A convenient solution is to choose a reference frequency for both big and LITTLE cores, compute a reference mapping ratio $\bar{p}_{ratio}$, and then obtain all the other mapping ratios from the reference one. Hence, we acquire $\bar{p}_{ratio}$ in this way:

$$\bar{p}_{ratio} = \frac{\overline{throughput}_B}{\overline{throughput}_L}$$

Then, a generic mapping ratio p_{ratio} may be compute as follows:

$$p_{ratio} = \frac{throughput_B}{throughput_L} = \frac{f_B}{f_L} \cdot \frac{\bar{f}_L}{\bar{f}_B} \cdot \bar{p}_{ratio}$$

This formula holds since, in first approximation, throughput scales with frequency, as we showed in Chapter 4. Finally, if the reference frequency for big and LITTLE cores is the same, the previous formula may be simplified like this:

$$p_{ratio} = \frac{f_B}{f_L} \cdot \bar{p}_{ratio}$$

This procedure allows us to steal just a small time to the policy execution, since we just need to acquire one mapping ratio, and then compute

the required mapping ratio when we find the new configuration to be actuated.
As a test, we both measured and computed the mapping ratios for each combination of frequencies. The configurations used to compute the reference mapping ratio are: $< 4, 800, 0, 800 >$ and $< 0, 800, 4, 800 >$, while the cores employed for the measurements are always 4 (for both big and LITTLE clusters). In Figure 5.1, we can see that the difference between the measured and the computed values is not meaningful (the average mean difference is -0.003, while the standard deviation of the difference is 0.009). This shows the goodness of our procedure. Finally, if we use a different number of big or LITTLE cores for the measurements, the difference between the measured and computed mapping ratios is still negligible.
The reference mapping ratio $\bar{p}_{ratio}$ is generated in Step 2 of the policy execution (Section 5.2.2), while, in Step 5, it is exploited to compute the mapping ratio of the configuration to be actuated (Section 5.2.5).

5.2 Implementation

In this section, we present of our policy design. The policy is implemented in C++ programming language on both SAVE Virtual Platform and Odroid XU3 development board.
The policy execution Steps that were briefly introduced in Section 4.2.4 are now being explained in detail by the following sections. As described in Figure 4.5, the policy steps can be summarized as follows:

Step 1) configuration table building (Section 5.2.1),

Step 2) mapping ratio generation (Section 5.2.2),

Step 3) table baseline update (Section 5.2.3),

Step 4) configuration retrieval (Section 5.2.4),

Step 5) configuration actuation (Section 5.2.5),

Step 6) power-efficient configuration retrieval (Section 5.2.6),

Step 7) configuration reset (Section 5.2.7).

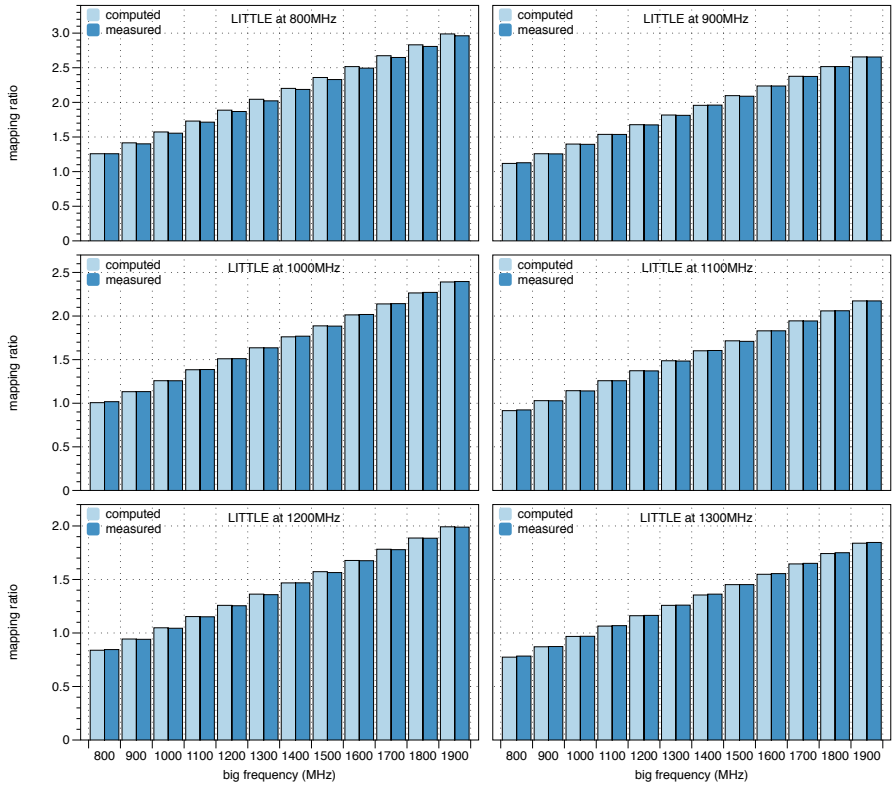


Figure 5.1: Measured and computed mapping ratios

5.2.1 Configuration Table

For our policy to work it is necessary to know the basic resources available on the underlying architecture. Hence, we first build a map that stores the data regarding the operating frequencies and the number of cores for each available family of computing resource in the system. This guarantees the scalability of our solution as the number and family of cores change. Such information are available in the SAVE Virtual Platform environment, and can be collected as well on a real system by analyzing for example *proc/cpuinfo* and the output of *cpufreq-info* command [115].
The content of this processor map is necessary to build the configuration table, i.e. a table containing all the possible combination of number of

cores (big and LITTLE) and their frequencies. As we showed in Section 4.2.1, the number of different configurations for the Exynos 5422 is 1224. As explained in Chapter 4, each configuration is associated with a power consumption value and an estimated throughput with respect to the slowest single processor execution. Such values are computed using the formulas showed in Section 4.2.
Once the resources configuration table has been built, configurations are sorted by speedup, and a first configuration has to be set for the system. We allow to configure the starting configuration point since, depending on the starting configuration the policy ends up in having different behaviors. As an example Figure 5.2 illustrates how the policy behaves picking different starting configurations in a test using PCC benchmark. Configurations are ordered by expected speedup, where `C0` is the minimum (i.e. 1 LITTLE core at `800MHz`) while `C1223` corresponds to the most powerful configuration (i.e. 4 big cores working at `1900MHz` and 4 LITTLE cores working at `1300MHz`). As we can see from the figure, `C0` converges slowly and this happens since it takes time for the policy to have a stable performance measurement from the application. A performance measurement is stable after few iterations of the application loop and, obviously, running with low performance cores increments the time needed to perform this number of iterations. On the opposite, using `C1223` forces the system to quickly change configuration if needed and this choice allows for a faster convergence than before; however this solution causes a peak in power consumption for the first iterations, even though this choice guarantees to satisfy performance requirements from the start. Finally other solutions are possible picking any configuration in the middle, basically the most the starting configuration is near the final one the faster the algorithm converges. As a rule of thumb we suggest to set either `C1223`, if paying the power overhead is not a problem, or `C612` (which is the middle configuration) because then the search space is reduced by an half. In general, since the policy is effective when the system is not overload (i.e. it does not have to work at full power) picking `C612` will help on average to have a faster convergence time.

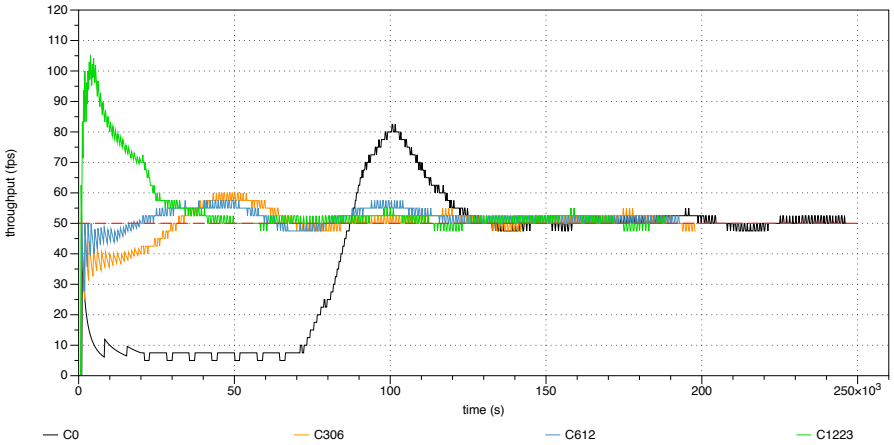


Figure 5.2: Experiments with different starting configurations.

5.2.2 Mapping Ratio Generation

The second step of the policy is the generation of the reference mapping ratio $\bar{p}_{ratio}$. During this profiling phase, we allocate one thread on each core. As stated previously, the reference configurations are $< 0, 800, 4, 800 >$ and $< 4, 800, 0, 800 >$. At first, we set the former configuration, and, before extracting the throughput, we wait the slowest application (i.e. the slowest one to produce Heartbeats) to go full speed. This is done to allow the application (or the slowest one when there are more applications) to converge in this configuration. At that point, we can retrieve the throughput of each application. Then, we set the latter configuration and repeat the procedure. Once we have throughput at both $< 0, 800, 4, 800 >$ and $< 4, 800, 0, 800 >$ configuration, we can compute the reference mapping ratio $\bar{p}_{ratio}$ for each application. This value will be used in the following configuration changes to spread properly the threads among the cores.
Now that we have both the configuration table and the reference mapping ratio $\bar{p}_{ratio}$ for each application, the very policy can start.

5.2.3 Table Baseline Update

The starting configuration we used is `C612`. Every `n` millisec the policy observes the throughput of every application (Heartbeats actually) and, every `k` measurements, the policy decides whether the system configuration has to be changed or not. The sampling period of the policy is fixed, e.g. `200ms`, while the throughput stability is application dependent.
The configuration table construction described before assumes as baseline, for the speedup estimation, the performance of a single core of the slowest family at the lowest operating frequency (i.e. the baseline is configuration $< 0, 800, 1, 800 >$). Starting from this, every time the resources configuration changes, the values of the table need to be updated. Based on the last chosen configuration, the policy knows what was the last estimated speedup for that particular configuration. We then divide every estimated speedup by the last chosen one. This means that the table now contains the values of the estimated speedups with respect to the last chosen configuration. We have hence set a new baseline.

5.2.4 Configuration Retrieval

As specified, the goal of the whole policy is to meet much performance goals as possible. Algorithm 1 reports the pseudocode for the selection of the new configuration to enforce. Given that each running application a performs differently and that it has a different goal, we have adopted the following strategy to meet the policy goal. For each application, we compute the required speedup S_a as the ratio between the current performance of the considered application T_a (obtained through online monitoring) and the declared minimum throughput requirement G_a, i.e. $S_a = T_a/G_a$. The speedup S_a is a simple way to understand, overall, how distant an application is from its performance requirement. The S_a computation can have three possible outcomes, i.e. it can be greater, equal, or less than 1. A S_a greater than 1 means that the application has not reached its goal yet, hence a speedup S_a is required. A S_a equal to 1 is the ideal condition we want to achieve, meaning that the application is performing exactly the way it should. A S_a less than 1 shows that the application is running faster than its declared goal, in this case we can switch to a slower configuration to be more power efficient. Given

that our main objective function is to maximize the number of applications that reach their goals, we choose to enforce the overall maximum required speedup whenever there has been at least a S_a greater than 1; we stick with the current configuration if all the S_a are 1; or we try to enforce the minimum slowdown required in case all the computed S_a are less than 1.
After a given speedup S_a (slowdown if $S_a < 1$) has been requested, the policy scans through the configuration table to find the configuration that can guarantee that speedup. Once we have the speedup (slowdown) required, we start looking for the closest speedup value in the configuration table. Since it would be useless to scan the whole table every time, we use two indexes (c_{Low} and c_{High}) to indicate the range of configurations we have to check. Before retrieving the new configuration, we update one of those indexes according to the speedup (slowdown) value we have just computed. In particular, if we a speedup value greater than 1, we set the c_{Low} index to the index of the last configuration. This is done because, if we now need a speedup, it means that the last configuration was too slow, and so all the configurations preceding that one, since they are sorted by speedup. On the other hand, if we have a slowdown, we set the c_{High} index to the index of the last configuration. As consequence, all the configurations following the last one are, for the moment, ignored because are too fast. In this way, the search has, on average, a logarithmic time complexity (i.e. it is $\mathcal{O}(log(n))$); hence, the policy should converge in, more or less, ten iterations. Finally, we now can retrieve the new configuration to be actuated.

