UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

TESIS DOCTORAL

Energy-Efficient Resource Management for Task-based Parallel Applications in Multi-Aplication Environments

Gestión de recursos energéticamente eficiente para aplicaciones paralelas basadas en tareas en entornos multi-aplicación

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Luis María Costero Valero

DIRECTORES

Francisco Daniel Igual Peña
Katzalin Olcoz Herrero
Francisco Tirado Fernández

Energy-Efficient Resource Management for Task-based Parallel Applications in Multi-Application Environments

Gestión de Recursos Energéticamente Eficiente para Aplicaciones Paralelas Basadas en Tareas en Entornos Multi-Aplicación

Ph.D. Thesis

Luis María Costero Valero

Supervised by: Francisco Daniel Igual Peña
Katzalin Olcoz Herrero
Francisco Tirado Fernández

Facultad de Informática
Universidad Complutense de Madrid

December 2020

Energy-Efficient Resource Management for Task-based Parallel Applications in Multi-Application Environments

---

Gestión de Recursos Energéticamente Eficiente para Aplicaciones Paralelas Basadas en Tareas en Entornos Multi-Aplicación

Memoria que presenta para optar al título de Doctor en Ingeniería Informática

Luis María Costero Valero

Dirigida por los Doctores:

Francisco Daniel Igual Peña
Katzalin Olcoz Herrero
Francisco Tirado Fernández

Facultad de Informática
Universidad Complutense de Madrid

Diciembre 2020

This thesis has been mainly supported by the Spanish MECD under grant No. FPU15/02050.

This thesis has been partially supported by the EU (FEDER) and Spanish MINECO (GA No. TIN2015-65277R, GA No. RTI2018-093684-B-I00), the Spanish CM (GA No. S2018-TC-4423), the EC H2020 MANGO project (GA No. 671668), and the HiEPAC network.

A mi familia, directores
y todos los que me han
acompañado en este camino

Agradecimientos

Cuando Katza y Fran me ofrecieron aquella pequeña colaboración el último año de carrera, poco o nada me imaginaba que aquello iba a desembocar en esta tesis 6 años después. Durante estos años muchas formidables personas han pasado a mi lado, y que de una u otra manera, se han ganado mis más sinceros agradecimientos.

En primer lugar, para mis directores Fran, Katza y Paco, por su apoyo durante todos estos años, por ser una fuente infinita de conocimiento e ideas a explorar, y en resumen, por ser unos grandes investigadores y mejores personas.

También me gustaría dar las gracias a David y Marina por recibirme en la EPFL, así como todos sus consejos e ideas que siempre nos han llevado a buen puerto. Y por supuesto a toda la gente de ESL que coincidió conmigo y que hicieron que aquellos 6 meses se me pasaran volando.

A toda la gente de ArTECS, compañeros y amigos, por su acogida y siempre buen ambiente. A la gente del laboratorio, por sus buenos momentos y absurdas discusiones. A Mercedes, Lucía y Ana, a David y Félix, técnicos y resto de personal de administración y servicios de ambas facultades. Y por supuesto a toda la gente del aula 16, que han vivido esta aventura conmigo desde el principio. Y muchas gracias a Pablo y Miguel, que llevamos juntos 10 años en esta aventura que nunca pensamos que ibamos a embarcarnos. Y por supuesto muchas gracias a Jesús, que ha aguantado todas mis quejas y sufrimientos durante estos años, y ha sido un pilar fundamental de esta tesis.

A mi familia y amigos, por su apoyo incondicional y paciencia, por estar ahí siempre.

Y por último, a toda esa gente que de una u otra manera han compartido conmigo una parte de este camino y no he mencionado.

A todos vosotros, mil gracias.

Madrid, 31 de agosto de 2020

Resumen

Gestión de Recursos Energéticamente Eficiente para Aplicaciones Paralelas Basadas en Tareas en Entornos Multi-Aplicación

El fin del escalado de Dennard, así como la llegada de la era *post-Moore* ha supuesto una gran revolución en la forma de obtener el rendimiento y eficiencia energética en los procesadores modernos. Desde un incremento constante en la frecuencia de reloj como la principal forma de aumentar el rendimiento a principios de los 2000, el incremento del número de núcleos dentro del procesador a frecuencias relativamente moderadas se ha impuesto como la tendencia actual para incrementar tanto el rendimiento como la eficiencia energética. El aumento del número de núcleos dentro del procesador ha venido acompañado en los últimos años por el aumento de la heterogeneidad en la plataforma, tanto dentro del procesador incorporando distintos tipos de núcleos en el mismo procesador (e.g., la arquitectura big.LITTLE) como añadiendo unidades de cómputo específicas (e.g., extensiones multimedia), como la incorporación de otros elementos de computo específicos, ofreciendo diferentes grados de rendimiento y eficiencia energética. La evolución de los procesadores no sólo ha venido dictada por el aumento del número de núcleos, sino que ha venido acompañada por la incorporación de diferentes técnicas permitiendo la adaptación de las arquitecturas de forma dinámica al entorno así como a las aplicaciones en ejecución. Entre otras, técnicas como el escalado de frecuencia, la limitación de consumo o el particionado de la memoria caché son ampliamente utilizadas en la actualidad como métodos para incrementar el consumo y/o la eficiencia energética.

Asociada a la evolución de los nuevos procesadores, las aplicaciones también han evolucionado para aprovechar todos los recursos ofrecidos por las mismas. De forma similar, las aplicaciones modernas ofrecen un conjunto de parámetros a modificar que permiten un óptimo aprovechamiento de todos los recursos. Aunque estos parámetros son ajustados típicamente de forma estática antes del comienzo de la ejecución, la modificación y ajuste de estos parámetros de forma dinámica favorece la obtención de un mayor rendimiento y eficiencia energética a costa de un proceso de decisión mucho más complejo.

En este escenario, los gestores de recursos cobran un papel de vital importancia, ya que no solo tienen que encargarse de realizar un reparto de recursos adecuado, sino de modificar

de forma dinámica y conjunta los distintos parámetros de la arquitectura y aplicaciones buscando un objetivo común. Además, el alto rendimiento ofrecido por las nuevas arquitecturas ha favorecido escenarios de co-scheduling, donde más de una aplicación es ejecutada simultáneamente compartiendo los mismos recursos. Este escenario complica aún más la labor de los gestores de recursos, donde ya no tienen que considerar los resultados de una única aplicación, sino que tienen que propiciar un escenario de cooperación entre diferentes aplicaciones por el uso de los mismos recursos para satisfacer los requisitos de todas ellas de forma simultánea.

El objetivo de esta tesis es el diseño, implementación y validación de diferentes propuestas para la gestión de recursos de forma dinámica para aplicaciones paralelas, maximizando tanto el rendimiento como la eficiencia energética.

Para ello, en esta tesis se muestra como diferentes técnicas de escalado de voltaje y planificación se pueden integrar dentro de los runtimes encargados de la gestión paralela de las aplicaciones para maximizar la eficiencia energética en plataformas asimétricas, así como se puede incorporar restricciones de potencia a los mismos para gestionar de forma dinámica la potencia disponible en servidores modernos, alcanzando resultados óptimos, similares a los obtenidos mediante mecanismos hardware. Además, el gran número de parámetros disponibles (tanto de aplicación como de la plataforma), así como las posibles relaciones entre ellas proporcionan escenarios muy complejos difícilmente manejables por aproximaciones más tradicionales. Con el propósito de gestionar estos escenarios, en esta tesis se muestra la creación de una solución completa para la gestión de múltiples aplicaciones en tiempo real mediante el uso de Aprendizaje por Refuerzo. Nuestra propuesta muestra cómo, a través del uso de técnicas de Aprendizaje Automático, es posible la gestión de escenarios modernos de gestión de recursos sin necesidad de realizar un modelado previo del mismo, identificando de forma automática las posibles relaciones entre parámetros mientras se persigue una meta multi-objetivo. En concreto, se muestra como este sistema es capaz de gestionar múltiples procesos de codificación de vídeo en tiempo real de forma simultánea, satisfaciendo requisitos de rendimiento, calidad y consumo simultáneamente.

Abstract

Energy-Efficient Resource Management for Task-based Parallel Applications in Multi-Application Environments

The end of Dennard scaling, as well as the arrival of the *post-Moore* era, has meant a big change in the way performance and energy efficiency are achieved by modern processors. From a constant increase of the clock frequency as the main method to increase performance at the beginning of the 2000s, the increase in the number of cores inside processors running at relatively conservative frequencies has stabilised as the current trend to increase both performance and energy efficiency. The increase of the number of cores inside processor has been accompanied in the last years by an increase of the heterogeneity in the systems, both inside the processors comprising different types of cores (e.g., big.LITTLE architectures) or adding specific compute units (like multimedia extensions), as well as in the platform by the addition of other specific compute units (like GPUs), offering different performance and energy-efficiency trade-offs. Together with the increase in the number of cores, the processor evolution has been accompanied by the addition of different technologies that allow processors to adapt dynamically to the changes in the environment and running applications. Among others, techniques like dynamic voltage and frequency scaling, power capping or cache partitioning are widely used nowadays to increase the performance and/or energy-efficiency.

Similar to processors evolution, applications have also evolved to take advantage of all the resources offered by the newest architectures. Identically to processors, newest applications offer a set of tunable parameters allowing optimal use of all the available resources. Although these parameters are typically set prior to the beginning of the execution, a dynamic tuning process can achieve greater performance and energy-efficient executions at the expense of a more complex decision process.

In this scenario, resource managers occupy a vital role, as they do not only have to perform an optimal distribution of the resources, but they also have to tune application and architectural parameters jointly to achieve a common goal. Even more, the high performance achievable by the newest processors has provoked the emergence of new co-scheduling scenarios, where multiple applications are run simultaneously sharing the same resources. In these scenarios, resource managers cannot focus on the results achieved by an unique

application, but they have to favour cooperative scenarios with applications sharing the same resources to achieve all their requirements.

The goal of this dissertation is the design, implementation and validation of different approaches of dynamic resource management for parallel applications, targeting performance and energy efficiency.

With this goal, we show how different frequency and scheduling techniques can be incorporated into the parallel runtimes to maximize energy efficiency on asymmetric platforms, as well as how power-capping can be incorporated to dynamically manage power on modern server processors, achieving optimal results, similar to the ones achievable by top-notch hardware based alternatives. In addition, the huge amount of available parameters (both application and platform), as well as the possible relations between them, makes the decision process a very complex scenario hardly manageable by traditional approaches. To manage these scenarios, we also describe the creation of a holistic solution to manage multiple applications running in real-time using Reinforcement Learning techniques. Our proposal shows how, thanks to the use of Artificial Intelligence techniques, it is possible to manage this kind of scenario without the need of previous profiling of the applications and the use of platform models, automatically identifying the possible relations between applications and parameters, achieving a multi-objective common goal. Specifically, we show how our proposal can manage multiple real-time encoding processes simultaneously, achieving real-time, quality and power consumption requirements simultaneously.

Contents

List of Figures

List of Tables

1

Introduction

1.1. Motivation

The strategies followed in processor and system designs suffered an important change around mid-2000s due to the end of Dennard scaling [52], hitting the integrated circuit (IC) industry a significant *power wall* [26, 80]. Newest technology trend allowed the industry to mitigate some of the power-related problems and still keep valid the Moore's law [30], reducing nodes from 65 nm in 2006 to current 5 nm [125]; in this sense, physical limits may not have been reached yet. However, the end of Moore's Law is close [124] and, unless a breakthrough technology revolution occurs soon, the race that has shaped the progress of computing technologies over the last years will probably come to an end, and will yield the so-called *Post-Moore era* [185].

Computer architectures have evolved drastically in the last decades, seeking an optimal combination of *performance* and *energy efficiency* in response to the growing demands of modern software. The strategies followed in early-2000s cannot be further pursued due to the lack of technological support, and hence, higher core frequencies have stalled around 2-4 GHz to keep heat dissipation under control [15]. The shift towards multi-core architectures alleviated the task of computer architects to improve energy efficiency, but it was revealed as a short-term solution since homogeneous multi-core designs rapidly faced additional walls, mainly due to power and memory bandwidth issues [15]. Actually, multi-core processors have only aggravated the old memory wall faced by High Performance Computing (HPC) systems. As a response to the diminishing performance of multi-core architectures, however, designers have pushed towards new, complex heterogeneous systems that incorporate cores with different energy/performance ratios, together with domain-specific co-processors, in the hope to obtain better overall energy/performance balances. In other words, specialization and heterogeneity have emerged as the most valuable strategies to address the limitations of current architectures [164, 77].

However, both in the HPC arena or in other fields, multi-core architectures are still a central part of high-performance systems. Technological countermeasures, in terms of *dynamic architectural knobs* [134], allow a fine-grained control of compute capabilities in

order to adapt the performance/energy consumption ratio to the specific necessities of the running applications and system's administrators. Among these techniques, Dynamic Voltage-Frequency Scaling (DVFS) [32], Power Capping strategies [62] or Cache Partitioning capabilities [71] are only three examples of hardware-assisted support to add *moldability* to modern computing systems, allowing a proper adaptation to external limitations or requirements in terms of combined performance and power consumption.

In the case of DVFS techniques, moldability is achieved due to the use of different ACPI-states [88] that allow a modification of the power consumption dynamically by means in changes on the frequency and/or voltage. Different power capping strategies take advantage of these power states to try to limit the power consumption based on the characteristics of the applications running on the system [132, 51, 34]. Recently, hardware vendors have incorporated hardware support to offer power-capping capabilities by means of adding specialized power sensors. Running average power limit (RAPL) [138] is one of the most notorious approaches to limit power consumption on modern Intel processors. Cache-partition is other example of moldability offered by modern architectures, that allow to divide non-uniformly the shared cache space and perform a custom assignation of the chunks to the running applications.

From the software perspective, applications do not only imply increasing requirements in terms of performance, but also exhibit unprecedented tuning capabilities in terms of a number of *dynamic software knobs* [66, 139], with direct implications not only in the specifics of the application behavior (e.g. quality, accuracy, performance, ...) but also in the use of computing resources, including core occupation or energy consumption, among others. For example, video encoders offer a plethora of parameters to adapt the quality and resource usage, or even OpenMP applications or Dense Linear Algebra (DLA) libraries allow the user to tune the number of threads to use before the start of the execution. Typically, a proper selection of values for these knobs is performed statically, and hence, the pre-reserved computing resources can be under- or over-utilized during the application lifespan, specially if resource demands are heavily dependent on input data, and hence hardly predictable a-priori [22] (e.g., in a video codification process, the amount of resources needed to encode a frame is ultimately determined by the complexity of the frame being encoded and not by complexity of the overall encoding process). As a response to the ever-increasing number of homogeneous or heterogeneous computing resources, and given the heterogeneity in demands of different applications, the co-existence of several concurrent applications (or virtual machines) seems to be a common trend in many fields, specially in the *Cloud Computing* arena in order to increase resource occupation [31, 33], where decisions are traditionally done by an external scheduler or hypervisor based on the information gathered from the applications. Another scenario in which multiple concurrent applications need to be simultaneously executed arises when applications are tightly coupled, and hence need to be executed concurrently [203, 204].

The existence of *co-scheduled applications*, each one featuring a number of application-level knobs, running on architectures that, at the same time, can be dynamically tuned, yields a number of challenges related with optimal resource management. In the next years, it is expected that these challenges will become a paramount problem in the design of optimal hardware/software interfaces [192]. Actually, the development of holistic solutions that consider the combination of all metrics (independently or simultaneously) and knobs

(at the system or application level) is far from being a trivial task [19, 47].

Beyond performance and energy efficiency, *programmability* can be considered as the third pillar that, in the future, will determine the success or failure of new computing systems [111]. The increase in the complexity of both heterogeneous architectures and parallel applications, together with the introduction of dynamic knobs at both levels, will ultimately come at the expense of programmability, harming code portability not only in terms of performance, but also energy efficiency. Ideally, resource management via a proper selection of dynamic knobs should be transparent for users or developers, so that the impact in their productivity can be considered negligible [134, 146].

In summary, the resource management problem in scenarios in which multiple applications co-exist on top of fully manageable architectures, with tight restrictions in terms of performance, application-specific constraints, energy consumption or power limits, is a problem of paramount interest that will be tackled through the rest of this document.

1.2. Background and definitions

The previous scenario can be formulated in terms of three different actors and their relations: the platform, the running applications, and the resource manager.

In the following, we use the term **platform** to refer to the underlying hardware and all the resources shared between the running applications. Modern processors offer a plethora of different resources to share and tune, from physical cores [38] or cache space [65] shared between applications to the clocking frequency of each core [32] or the power budget [75] shared between resources among others. The term **system-knobs** (called *system-actuators* in control theory) is used to designate all of these parameters offered by the platform and tunable by an external entity (like a resource manager, or even the application by themselves). System-knobs can be classified as *dynamic*- and *static*-knobs depending if they can be tuned in the middle of the execution (e.g., the operational frequency of a core) or they have to be set prior the execution of the application (e.g., enable/disable the hyperthreading technology) respectively.

To properly tune the aforementioned knobs, platforms offer a set of dynamic values exposing the internal status of the hardware (e.g., hardware counters [177] or instantaneous power consumption [138] among others) at each moment (called *sensors* in theory control). These metrics, accessible by the running software and called **system-metrics** in the following, allow resource managers to dynamically tune the different hardware knobs on-the-fly, outperforming other static approaches traditionally based on previous profiling of the applications and systems and built on top of models. However, system-metrics are traditionally exposed at system-wide level, being resource managers responsible to determine the impact of each application on the values measured.

Similar to the platforms, **applications** have evolved to take advantage of all the resources offered by the underlying hardware. One of the most notorious and promising paradigm that allow applications to exploit all the resources at the same time programma-

bility is increased is *task-based* programming model [156]. In this model, applications are divided into small pieces of code called *tasks* and a set of dependencies between tasks that ultimately determine the order in which tasks are executed [25, 56]. The orchestration of the different tasks, as well as the order in which they are executed is done through a **runtime**, and ultimately determines the performance achieved by the application. Commonly, runtimes are presented in the form of external libraries general enough to support different kinds of applications [57]. However, in the case of high-demanding applications, ad-hoc runtimes can be codified inside the applications [109]. Similar to the platforms, and aiming at increase the overall performance of the applications, runtimes offer a set of metrics describing the internal status of the scheduling process (called **application-metrics** in the following), as well as a set of tunable parameters (called **application-knobs**) [87, 66, 67] that influence in the scheduling process, and can be set prior the beginning of the execution (e.g., the number of threads to use), or modified in the middle of the execution (e.g., the policy used to map the different tasks to the different compute units). Similar to the system-knobs, we will refer to these as *static* and *dynamic* application-knobs respectively.