5.2.5 Configuration Actuation

The configuration found contains the tuple $< N_B, f_B, N_L, f_L >$ that can be used to enforce the configuration. Starting from this configuration the policies uses the N_B and N_L parameters to enforce the affinity mask for all the running applications and then f_B and f_L to control DVFS actuators. Both the actions can be performed on SAVE Virtual Platform by proper wrapper interfaces that drives the hardware component in the simulator, while they can be implemented in a real system by either `taskset` utility or `sched_setaffinity` for the affinity mask selection,

Algorithm 1: Pseudocode for the selection of next configuration.

```
Data: c_old, c_Low, c_High, A
Result: c_new
T ← acquireApplicationsPerformance();
G ← acquireApplicationsGoal();
forall the a ∈ A do
    S_a = T_a/G_a;
end
if max(S) ≥ 1 then
    c_Low = c_t;
else
    c_High = c_t;
end
rebaseSpeedups(c_old);
forall the c ∈ (c_Low, c_High] do
    if speedup(c) > max(S) then
        c_new = c;
        break;
    end
end
```

and `cpufreq-set` for enforcing DVFS decisions. Finally, once the new configuration has been actuated, the policy generates the new mapping ratios and generates the number of threads to be mapped on big and LITTLE cores. On the SAVE Virtual Platform, we just set the total number of threads to be created (a load-balancing algorithm takes care of properly spread the workload among the cores), while, on the Odroid XU3, we transfer information about thread mapping to the applications, which allocate threads to the cores using `sched_setaffinity` command.

5.2.6 Power Efficient Configuration Retrieval

Once the policy has converged to a configuration able to satisfy the goal of each application, a power efficient configuration can now be retrieved. It is important to notice that, if the policy has converged, but there is one or more applications whose goals are not satisfied (the configuration the policy has converged to is $< 4, 1900, 4, 1300 >$), it means that, with

that goal or application set, those applications cannot be satisfied at all. In this case, we may adopt different approaches; for instance, we could decide to drop applications, starting from the furthest one from its goal, or we could continue the execution until at least one application goal is satisfied. Section 5.2.7 shows what happens when the application set changes.
In order to find the most power efficient configuration, we scan all the configurations following the chosen one and select the one with the lowest power consumption. However, we have to be aware of the fact that a following configuration may not imply high throughput, according to the results we showed in Chapter 4. Therefore, after the configuration has been chosen, we actuated it and verify whether the goals are still satisfied or not. In the latter case, the policy excludes such configuration and looks for another. In the very worst case, the policy converges to the configuration it chose before entering in this step.

5.2.7 Configuration Reset

When the application set changes, the system usage changes and so we need to reset some values before retrieving a new configuration. For this reason, we reset the indexes c_{Low} and c_{High}, we restore the speedup values in the configuration table to their default values, and we recompute the reference mapping ratio $\bar{p}_{ratio}$. Finally, the policy restarts from `C612` and repeats its execution as usual.

Experimental Results 6

In this chapter, we provide the analysis we performed on our workload-aware run-time resource management policy. In particular, at first, we tested it on SAVE Virtual Platform (Section 6.1), in both a single and multiple applications contexts. Then, we ported our policy to the Odroid XU3 development board and evaluated it using BS and PCC benchmarks (Section 6.2). Finally, we analyzed the overhead introduced by our policy (Section 6.3).

6.1 Tests on SAVE Virtual Platform

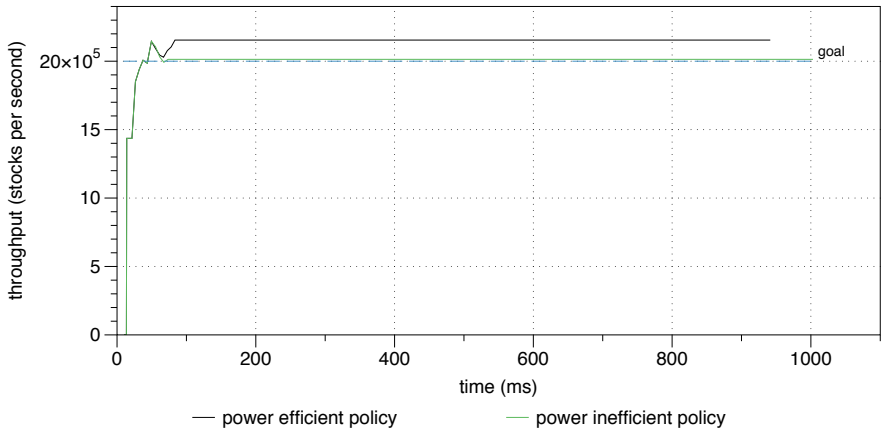
In this section, we present the tests we performed on SAVE Virtual Platform in order to prototype our workload-aware run-time resource management policy. Although SAVE Virtual Platform allows to select the underlying hardware (in our case, we chose the most similar architecture to the one mounted on Odroid XU3, i.e. 4 Samsung Exynos 5420 big cores and 4 Samsung Exynos 5420 LITTLE cores), it has some limitations. For instance, it is not possible to directly map threads on cores, but SAVE Virtual Platform exploits a load-balancing algorithm that, given `N` threads, distributes the workload as fair as possible. This means that we cannot completely test the efficiency of our thread mapping solution, since we can only enforce the number of threads we want to create. Another limitation is the fact that HMP scheduler is not implemented on SAVE Virtual Platform, hence we cannot compare our policy with it during the prototyping phase.

6.1.1 Single Application

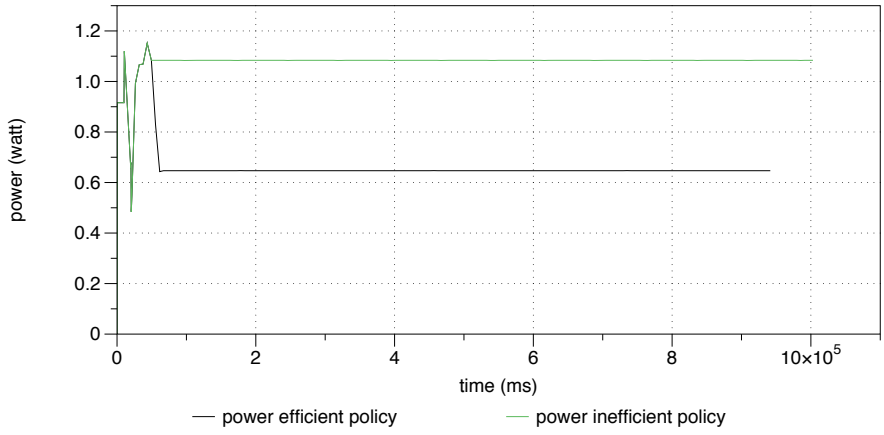
The application we considered for these tests is Black Scholes application since it is a well-balanced multi-threaded application. We ran several tests where we varied the throughput goal of Black Scholes to check the goodness of our policy.

6.1.1.1 Power efficiency test

In the first test we present, we set the minimum throughput goal of Black Scholes application to `2Msps`. In particular, we ran two different tests: the former test uses the complete policy, as described in Chapter 5 and now one called *power efficient policy*, the latter executes the policy without its *Step 7*, now on called *power inefficient policy*. In this way, we want to prove that our policy really converges to the most power efficient configuration, and satisfies the throughput goal as well.
In Figure 6.1(a), we show the throughput reached by both our policy executions. The power inefficient policy guarantees the goal by choosing `C264` ($< 1, 1400, 3, 900 >$), while power efficient policy provides an higher throughput since it is set to a higher speedup configuration, i.e. `C313` ($< 0, 800, 4, 1200 >$). On the other hand, we can notice that the configuration chosen by the power efficient policy results to be the least power-hungry configuration (Figure 6.1(b)). Indeed, `C313` turns out the be the most power efficient configuration of the ones following `C264`. Although the choice of switching to a more power efficient configuration is based on the formulas we introduced in Chapter 4, and so the computed power values may not be accurate, it still provides a convenient and reliable ordering of configurations, differently from the speedup values ordering.
This result shows that our policy is able to retrieve the best configuration, in terms of power consumption, that guarantees the required throughput.



(a) Policy throughput



(b) Policy power consumption

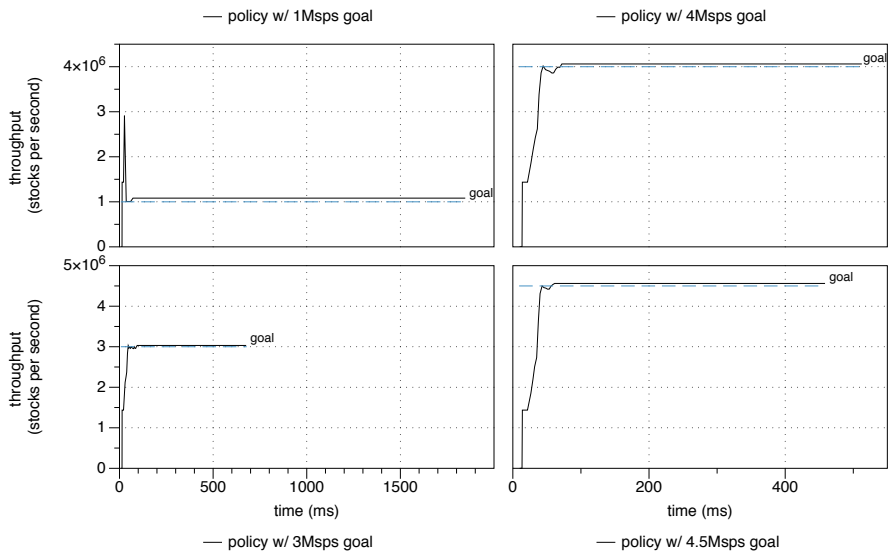
Figure 6.1: Single-app test on SAVE Virtual Platform (goal = `2Msps`)

Goal	Convergence
`1Msps`	`C42` $(< 0, 800, 3, 800 >)$
`3Msps`	`C708` $(< 2, 900, 4, 1200 >)$
`4Msps`	`C981` $(< 4, 1000, 4, 1000 >)$
`4.5Msps`	`C1124` $(< 4, 1100, 4, 1300 >)$

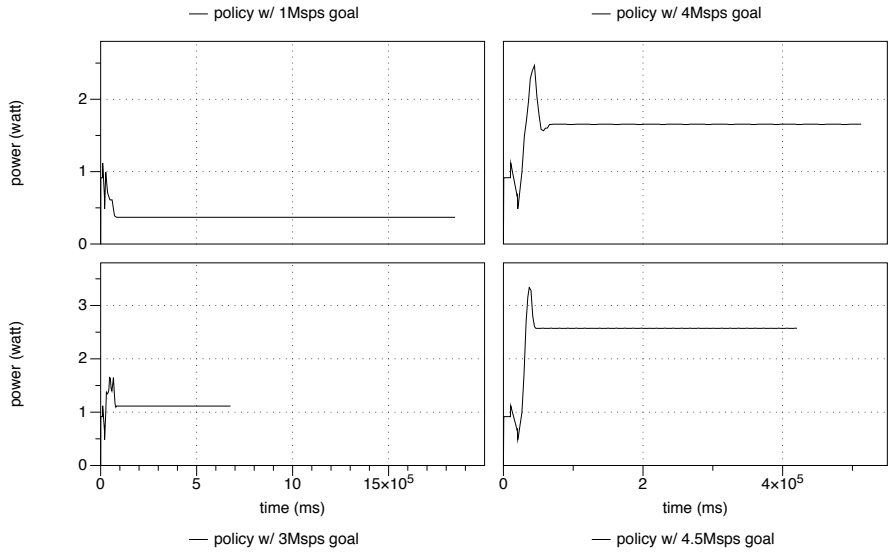
Table 6.1: Single-app other tests on SAVE Virtual Platform summary

6.1.1.2 Other tests

Here we presents all the other tests we ran on the SAVE Virtual Platform for the single application case.
In Figure 6.2(a) we can notice that our policy always satisfies the different throughput goals of Black Scholes benchmark. Table 6.1 contains the tested throughput goals and the configurations the policy converges to. In terms of power consumption, Figure 6.2(b) reports the power required by each different simulation. It is clear that the power consumption charts are not meaningful by themselves since they are not compared with other solutions. In Section 6.2, we will compare the power consumed by our policy with the power required by HMP scheduler.
These tests, among with the previous ones, demonstrate the goodness of our policy, at least during prototyping phase, when it has to deal with one single running application. In terms of throughput, the policy guarantees the QoS required by the benchmark, while, at power consumption end, these tests are not so meaningful since we only have an esteem of the real power required by the benchmark and, most important, we do not have a policy that simulate HMP scheduler. However, power consumption tests were useful to check whether our policy can select the most power efficient configuration, among the ones capable of respecting the throughput goal, or not.