Complementary to the traditional applications, the huge improvements in the performance achievable by modern processors have provoked the emergence of a new family of applications that do not suffer from a lack of performance but exhibit minimum requirements in other specific metrics (e.g., Quality of Service (QoS) or Quality of Experience (QoE)) [95, 87, 66, 67]. Traditionally, the tuning of these metrics has been carried out independently by the so-called **malleable applications**: applications that are able to self-adapt to changes in external parameters, as well as to dynamically modify their internal parameters and hence tune their behaviour at runtime. For example, an encoding process can adapt the quality of each frame dynamically to produced a specific framerate. Similar as before, we will also refer to these metrics as *application-metrics*, and to these parameters as *application-knobs*.

Resource managers close the gap between platform and application by dynamically tuning the application- and system-knobs in a joint manner, maximizing a (multi-objective) goal based on a continuous sensing of the application- and system-metrics. Modern resource managers, typically labeled as **autonomous systems** [174, 195, 146, 204] expose the ability to adapt themselves to environmental changes (i.e., adapt the internal decision process), as well as the capability to improve performance and reduce overloading of the underlying resources. In this sense, resource managers can be classified in terms of the well-known IBM-MAPE autonomic model [90], comprising four different steps (*Monitoring*, *Analyzing*, *Planning* and *Executing*) constituting what is called a *knowledge base*, running in a control loop that features the sensoring and acting capabilities. However, the creation of this knowledge base is far from being trivial, especially if the number of metrics and knobs to consider is high.

Multiple approaches have targeted the problems and scenarios previously described in an individual fashion. However, maximizing performance, quality, or resource usage under a power cap by means of dynamically modifying application- or system-wide knobs in a holistic fashion poses new challenges in the design of system software that orchestrates resource management across applications. The addition of application- and system-metrics and knobs to the resource managers, together with a multi-objective optimization goal

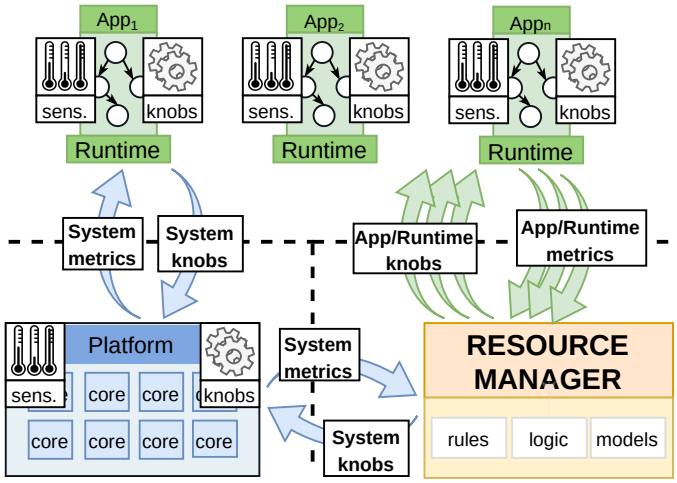


Figure 1.1: Diagram showing all the possible interactions between platform, applications and resource manager. This diagram will be used to explain later the contributions of each chapter.

result in a plethora of scenarios to consider and optimize, hardly manageable by traditional approaches. Fortunately, Artificial Intelligence techniques can be of great appeal to face this complexity burden.

Figure 1.1 shows a diagram with all possible relations between platform, applications and resource manager in terms of metrics and knobs. The rest of this dissertation is devoted to explore the different relations between the aforementioned parts, as well as to propose and validate different strategies to manage this scenario. The same diagram will be used at the beginning of each chapter to introduce the proposed approach.

1.2.1. Target applications and scenarios

Trying to be as most generic as possible, in this thesis we have explored different approaches, applications and scenarios targeting energy-efficient resource management in task-based parallel applications. The applications and scenarios explored are representative examples of any other application of its field. For example, parallel blocked versions of the Cholesky and QR factorization were used as representative examples of DLA parallel operations run through parallel runtimes (like OpenMP), while an HEVC video encoder was chosen as a representative example of other real-time applications widely used in the industry nowadays. In addition, each application was run in different scenarios covering all the possible uses in real-life environments (e.g., running one or multiple applications at the same time with different input sizes, or with different power requirements).

On-demand multi-user video transcoding [95] is one of the most representative scenarios where malleable applications do not require to achieve the maximum performance, but a minimum QoE and QoS. This scenario, which is taken as a driving example in a significant part of this manuscript, aims at encoding multiple video sequences concurrently under tight throughput, quality and power consumption restrictions. This problem, which

is common for many multimedia streaming providers nowadays, aims at reducing the complexity of the encoding process and saving storage space by performing on-the-fly video format re-codifications adapted to user demands [95, 47]. In 2015, real-time entertainment already accounted for more than 74% of downstream network traffic in North America, with streaming services, including Netflix, YouTube, and Amazon Video, accounting for 57% of the global share [160], and it is expected to continue growing in the next years. Indeed, North America is expected to be the first region surpassing the 80% downstream streaming traffic threshold by the end of 2020 [160]. Moreover, video streaming services continue to grow, and users are shifting towards the use of emerging video technologies, such as 4K video resolution, that require different versions of the video (i.e., different resolutions) depending on the device used. The current trend to alleviate the computational needs of serving different versions of the same video to different users involves the storage of all the different version of each video, serving the one that fits better to each user. However, this is a costly and inefficient solution [163, 191] non sustainable in the future as the amount of uploaded and streamed videos is increasing [160].

In this application, both general characteristics of the input data (e.g., video resolution), specific video contents and request arrival times ultimately determine the final throughput attained and can be hardly modeled a priori. In this scenario, a centralized resource manager will need to provide malleability in terms of a dynamic modification of different knobs and/or architectural parameters, consider the specific application metrics that ultimately depend on individual knob combinations and data inputs, and target external limitations in application- or system-wide metrics (e.g., quality or power consumption, respectively). Malleability in video transcoding has been explored in multiple works [207, 113] in terms of a dynamic adaptation of the Quality (by means of changes in the QP value) to guarantee a constant framerate. However, this alternatives focus only on the quality and other output metrics of the application, not considering the implications on the resource usage of the system.

Contrary to this scenario, multiple applications rely in the use of an external runtime to manage the parallelism inside the application instead of dealing with the parallelism by themselves. This is the case of OpenMP [57] or OmpSs [56] task-based applications, that rely in a set of directives (called *pragmas*) to directly instruct the runtime (and compiler) about the parallel flow of the program. Representative examples of this kind of applications (and used in the first part of this thesis), are the blocked Cholesky and QR factorizations [74]. These factorizations, widely used in the engineering and scientific fields, decompose the original matrix in other (smaller) dense linear algebra operations, managed by external state-of-the-art libraries (like Intel MKL [92] or BLIS [183]). In this sense, both factorizations are representative examples of any other DLA application.

The use of an external runtime to orchestrate the execution of the tasks allow the runtime to dynamically tune different runtime- and system-knobs in a transparent way to the user. In this sense, different frequency scaling and scheduling techniques can be incorporated into these runtimes to manage different metrics. In addition, having control of the amount and type of tasks executed at each moment can be useful in scenarios where different applications are competing for the same resources (e.g., physical cores or available power budget). However, in this scenario, a centralized resource manager is needed to

coordinate all the running runtimes.

Lastly, the use of external runtimes can be useful in specialized and heterogeneous architectures where complex scheduling is needed to take advantage of all the available resources [73]. Targeting energy-efficiency, Asymmetric Multiprocessors (AMPs) (like the ARM big.LITTLE platforms) are one of the most promising solution to reduce the energy consumption of modern architectures without affecting performance drastically [147]. In this scenario, an intelligent runtime would map each task to the appropriate compute unit, based on different internal metrics. In addition, different frequency scaling and scheduling techniques can be incorporated into this process to increase the overall energy efficiency of the applications.

1.3. Objectives

Targeting the aforementioned scenarios, the main goal of this dissertation is the *"design, implementation and validation of different autonomous resource management strategies for energy-efficient multi-application scenarios, dealing with different optimization goals combining application- and system-metrics and dynamically tuning application- and system-knobs"*.

In particular, we target two different approaches to this problem in three different scenarios: (I) different runtime-based policies for task-parallel applications targeting energy-efficiency in asymmetric platforms, and software-based power-capping for modern servers, and (II) an application-aware holistic solution targeting real-time applications.

This general goal can be divided into the following specific sub-objectives:

- To develop and validate a set of solutions targeting energy-efficiency on asymmetric platforms. The policies developed will aim at combining the tuning of system knobs (frequency scaling and scheduling policies) with specific runtime-metrics.
- To explore software-based power-capping techniques in modern high-end processors, and increase the overall performance of the system based on specific application metrics.
- To extend the previous ideas considering a power-limited scenario with multiple applications running simultaneously, maximizing the overall performance of the system.
- To design and develop a holistic resource manager able to dynamically tune application- and system- wide knobs, automatically detecting the relations between knobs, and identifying the metrics that are affected by each. A Reinforcement Learning approach will be developed to automatically extract all the dependencies producing high quality policies.
- To design a solution targeting multi-objective optimization goals, combining application- and system-metrics into its formulation. In addition, the solution will be generic enough to modify the optimization goal of the system with minimum effort.

- To study and incorporate inter-application dependency management into our Reinforcement Learning formulation, making a proper tuning of the different hardware-knobs based on the current status of each application.
- To incorporate power-capping techniques into the system to limit the power consumption at the same time other application metrics are considered, not affecting them negatively.
- On multi-application scenarios, to propose an effective methodology to obtain different policies for the same scenario with minimum learning time. In addition, to develop a strategy to apply those different policies to each application based on the current status of the applications and system, all with a common objective.
- To evaluate our approaches in real scenarios with real-world application, and compere the attained results with other state-of-the-art approaches.

1.4. Proposed approaches and contributions

This thesis proposes three different approaches aiming at resource management in different scenarios. Although all of them pursue the same objective, the first two proposals are developed on top of a task-based runtime system, being totally agnostic to the running applications, and therefore, being valid for any task-based parallel applications without any further modification. The proposed policies are based on simplified models of the platforms, not depending on the application. These approaches target energy efficiency on asymmetric platforms and modern high-end processors.

The third approach is an intelligent auto-tuning application-aware resource manager for malleable applications with QoS requirements. This approach is based on Reinforcement Learning, building a precise model for the running applications, combining all the output metrics with the different application- and system-knobs.

Specifically:

1. We introduce in Chapter 3 a set of energy-efficient policies based on frequency scaling and scheduling techniques for task-based parallel codes running on asymmetric architectures. All the proposed policies rely on the number of ready tasks classified as critical and non-critical, and their relation. This set of policies consider the type of core to assign the workload, the number of active cores and the frequency as the dynamic knobs to tune, while performance and energy consumption are considered as the metrics to optimize. This approach targets scenarios with only one application running at a time. In this set of policies:
    - We show how scaling the frequency of the LITTLE cluster does not achieve any improvements on the energy efficiency, but a decrease in the energy consumption.
    - On the contrary, we show how our approach increases the energy efficiency up to 29.3 % when scaling the big cluster.
    - We demonstrate that disabling a cluster in different moments of the execution does not increase the energy efficiency of the system, but a decrease in the power

consumption is achieved. However, our solution is able to increase the energy efficiency up to 17.1 % when switching off the whole cluster instead of disabling it.

- The correction of the different approaches is validated in terms of different highly-used linear algebra kernels (Cholesky and QR factorizations), and different big.LITTLE platforms comprising different processors (32 and 64 bit architectures), and different asymmetry levels (4+4 and 2+4 big and LITTLE cores respectively on each platform).

2. BAR & BACO form a holistic solution to apply a power cap to modern processors via software while the performance of the running applications is maximized. As this approach aims at running in modern multicore servers, it targets a scenario with multiple applications running simultaneously. BAR proposes a mechanism built on top of NANOS++ (the runtime used to orchestrate the task-based OmpSs applications) to achieve the maximum performance when limiting the power consumption. To do so, our solution redistributes the power used among the different idle and active threads, changing the frequency accordingly. We introduce BACO, a framework able to orchestrate multiple applications running BAR, performing an optimal distribution of the power budget between them maximizing the overall performance of the system. Specifically,

   - We motivate the distribution of the power budget between the different threads of the same application, and therefore, the use of different frequencies to maximize the performance while never exceeding the power cap configured.
   - When running on a modern processor, we show how BAR is able to increase the speed up of the application up to 1.9× and to obtain an average reduction of 46 % on energy consumption when compared with the default configuration in NANOS++.
   - We validate the behaviour of BAR respect to the optimal results obtained by RAPL, the state-of-the-art mechanism to apply hardware-based power capping. We show how our approach obtains optimal results in performance, and no violations of the different power caps tested.
   - Similarly, we validate the behaviour of BACO on a realistic scenario when multiple applications are executed concurrently over time. We demonstrate how our approach is able to redistribute the available power budget between applications to achieve the optimal performance.

3. Multi-Agent Learning (MAL) is a complete and holistic solution targeting real-time QoS- and QoE-aware scenarios with multiple applications running concurrently and competing for the same resources. Contrary to other approaches, the design of MAL considers a multi-objective function to optimize, formed by both application- and system-metrics. To achieve that goal, MAL incorporates application- and system-knobs into its formulation, learning how the different knobs can be prioritized and tuned to achieve a common goal. In particular,

- We describe how Q-Learning works internally, and how resource management can be formulated in terms of Q-Learning. Specifically, we propose a new methodology to manage this scenarios with minimum effort, offering a fast and easy method to identify the possible problems in the formulation, and facilitating the development of multiple policies for the same scenario.
- We motivate the use of a cooperative multi-agent approach to handle inter-applications dependencies. In addition, we extend the traditional Q-Learning formulation to support inter-application dependencies.
- We validate our framework with a real-time HEVC video encoder, showing how our approach can handle multiple encoding processes at the same time. In addition, we show how our approach outperforms other approaches, both in terms of heuristic- and machine learning-based.
- We provide insights of how our approach can be used to apply a power cap to the system, and compare our results with other state-of-the-art approach.
- Finally, we propose a methodology to obtain multiple policies, each maximizing a different multi-objective function goal, reducing the overall learning time of the process. In addition, a heuristic is built on top of our MAL system to apply different policies to different applications at the same time, based on the type of each application and the status of the system at each moment.

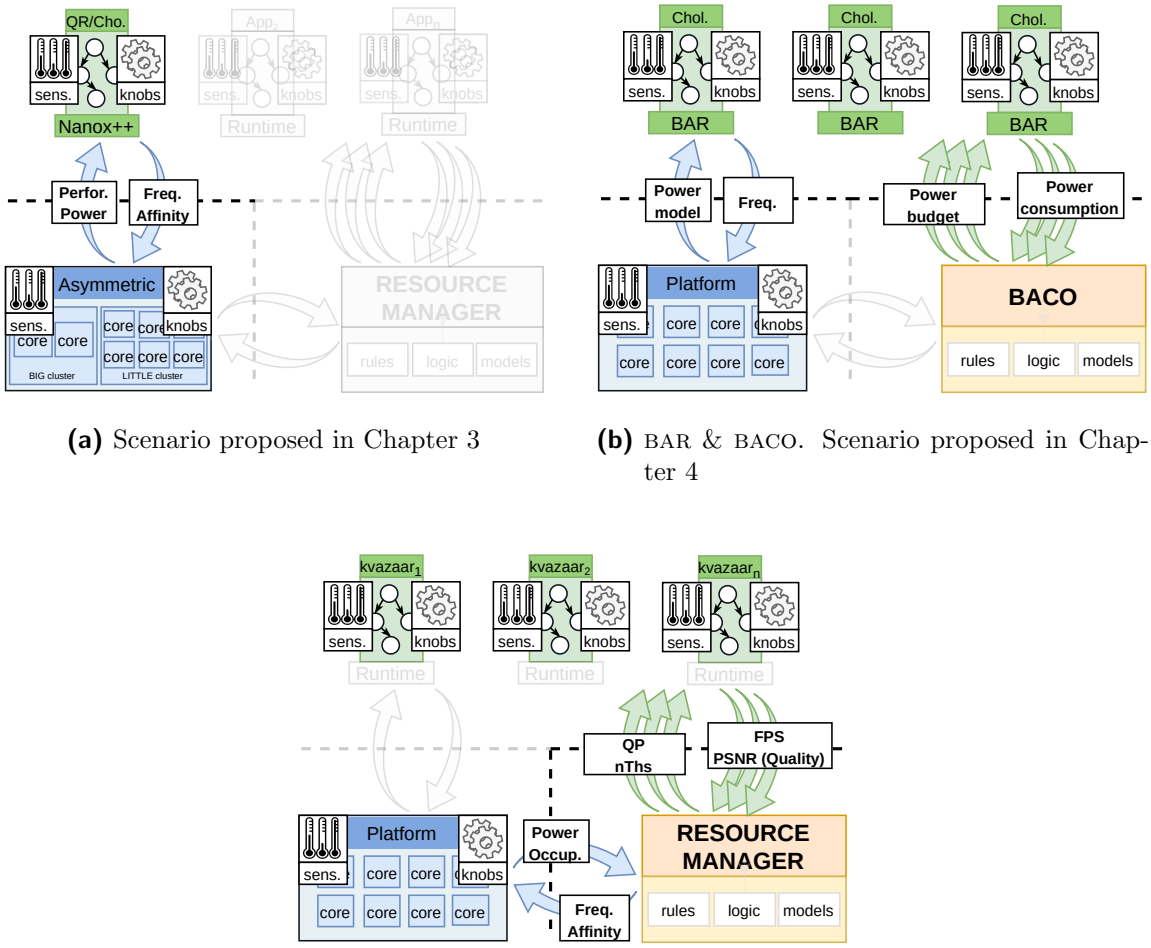
Figure 1.2 shows an schema of all the different approaches proposed in this thesis.

1.5. Document structure

This document is structured in two initial introductory chapters, and two main parts gathering all the approaches proposed.

Chapter 1 (this chapter) motivates the development of this thesis, describes the main and specific objectives we target, and summarizes the different approaches described in this dissertation and their characteristics. Chapter 2 presents the current state-of-the-art, showing different approaches to tackle resource management and auto-tuning frameworks in terms of heuristics and Machine Learning approaches. Additionally, it presents a comparative study of our approach with other state-of-the-art approaches.