(a) Policy throughput with different goals



(b) Policy power consumption with different goals

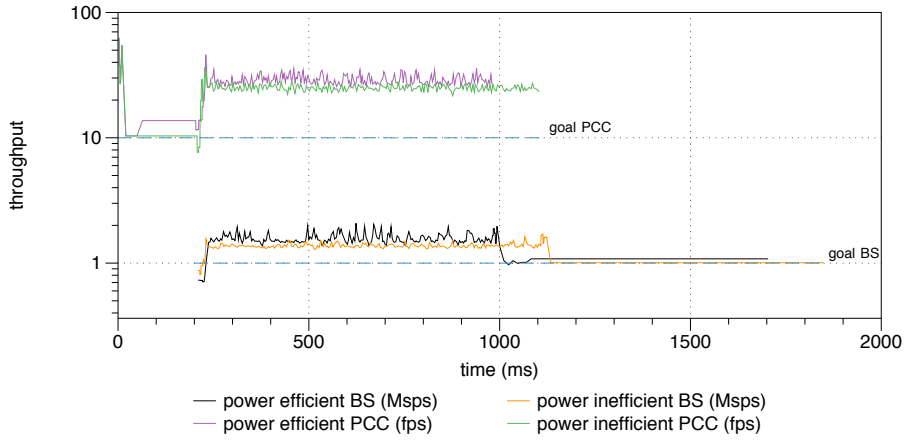
Figure 6.2: Single-app other tests on SAVE Virtual Platform

6.1.2 Multiple Applications

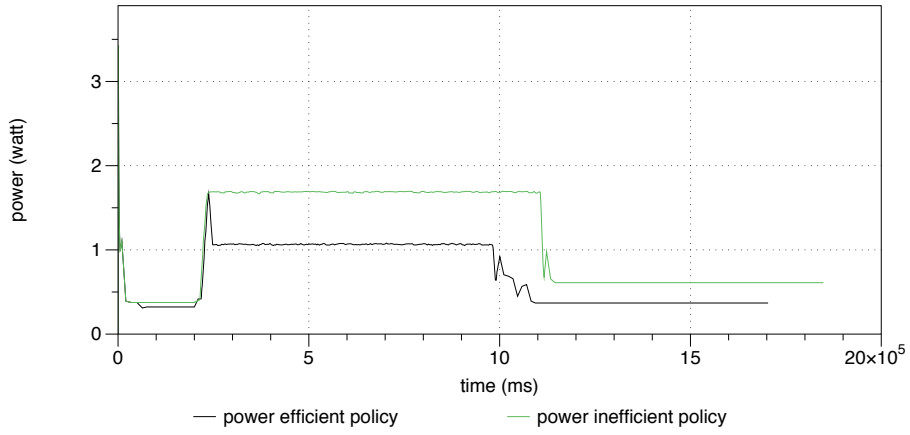
The applications we monitored for these tests are BS and PCC benchmarks. In the following tests, we first start PCC execution, then, after `200ms` (to be sure that the policy has converged), BS execution starts too. In this way, the policy execution can be split in three parts: PCC execution, BS and PCC execution, BS execution. Hence, the policy will converge to three different configurations according to the application set.
Like in single application case, we ran several tests where we varied the throughput goal of BS and PCC to check the goodness of our policy.

6.1.2.1 Power efficiency test

In the power efficiency test, we set the minimum throughput goal of BS application to `1Msps`, while PCC application goal to `10fps`. Again, we performed a test with power efficient policy and one with power inefficient policy.
Figure 6.3(a) displays the throughput curves in both cases. In the power efficient test, PCC reaches its throughput goal, and then moves to `C11` ($< 0, 800, 2, 800 >$). Once BS execution starts, the policy converges to a new configuration able to satisfy both PCC and BS goals, i.e. `C667` ($< 2, 800, 4, 1200 >$). After PCC end, the policy converges to power efficient configuration `C42` ($< 0, 800, 3, 800 >$), which is the same configuration BS converged to in single application case. On the other hand, using power inefficient policy, throughput goals are still reached; indeed, policy converges, respectively, to `C6` ($< 0, 800, 1, 1200 >$), `C613` ($< 2, 1500, 3, 900 >$) and `C32` ($< 1, 900, 1, 1100 >$). However, if we look at power consumption chart in Figure 6.3(b), the power efficiency of both tests are quite different. Indeed, we can notice that power efficient policy, at first, converges to the same configuration of power inefficient policy, but then it moves to a more power efficient configuration, in all three parts of policy execution (PCC only, BS and PCC, BS only).
This test proves that, also in a multiple applications context, our policy is able to retrieve the best configuration, in terms of power consumption.



(a) BS and PCC throughput



(b) Policy power consumption

Figure 6.3: Multi-apps test on SAVE Virtual Platform

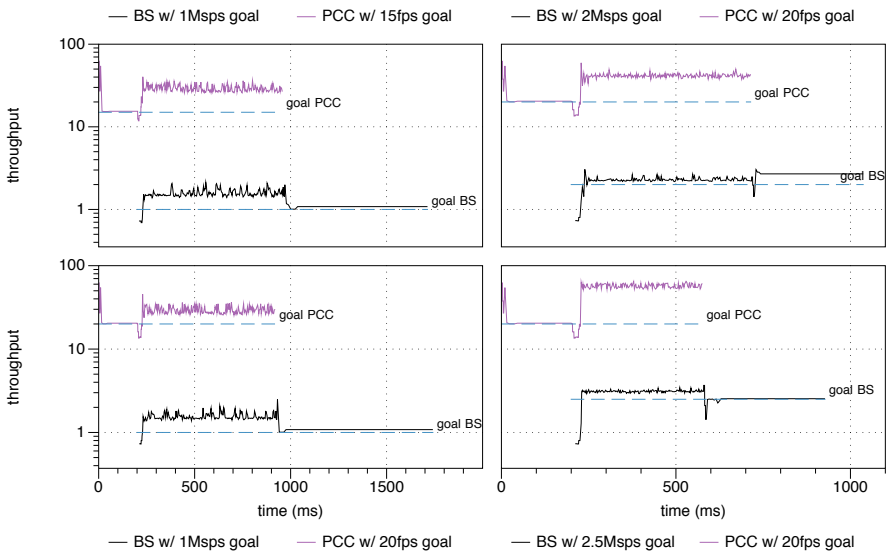
(goal = `1Msps` and `10fps`)

Goal	App	Convergence
15fps	PCC	C14 $(< 0, 800, 2, 900 >)$
1Msps - 15fps	BS - PCC	C667 $(< 2, 800, 4, 1200 >)$
1Msps	BS	C42 $(< 0, 800, 3, 800 >)$
20fps	PCC	C42 $(< 0, 800, 3, 800 >)$
1Msps - 20fps	BS - PCC	C667 $(< 2, 800, 4, 1200 >)$
1Msps	BS	C42 $(< 0, 800, 3, 800 >)$
20fps	PCC	C42 $(< 0, 800, 3, 800 >)$
2Msps - 20fps	BS - PCC	C1049 $(< 4, 900, 4, 1300 >)$
2Msps	BS	C667 $(< 2, 800, 4, 1200 >)$
20fps	PCC	C42 $(< 0, 800, 3, 800 >)$
3Msps - 20fps	BS - PCC	C1223 $(< 4, 1900, 4, 1300 >)$
3Msps	BS	C708 $(< 4, 1100, 4, 1300 >)$

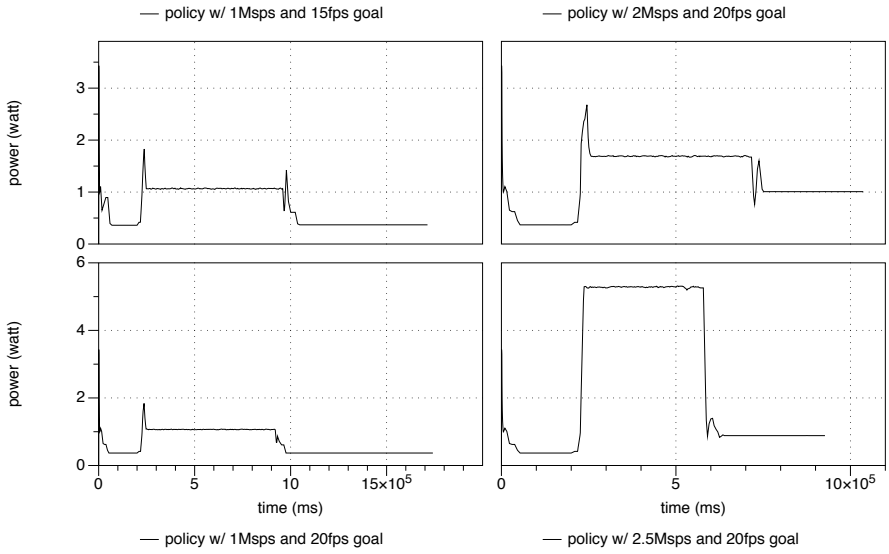
Table 6.2: Multi-apps other tests on SAVE Virtual Platform summary

6.1.2.2 Other tests

Here we report all the other tests we ran on the SAVE Virtual Platform for the multiple applications case.
In Figure 6.4(a) we show that our policy always guarantees the QoS required by both BS and PCC applications. Table 6.2 contains the tested throughput goals and the configurations the policy converges to, in each of the three execution parts. In terms of power consumption, Figure 6.4(b) reports an overestimation of the power that our policy, along with BS and PCC may require during its execution.
Like in the single application case, we demonstrate the efficiency of our policy during the simulations we performed on SAVE Virtual Platform. Now that both single and multiple applications cases have been tested, we can port our policy on the Odroid XU3 development board and test our solution on a real system, and, most important, compare its results with HMP scheduler.