Part I describes a set of heuristics targeting energy-efficiency on different platforms. These policies are built on top of a runtime for task-based applications, being totally agnostics of the running applications. Both chapters follow the same structure, introducing the problem the approach wants to solve, a detailed description of the solution proposed, and a complete discussion of the experiments carried out and their conclusions. Chapter 3 deals with asymmetric architectures, proposing an extension of traditional runtimes for task-based parallel applications to increase the energy efficiency on these platforms. In Chapter 4, we present a complete solution to increase the performance on power-limited scenarios targeting high-end processors. First, a solution for only one parallel application is presented, being extended later to consider multiple parallel applications running concurrently, performing a dynamic distribution of the available power budget between applications. Each chapter solves a different problem, presenting the developed work as well



(a) Scenario proposed in Chapter 3

(b) BAR & BACO. Scenario proposed in Chapter 4

(c) MAL approach. Scenario described in Chapters 5 to 8

Figure 1.2: Different scenarios considered in this thesis.

as the experimental results. In this sense, each chapter is self-contained and can be read independently.

Contrary to the previous part, Part II is devoted to develop a complete machine learning-based solution to manage multiple malleable applications running concurrently. The proposed solution targets both application- and system-metrics dealing with application- and system-knobs at the same time. Moreover, a detailed study of our policy applied to a real-time application used nowadays in the industry is presented.

Chapter 5 introduces the problem we want to tackle, and all the theoretical concepts needed to build our solution in the next chapters. In Chapter 6, a complete solution based on Q-Learning is presented targeting only one running application. In addition, multiple experimental results are shown comparing the approach proposed with other state-of-the-art approaches. Chapter 7 extends the system described in the previous chapter to incorporate inter-application dependencies into the formulation and power capping capabilities. Finally, Chapter 8 describes how the previous system can be used to produce and apply different policies at the same time. This chapter describes a heuristic solution built on top of the

previous system to determine the best policy to apply to each application based on the type of application and the status of the system at each moment.

Finally, in Chapter 9 we present the general conclusions of this thesis, a list of the publications derived from it, and a set of open research lines.

2

State of the art

2.1. Traditional Resource Management in parallel computing

2.1.1. Targeting performance optimization

The need for greater performance on the newest architectures has led to an increase in the number of processing units integrated on the same multi- or many-core chips [83]. However, increasing the parallelism entails a non-negligible impact on the programmability of such platforms. In the last decade, high-performance computing applications have relied on *task-based programming models* as an interesting solution [156] that combines a correct orchestration of parallel programs and an abstraction layer that reduces the impact on the complexity of coding the applications and the use of all the different compute units available on the platform. These models aim to perform an optimal execution of the application by splitting the applications in uninterruptible and indivisible fragments of code (called *tasks*) with data dependencies among them (provided by the programmer), orchestrated by a *runtime task scheduler* (or just *runtime* for short) that correctly schedules the execution order of the tasks as dependencies are satisfied, as well as the mapping of the tasks to the correct processor unit.

Several task-based programming models have been proved to be an efficient solution towards the exploitation of parallelism on multi-core, many-core and heterogeneous architectures. Among others, following the path pioneered by Cilk [39, 25, 64, 115], efforts like StarPU [168, 10], Superglue [170, 179, 180], QUARK [150, 198], Kaapi [99, 73], OmpSs [56], Intel TBB [154, 187], and OpenMP (version $\geq$3.0) [133, 57, 11], pursue a common goal: extracting and exploiting task parallelism on modern parallel architectures with minimal intervention of the programmer.

Targeting performance on task-based parallel models, multiple works have tried to achieve optimal scheduling policies for multi-core systems like [199] oriented to general task-based parallel codes, [2] for applications using the StarPU runtime, or [72] using Kappi. Similarly, multiple approaches have been developed targeting heterogeneous platforms for different runtimes: [1] using StartPU, [77] for the Quark framework, or [145] for the OmpSs language. However, as finding the optimal scheduling strategy is an NP problem [18],

most of the previous solutions rely on different performance models or simplified scenarios. Nevertheless, the performance achieved by these models is near the optimal.

The increase of parallelism and heterogeneity on the newest platforms has entailed a non-negligible increase in power consumption. AMPs aim at reducing the power consumption by implementing different processors with different power requirements but a common Instruction Set Architecture (ISA), allowing the migration of the tasks among cores according to their requirements and system status. The efficient use of AMPs on the HPC field is a topic of wide appeal nowadays, especially on task-based codes [37, 38, 41, 36] where dealing with the asymmetry is transparently managed by the runtime system. However, these works focus only on the scheduling process to manage the heterogeneity in the resources, ignoring other optimization opportunities like dynamic frequency scaling among others.

The huge amount of available resources on the platforms makes difficult the selection of the proper knobs for the applications. *Auto-tuning frameworks* have emerged as a valid alternative to reduce this burden. Auto-tuning frameworks can be classified between *static* (where knobs are predefined at installation time or process level) [174, 122, 195, 188], and *dynamic* (where knobs are tuned based on the knowledge extracted from the running applications) [190, 146, 204, 203, 134]. Orthogonally to auto-tuning frameworks, *malleable applications* offer a set of internal knobs to tune, adapting their execution to the changing knob values. Although the term malleability has traditionally referred only to a dynamic change in the number of threads [117, 155], this term can be extended to other application knobs, either managed by the application themselves, or by an external runtime. Combining auto-tune frameworks and malleable applications, multiple works have developed new frameworks [87, 66, 67, 19, 139, 162] to exploit the malleability of the applications based on different application- and system-metrics. A detailed description of these frameworks can be found later. Although all of them solve a specific problem involving knobs and/or metrics from applications and system, none of the aforementioned proposals describe a complete solution extensible to any application able to deal with the four dimensions of the problem at the same time (i.e., application metrics and knobs, and system metrics and knobs).

Additionally, the emergence of cloud computing as a new standard for high performance computations has favored the evolution of traditional resource management and auto-tuning approaches to properly orchestrate the execution of the applications among the different available nodes [101]. For example, [173] proposes a scheduling strategy to assign applications to nodes, considering the best way to pack different applications in the same node to avoid bottlenecks and increase performance based on previous profiling, or [128], that proposes new techniques to make a dynamic workload distribution in cloud environments where performance varies among nodes. However, most of the approaches in the cloud only consider the requirements and dependencies between applications prior their execution, ignoring the variable resource requirements during their execution, as well as the optimization opportunities derived from it. Similarly, the deployment of malleable applications in the cloud is also a topic of interest nowadays [31, 33]. However, opposite to malleability in the node, cloud systems considers malleability of the applications in terms of their ability to adapt to the changes in the amount of resources produced by changes in the cloud system (also called *elasticity of the cloud*), and not by their own internal requirements (e.g., changes in the amount of parallelism required in each phase of the application).

2.1.2. Targeting energy efficiency optimization

Considering energy consumption as a goal to optimize, several papers have been published trying to increase the energy efficiency of the applications (i.e., increase the performance obtained while reducing the power consumption) both in power-constrained scenarios as well as systems without any power cap set. Different policies and approaches have been developed targeting energy-efficient executions in terms of heuristics like [164] that proposes a specific approach for CPU/GPU systems, [65] that proposes a DVFS and L2-cache partition heuristic to control the energy consumption of the system, [9] specifically designed for cloud systems, or [184] that proposes an application-agnostic system to limit the energy consumption in the processor, or in terms of model-based approaches like [51]. Contrary to the approaches described in this thesis, none of the previous works bases its decision on the internal information of the running applications. In addition, some works propose specific scheduling policies to increase the energy efficiency, both for multi-core processors and cloud systems [86, 121, 193, 208], ignoring other techniques like DVFS among others.

Targeting heterogeneous systems [175], and especially asymmetric multi-processors, multiple works have explored how to achieve energy-efficient executions. Works like [147] or [123] have explored these ideas on big.LITTLE platforms, or SPARTA [55], that proposes a throughput-aware multi-application policy for big.LITTLE systems, based on a classification of the running applications. Similar as before, all of these approaches are proposed in terms of an external entity aware of the system status, and not of the internal status of the applications.

Energy efficiency (that is, increase performance while limiting power consumption) has been traditionally searched through the use of DVFS [32], modifying dynamically the operational frequency of the processors or cores (by means of changing among ACPI-states [88]) to reduce the power consumption while maximizing performance as [63], that proposes a Reinforcement Learning approach to manage the frequency or [9] that explores how DVFS can be used to consolidate Virtual Machines in a cloud infrastructure to increase the energy efficiency among others [107]. Specifically targeting at video encoders, several works have explore how to apply DVFS to HEVC encoders [131, 132]. Although these approaches achieves good results when considering energy efficiency, other application and system metrics are not considered on these works.

2.1.3. Targeting power-capping and thermal management

The increase of performance on the processors has entailed a non-negligible increase in power consumption and thermal dissipation, being close to the *dark silicon* problems [58] (the number of transistors is high enough to draw more power than they can sustain). Trying to keep processors safe under a strict power cap, several software systems have been proposed in the literature [62, 114, 152, 153, 196]. Targeting not only power capping but performance on power and thermal constrained scenarios, multiple approaches have been developed in terms of heuristics [75, 79, 143, 176] and model-based solutions [16, 98, 197, 166, 17].

Due to the importance of power capping on the newest processors, Intel has introduced a hardware-based power capping mechanism to its newest chips, called RAPL [138]. RAPL offers an automatic way to measure and limit the power consumption of different parts of processors (called *domains*). Although RAPL can set a strict power consumption limit in the most common situations, different approaches have explored how to use it in other situations. In particular, [76] explores how RAPL can be used to measure small pieces of codes shorter than the refresh frequency of the system, while [103] describes how RAPL works, using it over a set of different benchmarks in a public Amazon EC2 cloud.

Special mention requires the work of *Huazhe Zhang and Henry Hoffmann* [202] that explores a hybrid software-hardware approach to achieve high performance and energy-efficient executions under a power cap. While RAPL is used as the hardware-based mechanism to apply power cap, software power capping is done through the tuning of multiple hardware knobs: frequency, cores in use, hyperthreading use, number of sockets in use on the platform and number of memory controllers. Contrary to hardware-based power capping, software mechanisms can achieve greater performance as they can be aware of the status of the running applications. Other works as [190, 204, 203] have also explored the same idea of modifying hardware knobs dynamically to increase the performance while limiting power consumption. However, contrary to our approach, they do not consider the status of each application individually, ignoring the possibility of redistribute the resources (like power budget) between those applications that would benefit more of them.

2.2. Novel Resource Management strategies

2.2.1. Machine learning for resource management

Similar to other research fields, Machine Learning (ML) techniques, and in particular Reinforcement Learning (RL) algorithms [161], have been used to tackle different problems, mainly application-specific.

Addressing resource management problems, different approaches have been developed targeting different objectives (performance, power consumption, energy efficiency, etc.), both in servers and cloud infrastructures. [84] proposes a *model-based* solution to model a system using Reinforcement Learning. However, the proposal seems difficult to extrapolate to real and more complex scenarios than the one proposed in the paper. Contrary to the previous approach, [63] proposes a *model-free* approach to apply DVFS techniques on NoCs systems.

[54] and [118] propose the use of *rule-based* systems using *Learning Classifier Tables (LCTs)* to increase different application- and system- metrics. An LCT is a table formed by the different rules the system can apply, together with the conditions the system needs to meet to apply a specific rule and a fitness value indicating how good each rule is. The fitness value is modified dynamically based on the metrics measured from the system, using a Reinforcement Learning approach (typically, the tuning of the fitness value is done based on the Bellman's equation [108]). However, the specific rules has to made specifically to the system, being difficult to extrapolate to other systems with minimum effort.

Targeting power and thermal management, [97] propose the use of a mono-agent Q-Learning (QL) implementation to optimize the execution of a non-real-time sequential

HEVC implementation. Although the proposal is complete enough to be be extrapolated to other systems, the use of a non-optimized video encoder makes difficult to extrapolate the results to other real-time applications used in the industry.

Similar to the heuristic approaches, Reinforcement Learning-based approaches have been extensively studied in the cloud. For example, resource management has been explored in [14], in [206] using *Deep Reinforcement Learning*, or in [149] using *region-based reinforcement learning* (RRL) to determine the best cloud configuration to run a specific workload. Mixing Reinforcement Learning with other approaches, [189] proposes the use of Random Neural Networks (RNNs) to dispatch tasks in the cloud and [178] mixes Reinforcement Learning (RL) with Queuing theory.

Focusing on big.LITTLE cloud systems, [130] proposes HISPTER, a solution combining heuristics and Reinforcement Learning to use DVFS techniques targeting energy efficiency and QoS-aware executions. Using Reinforcement Learning techniques instead of a more traditional approach allows Hipster to adapt better to the dynamic workload and hidden dependencies between applications.

2.2.2. QoS- and QoE-aware resource management

Complementary to resource management, several approaches explore QoS- and QoE-aware alternatives to, not achieve the maximum possible performance, but a minimum of requirements in quality [110]. In the cloud environment, QoS-aware approaches usually target to achieve a minimum level of quality in terms of Service Level Agreement (SLA). [165] targets this problem in terms of resource management via a *self-configuration* and *self-optimization* policy, while [61] targets the problem of allocating and deallocating virtual machines in the cloud showing the results in a simulated scenario. Dynamic resource provisioning has also been tackled in terms of multiple priorities for heterogeneous QoS requirements [68, 69]. Targeting load balancing, [104, 3] approaches propose different solutions for video scenarios.

Specifically targeting video encoding scenarios, multiple approaches have been formulated in terms of heuristics [142, 119] and Machine Learning-based [35, 205, 141] solutions. However, some of these approaches do not consider multiple sequences being encoded simultaneously, while others do not consider the competition for the same resources between encoding processes. Alternatively, [116] and [135] propose specific cloud architectures to serve video streaming on demand.

2.3. Frameworks for Resource Management: a comparative study

Our approach gathers characteristics from traditional resource managers, auto-tuning frameworks and malleable applications modifying application- and system-knobs dynamically, and QoS- and QoE-aware solutions, targeting both application-specific metrics and system metrics as power consumption. In addition, our proposal is able to apply different policies to different applications, supporting multi-objective policies for scenarios with multiple parallel applications running concurrently. Although we have not identified any complete solution gathering all the previous characteristics, multiple solutions have been

proposed in the literature targeting one or multiple characteristics:

ReBudget [190] targets a scenario with multiple applications competing for the same resources, each with a different amount of requirements. In this scenario, ReBudget describes a set of metrics and algorithms targeting *efficiency* and *fairness*, together with theoretical proofs of their validation. Although ReBudget does not propose a centralized resource manager distributing the resources, it achieves a proper (and stable) distribution of the resources through an iterative process in which, at each step of the process, applications bid for the resources they need until the *market* is stable. This algorithm guarantees the convergence of the system, and therefore, an optimal distribution of the system. However, as a centralized resource manager is not used, different trade-offs based on internal metrics of the applications cannot be done. In addition, although ReBudget has been proved to be valid in a simulated scenario sharing the shared-cache space and power budget, considering only the speed-up of each application as the output metric, this is far from being executed on a real scenario.

Tangram [146] presents a complete solution targeting heterogeneous platforms, considering multiple hardware knobs and hardware metrics. Tangram proposes a set of hierarchical controllers each covering a different zone of the system and in charge of tuning a set of different hardware knobs. Tangram internally divides each controller in three different engines, each with a different goal and priority (a safety engine in charge to never violate system limits like maximum TDP or temperature, an engine in charge to apply a set of predefined rules in specific situations, and an enhancement engine that applies different rules following a dynamic polynomial model). Authors have demonstrated how the system is able to deal with performance, power, temperature and number of active threads modifying dynamically the frequency and number of active cores. However, this approach does not target multi-application domains nor application knobs, not considering a number of opportunities to improve the overall execution of the applications (e.g., a dynamic adaptation of the quality in an encoding process), as our approach does.

PoDD [204] targets a specific multi-application scenario: *coupled-applications* in a cluster system with a power cap set. Two applications are called *coupled applications* if the performance of one of them is limited by the performance of the other, being needed to speed up the slowest one to improve the performance of the others. Targeting this scenario, authors propose a three steps approach to classify the running applications and distribute the available budget between the different nodes of the cloud system that are running these. In the first step, different hardware counters are collected and send to a special node of the system where applications are classified using a 2-level classifier. Later, a power/performance model is created for each application following a binary search strategy. Finally, power and applications are distributed between nodes using the ideas described in PowerShift [203]. PowerShift proposes a set of different algorithms to redistribute the power between nodes of a cluster running coupled applications, classifying them between those that are close to the power budget assigned and those that not, and in the first case, if increasing the budget would improve the performance of the system or not. However, the ideas presented do not consider any application metric or knob, focusing

only on a cloud environment, ignoring the complexity of sharing resources in the node.

Authors in [134] describe a novel idea to automatically manage different hardware knobs for OpenMP applications. In this paper, authors propose libPRISM, a library that automatically and transparently to the user intercepts the different OpenMP calls, setting automatically the best hardware knobs for each parallel code region identified. libPRISM achieves this goal by an initial profiling of all the hardware knob values for the running application, sorting them in a specific way to later dynamically tune them with the minimum impact in performance. Although authors propose different policies to apply, the design of the system does not allow to change the policy in the middle of the execution, an useful characteristics in scenarios where the workload varies over time. In addition, although the experimental results show how the approach is perfectly valid for tuning multiple hardware knobs on an IBM POWER8 processor, the specific characteristics of this processor makes difficult to extrapolate the results to other commercial processors, being needed more experiments to ensure that.

Dealing with application metrics and knobs, PowerDial [87] presents a framework to automatically tune application knobs trading-off performance and QoS. Internally, PowerDial relies on previous profiling of the different knobs, classifying them in a performance vs QoS space. Dealing with scenarios with power caps, authors have shown how PowerDial is able to adapt to a changing power cap externally applied. However, no power cap logic is integrated into the system.

ARGO [66] addresses the same problem as PowerDial, however, contrary to PowerDial, the ARGO C++ framework allows the system to not consider QoS and performance as the only metrics in the system, but other application- and system- metrics as throughput, memory usage, CPU usage, or hardware counter among others. On its core, ARGO relies on an internal database storing all the application knobs configurations (called Operating Point) and an estimation of the effect of each knob value on the output metrics. In addition, ARGO maintains a set of coefficients that allows it to modify the estimations dynamically based on online measurements. In [67], authors extend the ideas of ARGO into a new framework called mARGOT. However, contrary to our approach, both approaches target a scenario with only one application running, not dealing with the possible inter-application dependencies.