(a) BS and PCC throughput with different goals



(b) Policy power consumption with different goals

Figure 6.4: Multi-apps other tests on SAVE Virtual Platform

6.2 Tests on Odroid XU3

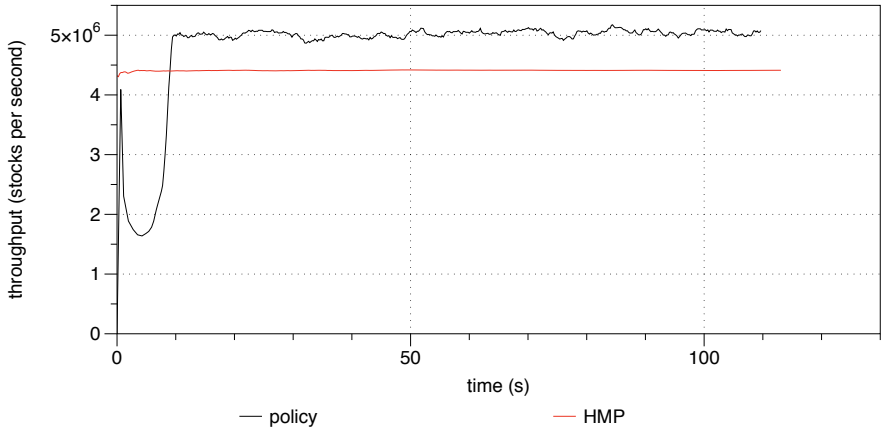
After that, the policy has been tested on SAVE Virtual Platform in both single and multiple applications cases, we can test it on a real system. In this section, we present the tests we ran using our policy on the Odroid XU3 development board. In this way, we can overcome the limitations imposed by the SAVE Virtual Platform, like the direct thread mapping on cores (so far, we relied on the load-balancing algorithm available on the SAVE Virtual Platform). Therefore, we now can completely test each part of our workload-aware run-time resource management policy. Moreover, here it is finally possible to compare the behavior of our policy with HMP scheduler. Indeed, in each of the tests we performed, we also ran an execution of the same benchmark using HMP and imposing the same configuration our policy converged to. This is done because we cannot impose a throughput goal to HMP scheduler, hence we need to execute the application using the same resources employed by our policy. In this way, performance, in terms of throughput and power consumption, of both our policy and HMP scheduler can be properly compared. As result, we can prove whether our policy is capable of satisfying application throughput requirements and, at the same time, reduce the power consumption.
Finally, in these tests we have to consider that, when we compare our policy and HMP scheduler, a power consumption comparison makes sense only if both solutions satisfy the application goal, otherwise, even though it may still be interesting, such comparison turns out to be less meaningful. The primary goal of our policy is the fulfillment of the application throughput goal. Once such requirement is respected, the policy can focus on its secondary goal; i.e. power efficiency.

6.2.1 Single Application

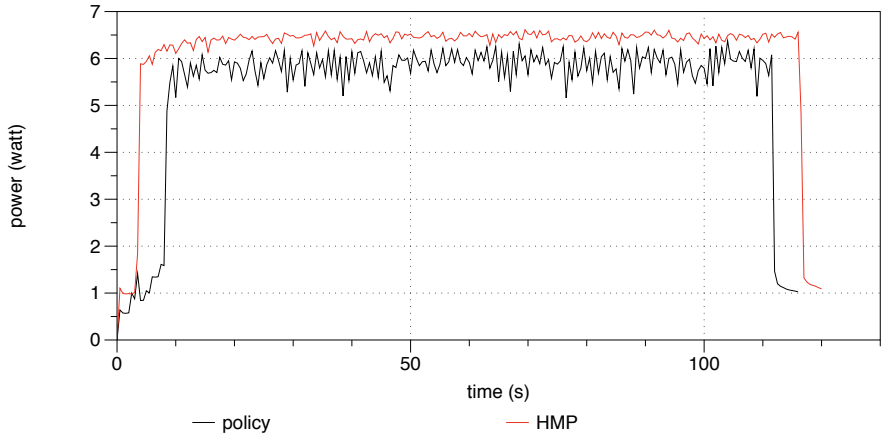
The application we considered for the tests on Odroid XU3 development board is, again, Black Scholes application, for the reasons stated before. For these tests, we set the number of Black Scholes kernel executions to 500, while each kernel computes 1000000 options. Like in the tests on SAVE Virtual Platform, we performed several tests varying the throughput goal of Black Scholes benchmark.

6.2.1.1 Full speed test

In this test, we compare the maximum throughput that can be provided by our policy and HMP scheduler at full speed, i.e. in `C1223` ($< 4, 1900, 4, 1300 >$). We set a sufficiently high goal in order to let the policy converge to `C1223`, while we directly enforced such configuration, using `taskset` command, when we used HMP scheduler.
In Figure 6.5(a), it is clear that our policy is definitely more efficient than HMP (in terms of throughput), since, thanks to thread mapping, our policy is able of reaching an average throughput of `5Msps`. On the other hand, HMP scheduler throughput is slightly smaller than `4.5Msps`. Therefore, our policy can satisfy throughput goals that HMP scheduler cannot, due to a not completely efficient load distribution. Finally, our policy takes `10s` to converge to `C1223`.
In terms of power consumption, Figure 6.5(b) demonstrates that the results reported by SAVE Virtual Platform were not correct; indeed, those power values were overestimating the power consumption of policy. On Odroid XU3, we can truly measure the power consumed by both solutions. As result, not only the power required by our policy is smaller than the one estimated by SAVE Virtual Platform, but it also turns out to be more power efficient than HMP scheduler.
Finally, we compared our policy and HMP with a *throughput/power* (`sps/watt`) metric. Differently from throughput/energy (`sps/joule`) metric, this one is not execution depended, i.e. it does not take into consideration execution time. Figure 6.5(c) proves that our policy is more efficient than HMP also under this metric. HMP execution in this test may be considered identical to a generic HMP execution of BS application; indeed, also in the latter case, HMP spreads the workload among all the available resources. Hence, we will use this HMP throughput-power ratio value as a criterion for comparison in the next tests. In this way, we compare the results of our policy with an usual HMP execution.
This test demonstrated that our policy results to perform better than HMP scheduler at both throughput and power consumption ends, thanks to an efficient workload distribution among available resources.

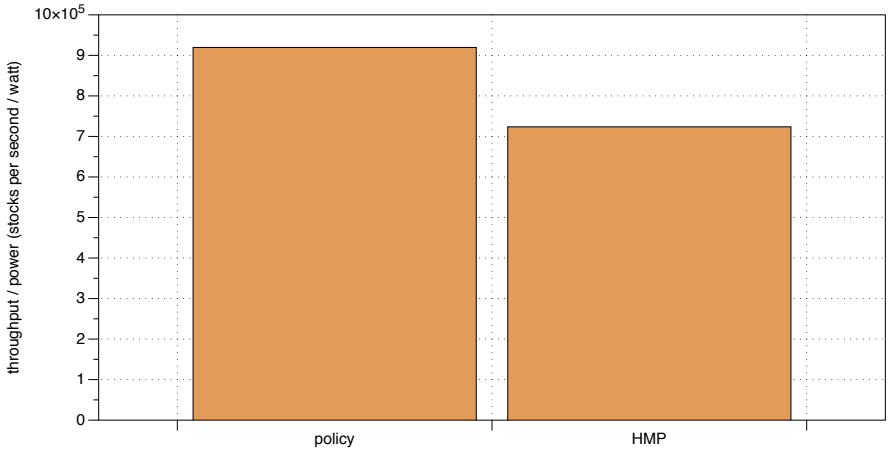


(a) Full speed policy and HMP throughput



(b) Full speed policy and HMP power consumption

Figure 6.5: Single-app full speed test on Odroid XU3



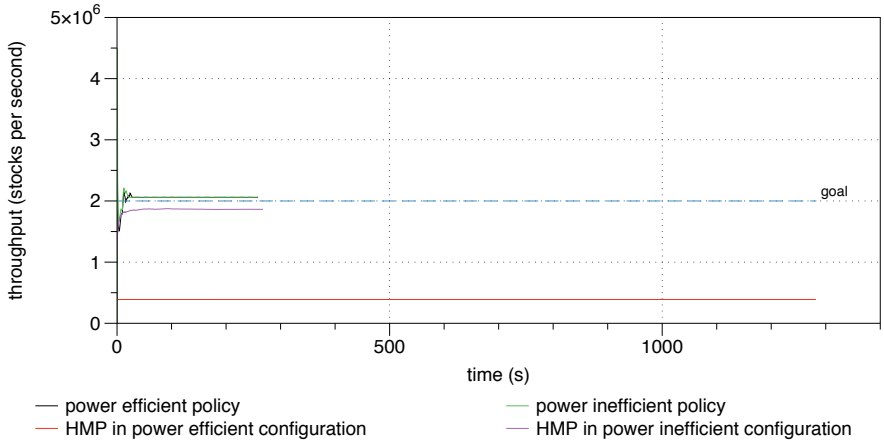
(c) Full speed policy and HMP throughput-power ratio

Figure 6.5: Single-app full speed test on Odroid XU3

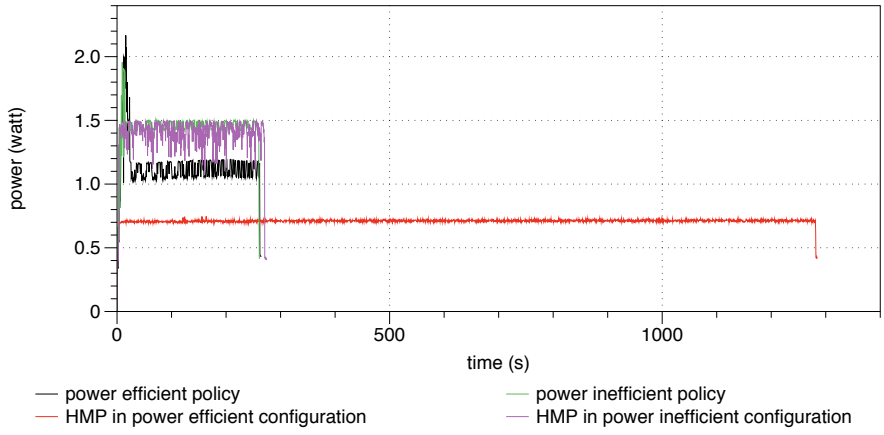
6.2.1.2 Power efficiency test

Like we did for the tests on SAVE Virtual Platform, we first want to check whether our policy finds the most power efficient configuration, once the throughput goal is satisfied. The minimum throughput goal of Black Scholes application is set to `2Msps`, and we ran two different tests, one using the power efficient policy, the other using the power inefficient policy. Then, for each test, we extracted the configurations the policy converged to and executed Black Scholes with HMP scheduler enforcing such configurations.
In Figure 6.6(a), we report the throughput reached by both policies execution and HMP scheduler. The power efficient policy converges to `C516` ($< 1, 900, 4, 1200 >$), which is a Critical Configuration for HMP, while the power inefficient policy converges to `C496` ($< 3, 900, 2, 1200 >$). Both policies guarantee the QoS required by Black Scholes, whereas neither of HMP executions are capable of satisfy such goal. In particular, HMP execution in `C516` is the one that has the lowest throughput (almost `0.4Msps`). This result is due to the fact that `C516` is a Critical Configuration, hence HMP does not spread the workload properly, but executes all the computation on the big core only. HMP execution in

C496 has certainly better throughput performance, but it still cannot respect Black Scholes goal. This means that our thread mapping solution effectively efficient, since it allows our policy to reach throughput values higher than HMP scheduler.
In terms of power consumption, HMP at C516 turns out to be the most power efficient execution. Actually, this is a consequence of HMP behavior in Critical Configurations, because HMP employs only the big core, while the LITTLE cores are in idle state. Hence, it cannot be considered as an effective power efficient solution, since it does not satisfy the throughput goal either. Excluding HMP at C516, the power efficient policy results to be the less power-hungry solution. On the other hand, after an initial transient, the power required by our power inefficient policy and HMP at C496 is very similar, our policy consumes slightly more power.
Figure 6.6(c) compares our policy and HMP using throughput/power metric. Not only both policy executions result to be more efficient than the respective HMP executions, but they also definitely surpass full speed HMP.
Finally, while power inefficient policy takes 9s (3% of whole execution time) to converge, power efficient one requires 19s (7%), since it has to find a power efficient configuration and test whether it satisfies the throughput goal.