[19] proposes a complete approach targeting both system- and application- knobs. In the paper, authors describe BarbequeRTRM, a framework to dynamically distribute the available resources to the running applications at the same time QoS is maximized. Similar to Tangram, BarbequeRTRM relies on a set of hierarchical distributed controllers to perform the proper knob configurations. The internal logic based on the control theory allows BarbequeRTRM to handle a multi-objective scenarios. However, contrary to our proposal, BarbequeRTRM does not offer an easy way to handle multiple policies at the same time.

Contrary to the previous approaches, SOSA [54] proposes a Machine Learning-based approach for a multi-application scenario. The main goal of SOSA is to increase the

performance of an application running in the foreground, while other applications run on background, as well as control energy consumption of the system. To do so, SOSA internally stores a table with all the possible rules the system can apply, together with the conditions the system has to meet to apply each rule, and a fitness value meaning how good is each rule. The fitness value is modified dynamically following a variation of the Bellman equation based on the measurements read from hardware sensors. Experimentally, authors simulate how the system behaves considering the change of the frequency and the migration of the application between cores as the only hardware knobs. In addition, although the system targets multi-application scenarios, the dependencies between them are not considering.

Table 2.1 summarizes all the characteristics of the previous approaches, and compare them against our proposal.

	Knobs		Metrics		Multi-app	Multi-obj	Power Capping	Logic		Real/ Simu.	Node/ cloud	Parallel. paradigm
	APP	SYS	APP	SYS				HEU	ML			
ReBudget [190]	✗	✓	✗	✓	✓	✗	✓	✓	✗	SIM	NODE	AGNOSTIC
Tangram [146]	✗	✓	✗	✓	✗	✓	✗	✓	✗	REAL	NODE	AGNOSTIC
PoDD [204]	✗	✓	✗	✓	✓	✓	✓	✓	✗	REAL	CLOUD	COUPLED
PowerShift [203]	✗	✓	✗	✓	✓	✗	✓	✓	✗	REAL	CLOUD	COUPLED
libprism [134]	✗	✓	✗	✓	✗	✗	✗	✓	✗	REAL	NODE	OpenMP
PowerDial [87]	✓	✗	✓	✓	✗	✗	✗	✓	✗	REAL	BOTH	AGNOSTIC
argo [66]	✓	✗	✓	✓	✗	✓	✗	✓	✗	REAL	NODE	AGNOSTIC
mARGOT [67]	✓	✗	✓	✓	✗	✓	✗	✓	✗	REAL	NODE	AGNOSTIC
BarbequeRTRM [19]	✓	✓	✓	✓	✓	✓	✗	✓	✗	REAL	NODE	AGNOSTIC
SOSA [54]	✗	✓	✗	✓	✓	✗	✓	✗	✓	SIM	NODE	AGNOSTIC
OUR PROPOSAL	✓	✓	✓	✓	✓	✓	✓	✓	✓	REAL	NODE	AGNOSTIC

Table 2.1: Comparative table of different similar approaches.

Part I

Runtime-based Resource Management

3

Policies for energy-efficient resource management on asymmetric architectures

Asymmetric Multiprocessors (AMPs) are a class of heterogeneous parallel architectures in which cores that implement different micro-architectures share a common ISA and, possibly, a subset of memory resources. Typically, the available architectural heterogeneity is exploited pursuing energy efficiency and performance by a proper redistribution of the applications between the different types of cores. Leveraging low-power architectures to the HPC arena is one of the main trends in the road towards the Exaflop barrier. Indeed, Fugaku has achieved the first position on the latest Top500[1] ranking comprising ARM processors. Same ARM processors occupy the fourth position in the Green500 ranking[1]. Among them, ARM Cortex-A processors, and more specifically, asymmetric System-on-chips (SoCs) based on this micro-architectural family, are nowadays on the spotlight as one of the most promising architectures to achieve such a goal.

To reduce the impact on the programmability of such platforms, task-based parallel programming models have emerged as a promising solution. The extension of these programming models and associated *runtimes* to heterogeneous architectures, managing data coherency and data transfers among isolated memory spaces has been implemented in a number of software efforts, together with techniques that drive to performance gains in multi-core, many-core, accelerator-based and distributed-memory architectures [172, 102]. The necessary efforts to adapt these programming models to AMPs is also a topic of interest of recent works [37, 38], pursuing the goal of boosting performance by correctly mapping critical tasks to the most appropriate element of the asymmetric architecture. These works complement energy-efficiency studies specifically targeting asymmetric architectures [55, 148]. However, the impact and possibilities of *task schedulers* in terms of *improving energy efficiency* of task-parallel implementations has not been previously studied in such a level of detail as needed. The development of policies that equip runtime

[1] `https://top500.org`

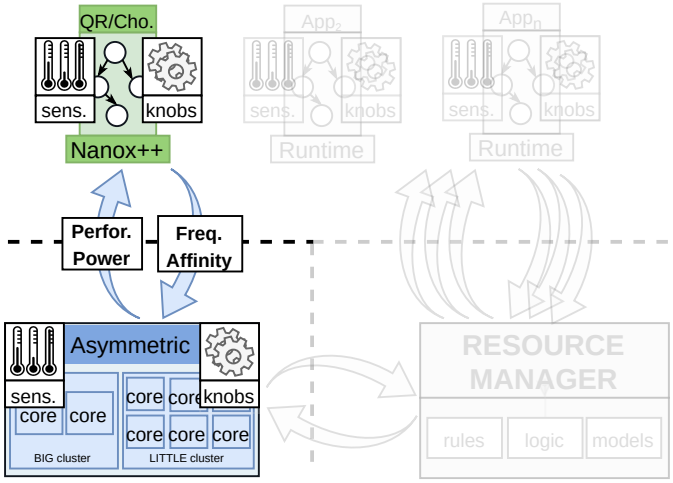


Figure 3.1: Diagram of the proposal described in this chapter.

schedulers with energy-efficient self-adaptive capabilities is still a field of maximum interest.

In this chapter, we pursue the goal of reducing energy consumption with minimal impact on performance and programmability. To that end, we propose extensions to Nanos++, the runtime task scheduler underlying the OmpSs [56] programming model. All these policies leverage internal application-level status to improve energy efficiency, and hence are completely transparent to the application developer, and orthogonal to other techniques that pursue, for example, performance improvements. Figure 3.1 shows a simplified version of our approach. The chapter also serves as an introduction to runtime task-schedulers in general and internal implementation generalities in Nanos++, in particular. Specifically, Section 3.1 is a detailed review of the state-of-the-art in the field of task-based programming models and system software support.

In Sections Section 3.2 and Section 3.3, respectively, we:

- Introduce a set of policies (FS policies) that modify the frequency of operation of modern AMPs via DVFS in an autonomous manner, depending on the internal status of the task scheduler targeting energy consumption reduction, and integrate them within Nanos++.
- Propose new policies implemented on top of the Nanos++ runtime to modify the task scheduling algorithm for AMPs (TS policies), disabling (switching off) a whole cluster based on the number of ready tasks at each moment, and hence improving energy efficiency only when needed.

In both cases, we include detailed evaluations of their benefits in terms of both performance and energy efficiency on two different commercial implementations of the big.LITTLE architecture (a Samsung Exynos 5422 SoC implementing a 32-bit architecture, and a Juno board implementing an ARMv8-a 64-bit architecture, described with more detail in Appendix A), and two different math kernels intensively used in the scientific and engineering fields (Cholesky and QR factorization of dense matrices).

3.1. OmpSs. Internals and asymmetry-aware implementations

OmpSs is one of the most widely accepted programming models nowadays, having influenced greatly on the Open Multi-Processing (OpenMP) task directives design. At a glance, this programming model is based on the inclusion of directives (`pragma`s) similar to those used in OpenMP, that annotate specific sections of code as *tasks*, that is, minimum uninterruptible scheduling units. These annotations include information about operands and their directionality (input, output and input/output). At runtime, this information is handled by a *task scheduler* (named NANOS++) that maps each task to the most appropriate computational resource available as the inferred data dependencies are satisfied.

In the following, we employ the Cholesky factorization of a dense matrix as an illustrative example of the necessary modifications required by OmpSs to extract and exploit the available task parallelism in a specific operation. The Cholesky factorization is at the same time a widely used routine in many problems that arise in science and engineering, and illustrative of other DLA implementations with similar features. Given a symmetric positive definite matrix A of dimension $n \times n$, the Cholesky factorization decomposes it into $A = U^T U$, where the Cholesky factor U is an upper triangular matrix. Listing 3.1 sketches a C implementation of a blocked Cholesky factorization for a blocked matrix A composed of $s \times s$ blocks of dimension (block size) $b \times b$ each. Note that the routine decomposes the global operation into a collection of basic kernels or fundamental operations, namely: `po_cholesky` (Cholesky factorization of the diagonal block); `tr_solve` (solution of a triangular system); `ge_multiply` (general matrix-matrix multiplication); and `sy_update` (symmetric rank-b update).

```
1  void cholesky (double *A[s][s], int b, int s) {
2      for (int k = 0; k < s; k++) {
3        //Cholesky factorization (diagonal block)
4        po_cholesky (A[k][k], b, b);
5
6        for (int j = k + 1; j < s; j++)
7          // Triangular system solve
8          tr_solve (A[k][k], A[k][j], b, b);
9
10       for (int i = k + 1; i < s; i++) {
11         for (int j = i + 1; j < s; j++){
12           // Matrix-matrix multiplication
13           ge_multiply (A[k][i], A[k][j], A[i][j], b, b);
14         }
15         // Rank-b update
16         sy_update (A[k][i], A[i][i], b, b);
17       }
18     }
19 }
```

Listing 3.1: Simplified C version of a blocked Cholesky factorization.