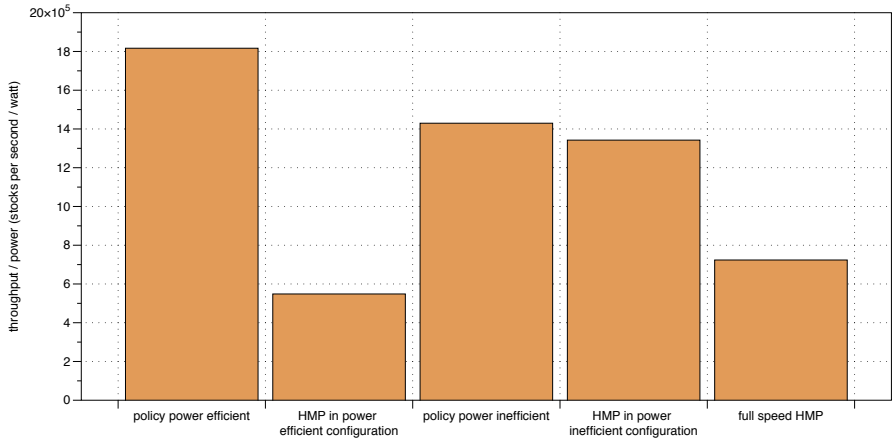
(a) Policy and HMP throughput



(b) Policy and HMP power consumption

Figure 6.6: Single-app test on Odroid XU3

(goal = `2Msps`)



(c) Policy and HMP throughput-power ratio

Figure 6.6: Single-app test on Odroid XU3

(goal = `2Msps`)

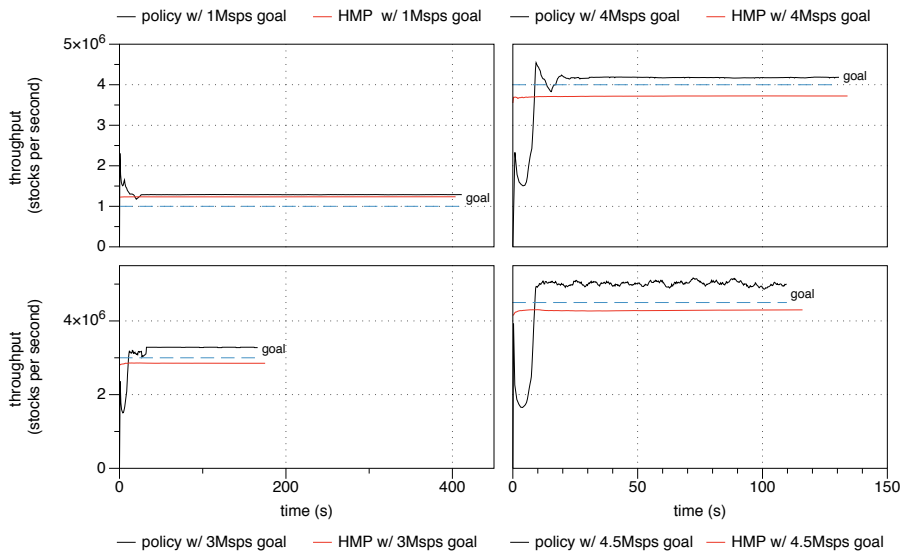
6.2.1.3 Other tests

Here we presents all the other tests we ran on Odroid XU3 development board for the single application case.
Figure 6.7(a) shows that our policy always satisfies the throughput goal of Black Scholes on the four tested cases (Table 6.3 contains the tested throughput goals, the configurations the policy converges to, and the convergence times). On the other hand, HMP scheduler is able to respect the goal only in one configuration (`C106`), where the system is actually used as a homogeneous architecture, since no big cores are employed. In all the other configurations, HMP does not guarantee the minimum QoS.
Figure 6.7(b) reports the chart power consumption charts for each test. In `C106` and `C967`, after an initial transient, the power required by our policy and HMP is almost identical, while, in `C1192` and `C1222`, our policy consumes definitely less power than HMP scheduler.
Figure 6.7(c) shows that, in terms of throughput-power ratio, our policy is as efficient as HMP in `1Msps` and `3Msps` test cases, whereas it is more efficient than HMP in the other two cases. Moreover, each execution of

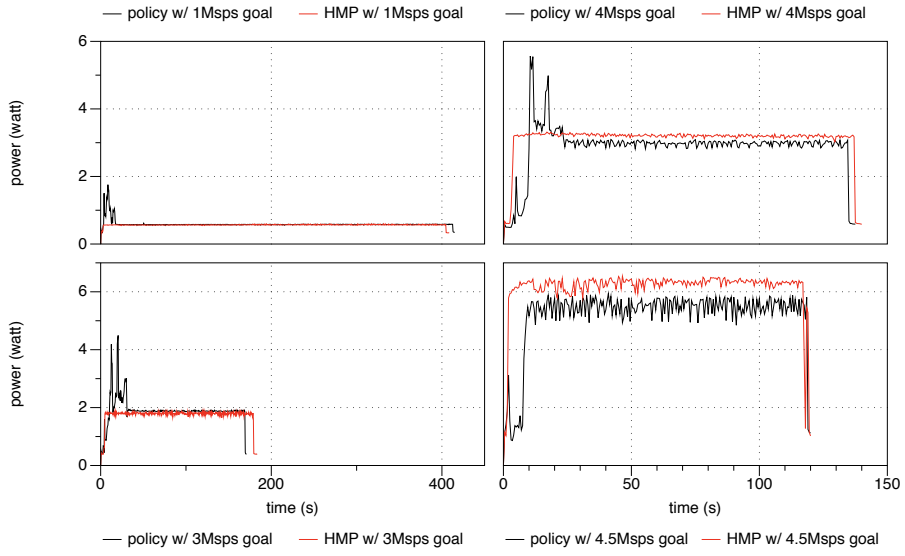
Goal	Convergence	Time (s)
1Msps	C106 ($< 0, 800, 4, 800 >$)	12 (3%)
3Msps	C967 ($< 4, 900, 4, 1100 >$)	27 (16%)
4Msps	C1192 ($< 4, 1400, 4, 1300 >$)	19 (15%)
4.5Msps	C1222 ($< 4, 1900, 4, 1200 >$)	5 (5%)

Table 6.3: Single-app other tests on Odroid XU3 summary

our policy is more throughput-power efficient than full speed HMP.
In terms of convergence time, in the worst case, our policy requires 27s, i.e. 16% of BS execution time, while, in the best case, it takes 3%.
These tests, like the previous ones, confirm the tests we performed on SAVE Virtual Platform and the quality of our policy, in a single application case. Most important, these tests demonstrate the goodness of our thread mapping solution, which is capable of increasing performance (at throughput and power consumption ends) since it distributes the workload among the cores in more efficiently than HMP scheduler. In this way, configurations that would not guarantee the minimum QoS using HMP, may be employed to both respect the throughput goal and reduce power consumption. Finally, each of our policy executions resulted to be more throughput-power efficient than full speed HMP execution, which, as we stated, it may be considered as a regular and usual HMP execution in a multi-threaded context.

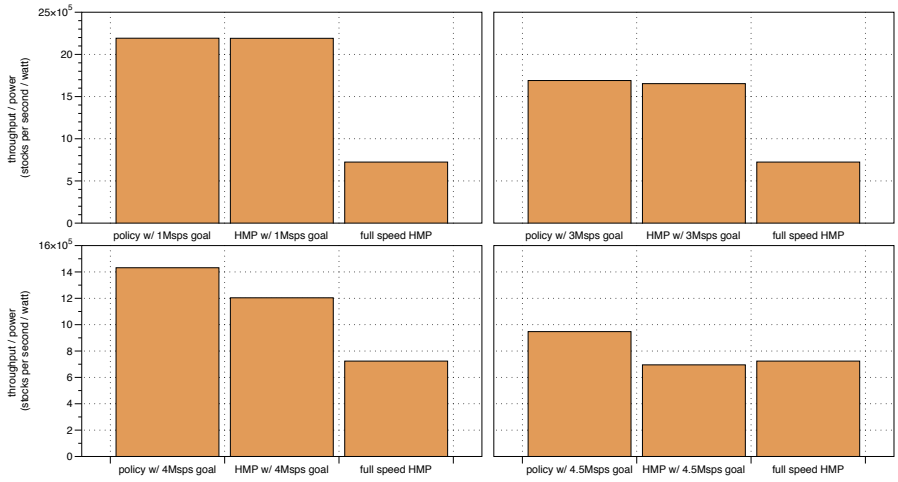


(a) Policy and HMP throughput with different goals



(b) Policy and HMP power consumption with different goals

Figure 6.7: Single-app other tests on Odroid XU3 summary



(c) Policy and HMP throughput-power ratio with different goals

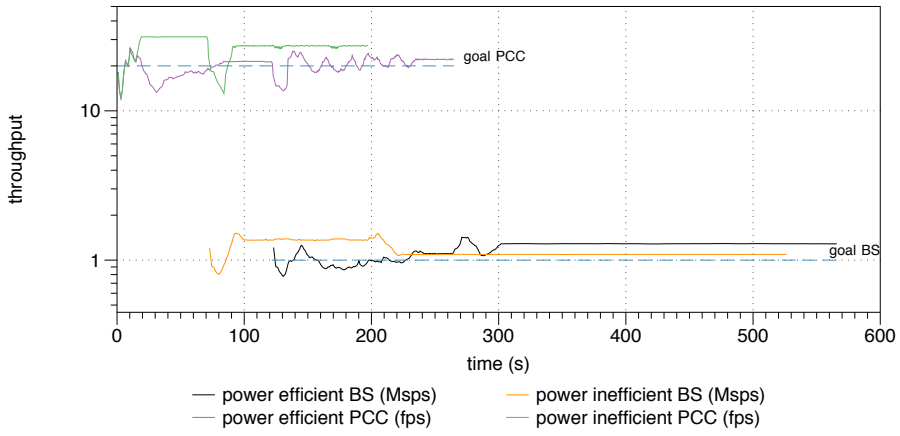
Figure 6.7: Single-app other tests on Odroid XU3

6.2.2 Multiple Applications

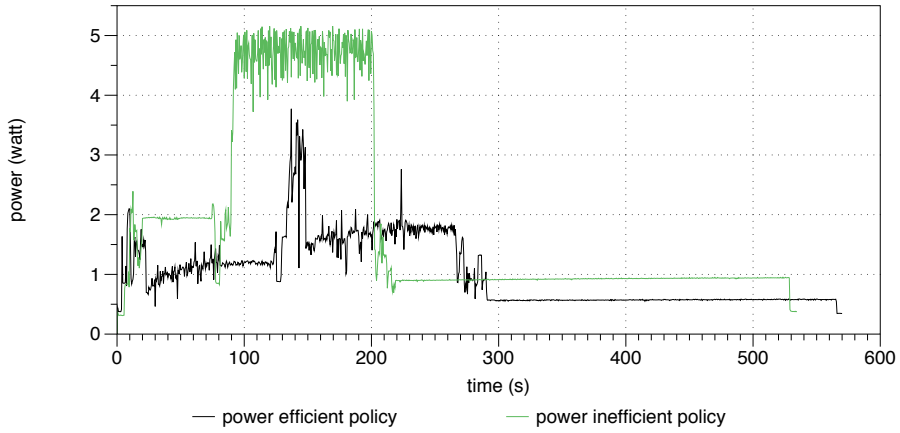
Like for the multiple applications tests on SAVE Virtual Platform, we monitored BS and PCC benchmarks. Both the considered applications are multi-threaded, but PCC is not well-balanced. In fact, PCC benchmark on SAVE Virtual Platform was designed to be more balanced and stable than its actual implementation. Therefore, our policy may take more time to converge, in particular when a power efficient configuration has to be found; indeed, the fact that our speedup formula is not 100% accurate, along with PCC features, may force our policy to try different power efficient configuration before retrieving the right one, in terms of both throughput and power efficiency.
In the following tests, we first start PCC execution, then, after PCC convergence, BS execution starts too. Again, the policy execution can be split in three parts: PCC execution, BS and PCC execution, BS execution.

6.2.2.1 Power efficiency test

Here we present the power efficiency test for multiple applications case. BS application goal is `1Msps`, while PCC goal is `20fps`.
Figure 6.8(a) shows the throughput curves in both cases. In the power efficient test, PCC reaches its throughput goal, and then, after trying different configuration, it eventually converges to `C442` ($< 1, 900, 4, 1100 >$). Once BS execution starts, the policy converges to an initial configuration able to satisfy both both, and then looks for a power efficient one. The policy converges to `C825` ($< 3, 800, 4, 1200 >$). Finally, after PCC end, BS easily converges to its power efficient configuration, i.e. `C106` ($< 0, 800, 4, 800 >$). On the contrary, the power inefficient policy converges after few iterations to a configuration able to guarantee PCC QoS, i.e. `C300` ($< 3, 1000, 1, 900 >$); then, once BS starts its execution, the policy converges again quite quickly to a new configuration capable of satisfying PCC and BS goals `C811` ($< 3, 1800, 1, 900 >$). Finally, when there is only BS running, the policy converges to `C78` ($< 2, 800, 1, 900 >$).
In terms of power consumption, as reported in Figure 6.8(b), power efficient policy, at first, has a behavior similar to power inefficient one, but then it moves to power efficient configurations, although the policy does not converge immediately to a configuration able to satisfy both power efficiency and QoS, as a consequence of PCC unbalanced behavior.
PCC convergence time and, in particular, its percentage with respect to PCC execution time only may not be as interesting as the other values since it depends on the time instant we decided to start BS execution. Power efficient policy takes `77s` to converge to `C442` (63% of PCC execution time). After BS starts its execution, the policy converges in `105s` (73% of BS and PCC execution time together). Finally, after PCC end, the policy converges in `23s` (7% of BS execution time). On the other hand, power inefficient policy does not look for a power efficient configuration, hence, when only PCC is running, it converges in `13s` (18%). Then, as BS begins, the policy converges again in `15s` (12%). Finally, the policy with only BS running converges in `13s` (4%).
This test proves that, also in a multiple applications context, our policy is able to converge to a configuration that both guarantees the QoS and is power efficient.



(a) BS and PCC throughput



(b) Policy power consumption

Figure 6.8: Multi-apps test on Odroid XU3

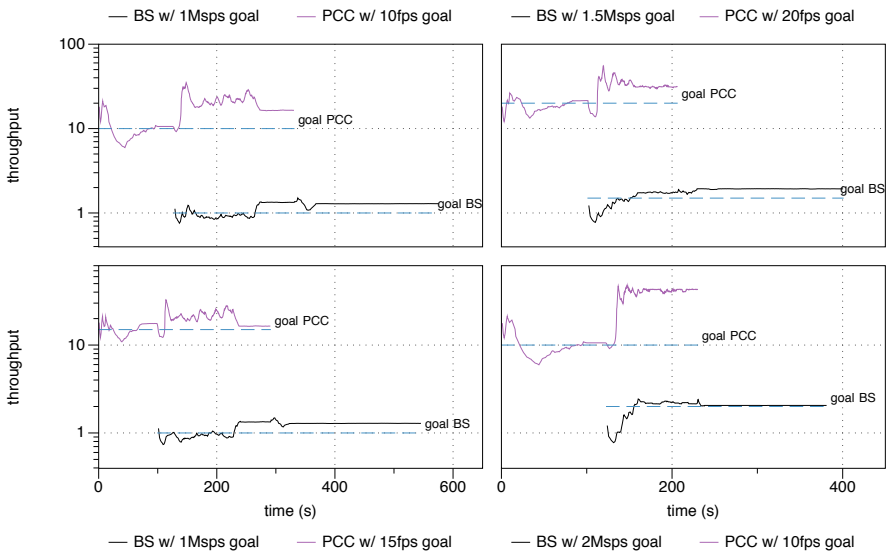
(goal = `1Msps` and `20fps`)

Goal	App	Convergence	Time
10fps	PCC	C20 $(< 1, 800, 1, 1000 >)$	86s (67%)
1Msps - 10fps	BS - PCC	C727 $(< 2, 1100, 4, 1100 >)$	136s (67%)
1Msps	BS	C106 $(< 0, 800, 4, 800 >)$	22s (9%)
15fps	PCC	C286 $(< 1, 800, 4, 900 >)$	65s (66%)
1Msps - 15fps	BS - PCC	C727 $(< 2, 1100, 4, 1100 >)$	127s (67%)
1Msps	BS	C106 $(< 0, 800, 4, 800 >)$	18s (7%)
20fps	PCC	C442 $(< 1, 900, 4, 1100 >)$	72s (71%)
1.5Msps - 20fps	BS - PCC	C1066 $(< 4, 1200, 3, 1300 >)$	48s (45%)
1.5Msps	BS	C313 $(< 0, 800, 4, 1200 >)$	12s (6%)
10fps	PCC	C20 $(< 1, 800, 1, 1000 >)$	86s (70%)
2Msps - 10fps	BS - PCC	C1213 $(< 4, 1900, 3, 1200 >)$	31s (29%)
2Msps	BS	C516 $(< 1, 900, 4, 1200 >)$	30s (20%)

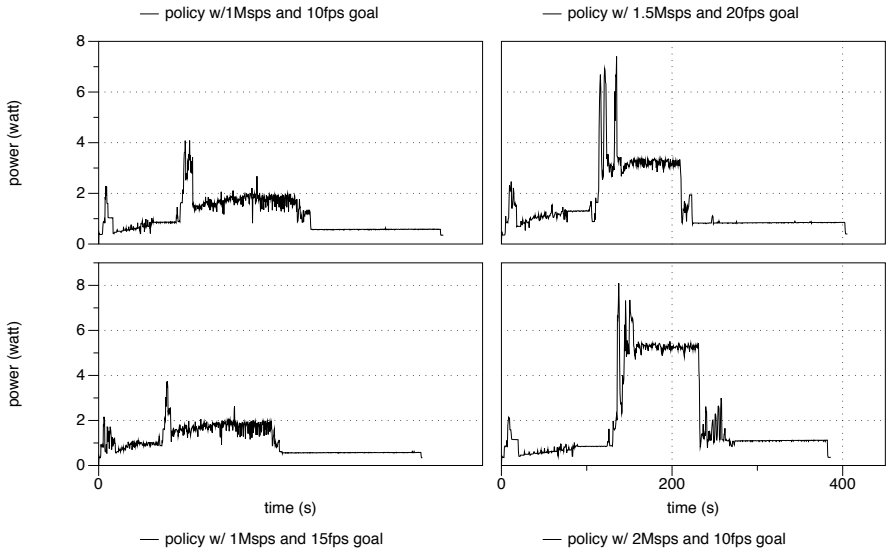
Table 6.4: Multi-apps other tests on Odroid XU3 summary

6.2.2.2 Other tests

Here we presents all the other tests we ran on Odroid XU3 development board for the multiple applications case.
Figure 6.9(a) shows that our policy always satisfies the throughput goal of BS and PCC on the four tested cases, although it does not converge to a power efficient configuration as fast as it did with only BS benchmark (Table 6.4 contains the tested throughput goals, the configurations the policy converges to, and the convergence times)
Figure 6.9(b) reports the chart power consumption charts for each test. In all the cases, we can notice that, after an initial transient where the policy converges to a suitable configuration, the power consumption decreases since the possible power efficient configurations are tested in order to find one able to also guarantee QoS of PCC and BS.
Finally, in terms of convergence times, when PCC is running, our policy always takes more time than the time needed to converge when only BS is running. As stated previously, this is a consequence of PCC unbalanced nature.



(a) BS and PCC throughput with different goals



(b) Policy and HMP power consumption with different goals

Figure 6.9: Multi-apps other tests on Odroid XU3

6.3 Overhead Analysis

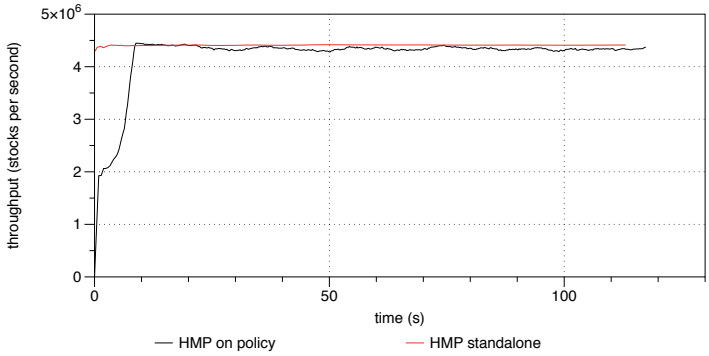
So far, we have reported the tests performed on our policy on both SAVE Virtual Platform and Odroid XU3 development board. In this section, we analyze the overhead introduced by our policy. More specifically, we compare the performance of full speed HMP standalone execution, already reported in Section 6.2.1.1, with HMP on policy execution. To achieve such goal, we modified our policy in order to remove task allocation and allow HMP to distribute the workload. We ran BS benchmark with a sufficient high throughput goal, so that the policy quickly converges to `C1223`.

Figure 6.10(a) displays HMP standalone and HMP on policy throughput. We can notice that, while HMP standalone throughput results to be quite stable, HMP on policy throughput is slightly unstable, due to the interaction with the policy. HMP standalone average throughput is `4.41Msps`, whereas HMP on policy average throughput is `4.26Msps`.

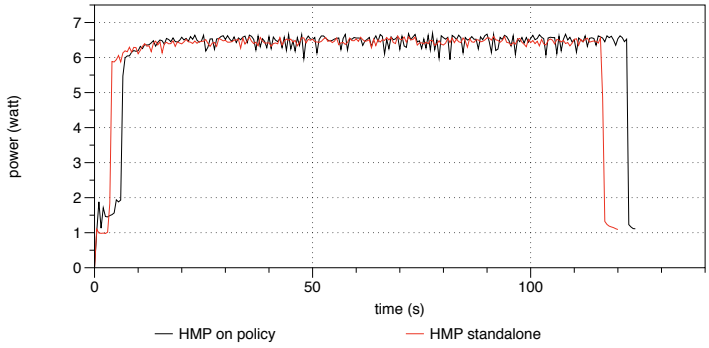
Figure 6.10(b) reports the power consumption curves. On average, HMP standalone consumes `6.09W`, while HMP on policy `6.12W`.

Finally, we compared HMP standalone with HMP on policy in terms of throughput-power ratio metric, as showed in Figure 6.10(c). As result, HMP standalone throughput-power ratio is higher than HMP on policy (`0.72Msps/W` and `0.69Msps/W`, respectively).

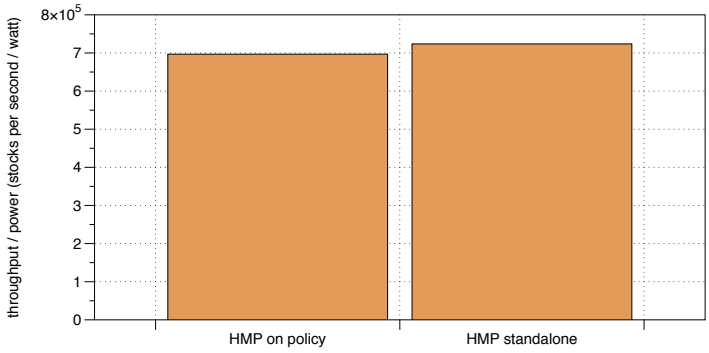
As we expected, our policy introduces an overhead in terms of throughput and power consumption. The policy overhead results in a 3.5% throughput loss, while, in terms of power consumption, the overhead is 0.5%. Therefore, the power consumption overhead is negligible, whereas the throughput loss is not so relevant but it does not affect our policy execution since, as showed in Section 6.2.1.1, the task allocation solution is able to provide higher throughput performance than HMP scheduler.



(a) Throughput overhead



(b) Power consumption overhead



(c) Throughput-power ratio overhead

Figure 6.10: Policy overhead

Conclusions 7

This chapter reviews the work done within this thesis and derives the conclusions from what has been presented so far. Section 7.1 discusses the contributions of this thesis, while Section 7.2 analyzes the limitations of the proposed work. Finally, Section 7.3 presents some possible future works and enhancements starting from this thesis.

7.1 Contributions

Considering the state of the art in chapter 3, this work provides several contributions. A first contribution of this thesis is that it propose a global task scheduling able to distribute a multi-threaded application among all the available cores. Indeed, the works we presented in the state of the art, employ either a cluster migration or a core migration scheduler, while only HMP, the state of the art scheduler for ARM big.LITTLE architecture, is able to spread the computation across the available heterogeneous resources. However, HMP has no procedure that controls and ensures the the requirements of an application, in particular in terms of throughput. Our work is the first one that proposes a policy capable of achieving such result, and, at the same time, employing all the computing cores available on the underlying architecture.
Another contribution of this work is the fact it results to provide high performance with respect to HMP scheduler. In Chapter 6 we proved that, in the same configuration, our policy is more throughput and power efficient than HMP, thanks to a properer workload distribution and thread assignment. Moreover, in the tests we performed, our policy re-

sulted to be also more efficient in terms or throughput per watt. Indeed, all our policy tests were compared with full speed HMP throughput per watt value and our policy always surpassed HMP results.
Finally, this thesis exposed HMP misbehavior in the so called Critical Configurations. This was an interesting discovery, because it was the starting point for our task allocation solution.

7.2 Limits of the present work

In spite of the contributions brought by this work, there are still some aspects that need to be improved.
First of all, the speedup formula we exploited in this work has to be inspected and enhanced. Such formula holds as we deal with either big or LITTLE cluster, but, when we deal with mixed configurations (i.e. configurations employing both big and LITTLE cores), it does not hold anymore, or, at least, is not as accurate as in the single cluster case. This is due to the fact that, in mixed configurations the throughput, as well as the speedup, does not remain linear and, most important, monotone, as we proved in Chapter 4. Hence, one configuration that is supposed to provide an higher throughput value, actually has worse performance. The motivation for such a behavior is the synchronization between big and LITTLE clusters at different frequencies. Also for this reason, we introduced the concept of mapping a convenient number of threads on big and LITTLE cores, to reduce the waiting time between threads on different clusters. However, the speedup formula does not take into account either the synchronization time or the mapping ratio of mixed configurations, and, as result, it is not accurate in mixed configurations. Another limitation due to the speedup formula is the different time the policy requires to converge when not perfectly balanced applications, like PCC, are running. We showed in Chapter 6 that, when PCC was running, the policy took more time to converge to a power efficient configuration, since some of the tested configurations did not provide as throughput as supposed.
Of course, the speedup formula proposed in this work aims to be as general as possible, hence, as consequence, it may have drawbacks when the application is not well-balanced.

7.3 Future work

One of the future works is the improvement of speedup formula, in order to make it more robust and accurate. Such formula may be enhanced in different ways; for instance, it could include information about the applications are running, like their mapping ratios, or the application speedups may be extracted only once when it is run for the first time, stored in memory, and then retrieved when that application is executed again.
Another possible future work is to expand the policy for other heterogeneous resources like GPUs and FPGAs. In fact, Ordroid XU3 development board is also powered by ARM Mali GPU, but it was not employed in this work. It would be interesting to include Mali GPU as an alternative to big and LITTLE cores. This extension will require some changes in the policy; for instance, the policy has to be aware whether the running application has a GPU implementation or not, and GPU power consumption has to be taken into account to retrieve a suitable power efficient configuration. In this way, the policy may turn into a more general resource manager for heterogeneous system, able to allocate a task on a convenient resource according to different aspects like available resources, available implementation for that particular resource and application requirements.

List of abbreviations

Glossary

ACA Asynchronous Clock Architecture. 58

ACE AXI Coherency Extensions. 12

API Application Programming Interface. 30, 31, 58, 59

AXI Advanced eXtensible Interface. 12

BS Black Scholes. 29, 30, 39, 47, 52, 56, 58, 75, 80, 82, 85, 91, 93, 94, 96, 98

CCI Cache Coherent Interconnect. 8, 9, 12

CPU Central Processing Unit. i, ii, 3, 7, 12–14, 16–19, 22–24, 26–28, 30, 52, 58

DVFS Dynamic Voltage and Frequency Scaling. i, ii, 2, 5, 14, 15, 19, 21–26, 28, 54, 60, 62, 72, 73

EAS Energy-Aware Scheduling. 27

FPGA Field Programmable Gate Array. i, ii, 3, 7, 17, 52, 103

GIC Generic Interrupt Controller. 9

GPU Graphic Processing Unit. i, ii, 3, 7, 16–18, 22, 23, 52, 103

GTS Global Task Scheduling. 12, 14

HMP Heterogeneous Multi-Processing. 5, 6, 14–16, 19, 21, 22, 28–30, 32, 33, 36, 39–41, 44, 47, 48, 58, 75, 78, 82, 84, 85, 87, 88, 90, 91, 98, 101, 102

HPC High Performance Computing. 3, 4, 17, 22

HSA Heterogeneous System Architecture. i, 2–5, 7, 8, 17, 18, 21, 24, 25, 28, 51

IP Intellectual Property. 8
IPC Instructions Per Cycle. 53
ISA Instruction Set Architecture. i, ii, 4, 9, 25

OS Operating System. 13, 14, 16
OTG On-The-Go. 16

PCC Pearson Correlation Coefficient. 52, 56, 58, 69, 75, 80, 82, 93, 94, 96, 102
PoP Package on Package. 16

QoS Quality of Service. i, ii, 4, 5, 19, 21, 25, 26, 51, 63, 65, 78, 82, 87, 90, 91, 94, 96

RAM Random Access Memory. 16
RTM Reverse Time Migration. 52

SAVE Self-Adaptive Virtualisation-Aware High-Performance/Low-Energy Heterogeneous System Architectures. 5, 6, 17–19, 55, 67, 68, 72, 73, 75, 78, 82, 84, 85, 87, 91, 93, 98
SLA Service Level Agreement. 17
SMP Symmetric Multi-Processing. 14, 15
SoC System on Chip. 4, 8, 16
sps stocks per second. 32, 49
SSA Stochastic Simulation algorithm. 52

TLM Transaction Level Modeling. 18

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff." *IEEE Solid-State Circuits Newsletter*, vol. 20, no. 3, pp. 33 – 35, 2006.

[2] R. H. Dennard, V. Rideout, E. Bassous, and A. Leblanc, "Design of ion-implanted mosfet's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, 1974.

[3] Intel, "Microprocessor Quick Reference Guide." [Online]. Available: http://www.intel.com/pressroom/kits/quickreffam.htm

[4] Intel, "The Story of the Intel 4004." [Online]. Available: http://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html

[5] Intel, "Intel Xeon Processor E5-2699 v3." [Online]. Available: http://ark.intel.com/products/81061/Intel-Xeon-Processor-E5-2699-v3-45M-Cache-2_30-GHz

[6] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[7] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar, "Critical power slope: Understanding the runtime effects of frequency scaling," in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS '02. New York, NY, USA: ACM, 2002, pp. 35–44. [Online]. Available: http://doi.acm.org/10.1145/514191.514200

[8] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi *et al.*, "The design and implementation of a first-generation cell processor-a multi-core soc," in *Integrated Circuit Design and Technology, 2005. ICICDT 2005. 2005 International Conference on.* IEEE, 2005, pp. 49–52.

[9] J. Parkhurst, J. Darringer, and B. Grundmann, "From single core to multi-core: Preparing for a new exponential," in *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '06. New York, NY, USA: ACM, 2006, pp. 67–72. [Online]. Available: http://doi.acm.org/10.1145/1233501.1233516

[10] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[11] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, May 2005.

[12] Intel, "6th Generation Intel Core i5 Processors." [Online]. Available: http://www.intel.com/content/www/us/en/processors/core/core-i5-processor.html

[13] Intel, "6th Generation Intel Core i7 Processors." [Online]. Available: http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html

[14] AMD, "AMD A-Series APU Processors." [Online]. Available: http://www.amd.com/en-us/products/processors/desktop/a-series-apu

[15] Intel, "Intel Xeon Processor E7 Family." [Online]. Available: http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e7-family.html

[16] IBM, "IBM Power Systems." [Online]. Available: http://www-03.ibm.com/systems/power/index.html

[17] Oracle, "Oracle SPARC Systems." [Online]. Available: https://www.oracle.com/servers/sparc/index.html

[18] "HSA Fundation." [Online]. Available: http://www.hsafoundation.com

[19] T. Sterling and D. Stark, "A high-performance computing forecast: Partly cloudy," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 42–49, 2009. [Online]. Available: http://scitation.aip.org/content/aip/journal/cise/11/4/10.1109/MCSE.2009.111

[20] S. Garg, S. Gopalaiyengar, and R. Buyya, "Sla-based resource provisioning for heterogeneous workloads in a virtualized cloud datacenter," in *Algorithms and Architectures for Parallel Processing*, ser. Lecture Notes in Computer Science, Y. Xiang, A. Cuzzocrea, M. Hobbs, and W. Zhou, Eds. Springer Berlin Heidelberg, 2011, vol. 7016, pp. 371–384. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24650-0_32

[21] H. Viswanathan, E. Lee, I. Rodero, D. Pompili, M. Parashar, and M. Gamell, "Energy-aware application-centric vm allocation for hpc workloads," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, May 2011, pp. 890–897.

[22] "TOP500 List - June 2015." [Online]. Available: http://www.top500.org/lists/2015/06/

[23] Intel, "Intel Xeon Phi 31S1P coprocessor." [Online]. Available: http://ark.intel.com/products/79539/Intel-Xeon-Phi-Coprocessor-31S1P-8GB-1_100-GHz-57-core

[24] "Green500 List - June 2015." [Online]. Available: http://www.green500.org/news/green500-list-june-2015?q=lists/green201506

[25] Intel, "Haswell cpu." [Online]. Available: http://ark.intel.com/products/codename/42174/Haswell#@All

[26] PEZY Computing, "PEZY-SC Many Core Processor." [Online]. Available: http://pezy.co.jp/en/products/pezy-sc.html

[27] ARM, "ARM big.LITTLE Technology." [Online]. Available: http://www.arm.com/products/processors/technologies/biglittleprocessing.php

[28] GSMArena, "Samsung Galaxy S4 Specs." [Online]. Available: http://www.gsmarena.com/samsung_i9500_galaxy_s4-5125.php

[29] GSMArena, "Samsung Galaxy S5 Specs." [Online]. Available: http://www.gsmarena.com/samsung_galaxy_s5-6033.php

[30] GSMArena, "Samsung Galaxy S6 Specs." [Online]. Available: http://www.gsmarena.com/samsung_galaxy_s6-6849.php

[31] GSMArena, "Specs." [Online]. Available: http://www.gsmarena.com/lg_g4-6901.php

[32] GSMArena, "Specs." [Online]. Available: http://www.gsmarena.com/htc_one_m9-6891.php

[33] "EU SAVE Project." [Online]. Available: http://www.fp7-save.eu

[34] "Save Virtual Platform Simulator." [Online]. Available: save.vp.necst.it

[35] Hardkernel co., "Odroid XU3." [Online]. Available: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127

[36] AMD, "What is Heterogeneous System Architecture (HSA)?" [Online]. Available: http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/

[37] AMD, "Heterogeneous System Architecture: A Technical Review." [Online]. Available: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/hsa10.pdf

[38] "ARM website." [Online]. Available: http://www.arm.com

[39] ARM, "big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7 (Improving Energy Efficiency in High-Performance Mobile Platforms)."

[40] ARM, "big.LITTLE Technology: The Future of Mobile." [Online]. Available: http://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf

[41] Samsung, “Benefits of the big.LITTLE Architecture.” [Online]. Available: http://www.samsung.com/global/business/semiconductor/minisite/Exynos/data/benefits.pdf

[42] “Samsung website.” [Online]. Available: http://www.samsung.com

[43] Samsung, “Exynos 5 octa - exynos 5422.” [Online]. Available: http://www.samsung.com/global/business/semiconductor/minisite/Exynos/w/solution.html#?v=octa_5422

[44] ARM, “Cortex-A15 Processor.” [Online]. Available: http://www.arm.com/products/processors/cortex-a/cortex-a15.php

[45] ARM, “Cortex-A7 Processor.” [Online]. Available: http://www.arm.com/products/processors/cortex-a/cortex-a7.php

[46] ARM, “CoreLink CCI-400 Cache Coherent Interconnect.” [Online]. Available: http://www.arm.com/products/system-ip/interconnect/corelink-cci-400.php

[47] ARM, “CoreLink Multi-Cluster CPU Interrupt Controllers.” [Online]. Available: http://www.arm.com/products/system-ip/controllers/interrupt.php

[48] ARM, “Introduction to AMBA 4 ACE and big.LITTLE Processing Technology.” [Online]. Available: http://www.arm.com/files/pdf/CacheCoherencyWhitepaper_6June2011.pdf

[49] ARM, “AMBA AXI and ACE Protocol Specification.” [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022e/index.html

[50] M. Larabel, “Linux 4.3 Scheduler Change "Potentially Affects Every SMP Workload In Existence".” [Online]. Available: http://phoronix.com/scan.php?page=news_item&px=Linux-4.3-Scheduler-SMP

[51] “Ubuntu Operating System.” [Online]. Available: http://www.ubuntu.com

[52] "Android Operating System." [Online]. Available: https://www.android.com

[53] "Lubuntu Operating System." [Online]. Available: http://lubuntu.net

[54] K. Yu, D. Han, C. Youn, S. Hwang, and J. Lee, "Power-aware task scheduling for big. LITTLE mobile processor," in *SoC Design Conference (ISOCC), 2013 International*. IEEE, 2013, pp. 208–212.

[55] Samsung, "Heterogeneous Multi Processing Solution of Exynos 5 Octa with ARM big.LITTLE Technology." [Online]. Available: http://www.arm.com/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf

[56] Samsung, "Exynos 5 Octa: Heterogeneous Multi-Processing Capability." [Online]. Available: http://www.samsung.com/global/business/semiconductor/minisite/Exynos/w/mediacenter.html#?v=blog_Exynos_5_Octa_Heterogeneous_Multi_Processing_Capability

[57] Samsung, "Samsung primes exynos 5 octa for arm big.little technology with heterogeneous multi-processing capability." [Online]. Available: http://www.samsung.com/global/business/semiconductor/minisite/Exynos/w/mediacenter.html#?v=news_Samsung_Primes_Exynos5Octa_for_ARM_bigLITTLE_Technology_with_Heterogeneous_Multi_Processing_Capability

[58] Samsung, "Exynos 5 octa - exynos 5420." [Online]. Available: http://www.samsung.com/global/business/semiconductor/minisite/Exynos/w/solution.html#?v=octa_5420

[59] Samsung, "Exynos 5 octa - exynos 5410." [Online]. Available: http://www.samsung.com/global/business/semiconductor/minisite/Exynos/w/solution.html#?v=octa_5410

[60] ARM, "Mali-T628 GPU." [Online]. Available: http://www.arm.com/products/multimedia/mali-performance-efficient-graphics/mali-t628.php

[61] G. Durelli, M. Coppola, K. Djafarian, G. Kornaros, A. Miele, M. Paolino, O. Pell, C. Plessl, M. D. Santambrogio, and C. Bolchini, "Save: Towards efficient resource management in heterogeneous system architectures," in *Reconfigurable Computing: Architectures, Tools, and Applications*. Springer, 2014, pp. 337–344.

[62] G. C. Durelli, M. Pogliani, A. Miele, C. Plessl, H. Riebler, M. D. Santambrogio, G. Vaz, and C. Bolchini, "Runtime resource management in heterogeneous system architectures: The save approach," in *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 142–149.

[63] "CORDIS - Seventh Framework Programme." [Online]. Available: http://cordis.europa.eu/fp7/home_en.html

[64] A. R. Mieli, G. C. Durelli, M. D. Santambrogio, and C. Bolchini, "A System-Level Simulation Framework for Evaluating of Resource Management Policies for Heterogeneous System Architectures," in *Proceedings of the 18th IEEE/Euromicro Conference On Digital System Design (DSD 2015)*. IEEE Computer Society Press, 2015.

[65] Accelera Systems Initiative, "SystemC." [Online]. Available: http://www.accellera.org

[66] S. Mittal, "A survey of techniques for improving energy efficiency in embedded computing systems," *International Journal of Computer Aided Engineering and Technology*, vol. 6, no. 4, pp. 440–459, 2014.

[67] Y. Abe, H. Sasaki, M. Peres, K. Inoue, K. Murakami, and S. Kato, "Power and Performance Analysis of GPU-Accelerated Systems," in *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems*. Berkeley, CA: USENIX, 2012. [Online]. Available: https://www.usenix.org/conference/hotpower12/workshop-program/presentation/Abe

[68] NVIDIA, "Geforce gtx 480." [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications

[69] R. Ge, R. Vogt, J. Majumder, A. Alam, M. Burtscher, and Z. Zong, "Effects of Dynamic Voltage and Frequency Scaling on a K20 GPU," in *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ser. ICPP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 826–833. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2013.98

[70] Intel, "Sandy bridge cpu." [Online]. Available: http://ark.intel.com/products/codename/29900/Sandy-Bridge#@All

[71] NVIDIA, "K20c kepler." [Online]. Available: http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf

[72] X. Mei, L. S. Yung, K. Zhao, and X. Chu, "A Measurement Study of GPU DVFS on Energy Conservation," in *Proceedings of the Workshop on Power-Aware Computing and Systems*, ser. HotPower '13. New York, NY, USA: ACM, 2013, pp. 10:1–10:5. [Online]. Available: http://doi.acm.org/10.1145/2525526.2525852

[73] V. Spiliopoulos, A. Bagdia, A. Hansson, P. Aldworth, and S. Kaxiras, "Introducing DVFS-Management in a Full-System Simulator," in *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium on*, Aug 2013, pp. 535–545.

[74] "The gem5 simulator." [Online]. Available: http://www.gem5.org/Main_Page

[75] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "MultiScale: Memory System DVFS with Multiple Memory Controllers," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '12. New York, NY, USA: ACM, 2012, pp. 297–302. [Online]. Available: http://doi.acm.org/10.1145/2333660.2333727

[76] K. Choi, R. Soma, and M. Pedram, "Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, no. 1, pp. 18–28, 2005.

[77] S. Saewong and R. Rajkumar, "Practical voltage-scaling for fixed-priority rt-systems," in *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE.* IEEE, 2003, pp. 106–114.

[78] V. Kianzad, S. S. Bhattacharyya, and G. Qu, "Casper: an integrated energy-driven approach for task graph scheduling on distributed embedded systems," in *Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE International Conference on.* IEEE, 2005, pp. 191–197.

[79] P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins, "Managing dynamic concurrent tasks in embedded real-time multimedia systems," in *Proceedings of the 15th international symposium on System Synthesis.* ACM, 2002, pp. 112–119.

[80] M. Pricopi, T. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*, Sept 2013, pp. 1–10.

[81] S. Yoo, "An empirical validation of power-performance scaling: DVFS vs. multi-core scaling in big.LITTLE processor," *IEICE Electronics Express*, vol. 12, no. 8, pp. 20 150 236–20 150 236, 2015.

[82] C. Imes and H. Hoffmann, "Minimizing Energy Under Performance Constraints on Embedded Platforms: Resource Allocation Heuristics for Homogeneous and single-ISA Heterogeneous Multi-cores," *SIGBED Rev.*, vol. 11, no. 4, pp. 49–54, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2724942.2724950

[83] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite Cores: Pushing Heterogeneity Into a Core," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 317–328. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.37

[84] F. Gaspar, A. Ilic, P. Tomas, and L. Sousa, "Performance-Aware Task Management and Frequency Scaling in Embedded Systems," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, Oct 2014, pp. 65–72.

[85] Hardkernel co., "Odroid XU+E." [Online]. Available: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G137463363079

[86] T. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical power management for asymmetric multi-core in dark silicon era," in *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, May 2013, pp. 1–9.

[87] ARM, "Versatile Express." [Online]. Available: http://www.arm.com/products/tools/development-boards/versatile-express

[88] M. Kim, K. Kim, J. Geraci, and S. Hong, "Utilization-aware load balancing for the energy efficient operation of the big.LITTLE processor," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014, pp. 1–4.

[89] S. Holmback, S. Lafond, and J. Lilius, "Performance Monitor Based Power Management for big.LITTLE Platforms," in *Proceedings of the HIPEAC Workshop on Energy Efficiency with Heterogeneous Computing*, D. Nikolopoulos and J. L. Nunez-Yanez, Eds. HiPEAC, 2015, pp. 1–6.

[90] Morten Rasmussen, "Energy cost model for energy-aware scheduling." [Online]. Available: https://lkml.org/lkml/2014/7/3/884

[91] Mike Turquette, "Summary of Energy-Aware Scheduling workshop, Linux Kernel Summit 2014." [Online]. Available: https://www.linaro.org/blog/core-dump/summary-energy-aware-scheduling-workshop-linux-kernel-summit-2014/

[92] Nicolas Pitre, "The Road to Energy-Aware Scheduling." [Online]. Available: https://www.linaro.org/blog/core-dump/road-energy-aware-scheduling/

[93] Amit Kucheria, "Energy-Aware Scheduling (EAS) Project." [Online]. Available: https://www.linaro.org/blog/core-dump/energy-aware-scheduling-eas-project/

[94] "CPU frequency scaling." [Online]. Available: https://wiki.archlinux.org/index.php/CPU_frequency_scaling

[95] "CPUidle." [Online]. Available: http://www.cpuidle.de

[96] F. Black and M. Scholes, "The Pricing of Options and Corporate Liabilities," *Journal of Political Economy*, vol. 81, no. 3, pp. pp. 637–654, 1973. [Online]. Available: http://www.jstor.org/stable/1831029

[97] "cpufreq-set command." [Online]. Available: http://linux.die.net/man/1/cpufreq-set

[98] "taskset command." [Online]. Available: http://linux.die.net/man/1/taskset

[99] "htop - an interactive process viewer for Linux." [Online]. Available: http://hisham.hm/htop/

[100] "sched_setaffinity command." [Online]. Available: http://linux.die.net/man/2/sched_setaffinity

[101] "OpenMP." [Online]. Available: http://openmp.org

[102] C. Jacobi, "Ueber eine neue auflösungsart der bei der methode der kleinsten quadrate vorkommenden lineären gleichungen," *Astronomische Nachrichten*, vol. 22, no. 20, pp. 297–306, 1845.

[103] D. T. Gillespie, "Exact stochastic simulation of coupled chemical reactions," *The journal of physical chemistry*, vol. 81, no. 25, pp. 2340–2361, 1977.

[104] D. T. Gillespie, "Approximate accelerated stochastic simulation of chemically reacting systems," *The Journal of Chemical Physics*, vol. 115, no. 4, pp. 1716–1733, 2001.

[105] J. Barnes and P. Hut, "A hierarchical o (n log n) force-calculation algorithm," 1986.

[106] E. Baysal, D. D. Kosloff, and J. W. Sherwood, "Reverse time migration," *Geophysics*, vol. 48, no. 11, pp. 1514–1524, 1983.

[107] K. Pearson, "Note on Regression and Inheritance in the Case of Two Parents," *Proceedings of the Royal Society of London*, vol. 58, pp. pp. 240–242, 1895. [Online]. Available: http://www.jstor.org/stable/115794

[108] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application Heartbeats for Software Performance and Health," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010, pp. 347–348. [Online]. Available: http://doi.acm.org/10.1145/1693453.1693507

[109] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "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*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 79–88. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809065

[110] F. Sironi, D. B. Bartolini, S. Campanoni, F. Cancare, H. Hoffmann, D. Sciuto, and M. D. Santambrogio, "Metronome: Operating System Level Performance Management via Self-adaptive Computing," in *Proc. of Design Automation Conference*, 2012, pp. 856–865.

[111] “stress command.” [Online]. Available: http://linux.die.net/man/1/stress

[112] NVIDIA, “CUDA Parallel Computing Platform.” [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[113] “CFS Scheduler.” [Online]. Available: https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt

[114] W. Mauerer, *Professional Linux Kernel Architecture*. Birmingham, UK, UK: Wrox Press Ltd., 2008.

[115] “cpufreq-info command.” [Online]. Available: http://linux.die.net/man/1/cpufreq-info