Each of these kernels composes the fundamental parts of the overall computation, or `tasks`. Obviously, provided each task is internally executed in a sequential fashion, the aforementioned code would not extract any further level of parallelism. Figure 3.2a includes the necessary modifications in the definitions of each task in order to exploit the OmpSs programming model and, thus, to extract task parallelism in a transparent manner. Note how each task is annotated with the corresponding `#pragma omp task` directive, including the directionality of each operand involved in the computation. At runtime, the invoca-

tion of each task in Figure 3.2a is intercepted by the runtime task scheduler (NANOS++), that dynamically builds a Direct Acyclic Graph (DAG) as the one shown in Figure 3.2b, including tasks (nodes) and data dependencies among them (edges). Only when all the data dependencies for a given task are satisfied, the runtime dispatches that task to an available processor, effectively exploiting task parallelism.

3.1.1. NANOS++ implementation design

The OmpSs programming model relies on two different pieces of software: MERCURIUM, a source-to-source compiler in charge of parsing and translating the user defined tasks and their dependencies into runtime calls, and NANOS++, the runtime in charge of executing the applications, creating and managing the tasks and data dependencies during the execution. At runtime, the invocation of a fragment of code identified as a task is intercepted by NANOS++, inserting the task into the DAG of the application with its data dependencies. Once a task is executed, its output dependencies are released, checking if new tasks can be executed as all their input dependencies have already been satisfied. Observe how, contrary to other models, OmpSs performs a dynamic creation of the DAG, not knowing any information of the application structure before the start of the execution.

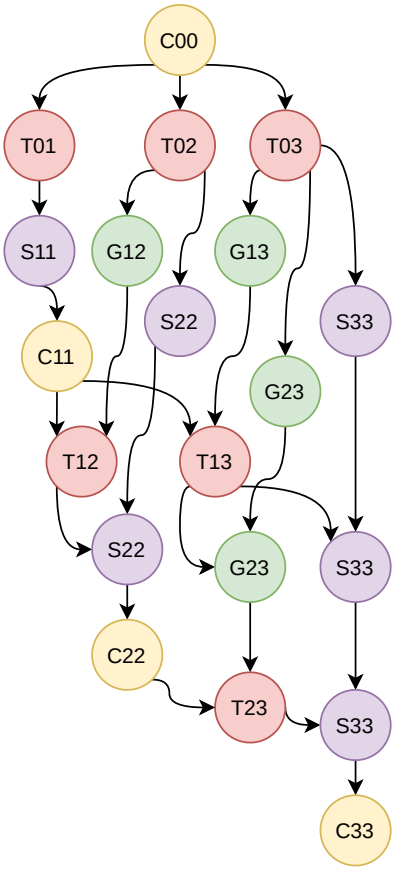
The NANOS++ runtime comprises a pool of threads in charge of executing the different tasks (called *worker threads* or *workers*), and a set of highly customizable modules (called *plugins*) defining the behaviour of different aspects of the runtime, as the *task scheduling* plugin, the *throttling* plugin or the *dependencies management* plugin, among others. Contrary to other approaches, the strategy followed by nanox to achieve the maximum performance relies on the creation of all the workers before the start of the execution of the application, reusing them to execute all the tasks during the whole execution. If no tasks are ready to be executed, workers are called *idle workers*, and are blocked until new tasks are released. In addition, all the serial code is also executed by one of these workers, called the *main thread*. By default, the NANOS++ runtime creates as many workers as compute units exist in the architecture (i.e., number of cores for a multi-core system), setting the affinity of each worker to a different compute unit, avoiding oversubscription situations. Every time a worker finishes to execute a task, its output dependencies are released, checking if new tasks can be executed (because all its input dependencies have already been satisfied). If a new task can be executed, it is called a *ready task*, and it is inserted into a queue of tasks (called *ready queue*). The order in which tasks are executed, as well as the mapping between tasks and workers is determined by the scheduling plugin selected by the user.

3.1.2. Asymmetry-aware modifications in NANOS++

The design of efficient task scheduling algorithms on multi-core and heterogeneous systems has been extensively studied in the past. In platforms with one or multiple accelerators, specialized versions of the runtimes (for example OmpSs, StarPu, MAGMA, Kappi or libflame [201] among others) were developed to schedule the different tasks to both general purpose cores (CPUs) or accelerators (like GPUs or Intel Xeon Phi), mapping each task to a different compute unit based on the task characteristics and the platform status (for example, [10, 12, 73, 151, 180]).

```
1  #pragma omp task inout([b][b]A)
2  void po_cholesky(double *A, int b, int ld){
3    static int         INFO = 0;
4    static const char UP   = 'U';
5
6    // LAPACK Cholesky factorization
7    dpotrf(&UP, &b, A, &ld, &INFO);
8  }
9
10 #pragma omp task in([b][b]A) inout([b][b]B)
11 void tr_solve(double *A, double *B, int b, int ld){
12   static double      DONE = 1.0;
13   static const char LE   = 'L', UP = 'U',
14                      TR = 'T',   NU = 'N';
15
16   // BLAS-3 triangular solve
17   dtrsm(&LE, &UP, &TR, &NU, &b, &b, &DONE, A,
18         &ld, B, &ld);
19 }
20
21 #pragma omp task in([b][b]A,[b][b]B) inout([b][b]C)
22 void ge_multiply(double *A, double *B,
23                  double *C, int b, int ld){
24   static double      DONE = 1.0, DMONE = -1.0;
25   static const char TR   = 'T', NT    = 'N';
26
27   // BLAS-3 matrix multiplication
28   dgemm(&TR, &NT, &b, &b, &b, &DMONE, A, &ld, B,
29         &ld, &DONE,  C, &ld);
30 }
31
32 #pragma omp task in([b][b]A) inout([b][b]C)
33 void sy_update(double *A, double *C, int b, int ld){
34   static double      DONE = 1.0, DMONE = -1.0;
35   static const char UP   = 'U', TR    = 'T';
36
37   // BLAS-3 symmetric rank-b update
38   dsyrk(&UP, &TR, &b, &b, &DMONE, A, &ld, &DONE,
39         C, &ld);
40 }
```

(a) Annotated tasks for the blocked Cholesky factorization.



(b) DAG with tasks and data dependencies extracted from the code on the left when $s = 4$.

Figure 3.2: Task-based parallel implementation of the Cholesky factorization, and its associated DAG on a matrix with 4×4 blocks (s=4). The labels in the DAG specify the type of kernel/task as follows: "C" for the Cholesky factorization; "T" for the triangular system solve; "G" for the matrix-matrix multiplication, and "S" for the symmetric rank-b update. The sub-indices (starting at 0) specify the sub-matrix updated by the corresponding task.

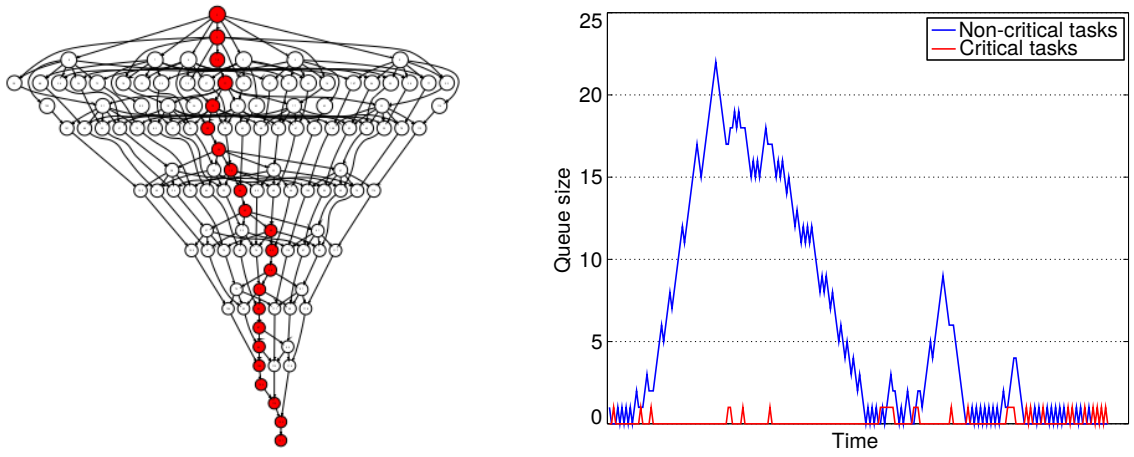
Due to the increasing interest of the asymmetric architectures, some of these works have been recently extended in order to accommodate AMPs as the target platform. Examples of these efforts are CATS [37] or CPATH and HYBRID [38], extending the ideas previously developed for heterogeneous systems, differentiating only two types of compute nodes (a fast node formed by the big cores, and a slow node formed by the LITTLE cores), removing the computing of the costs associated to data transfers between nodes.

These algorithms aim at dynamically identify which tasks belong to the critical path of the DAG, assigning them to the fastest cores, thus reducing the total execution time. A task is considered critical if, in the case the task is delayed, the total execution time of the application is extended too. The CATS algorithm tries to identify dynamically those tasks belonging to the critical path to execute them on the fastest cores, and therefore, to improve the overall performance of the application. Speeding up the critical tasks does not only improve performance by reducing the execution time of these specific tasks, but also speeds up the release of data dependencies, possibly increasing the number of new ready tasks, and therefore, increasing the parallelism of the applications. The main characteristic of CATS versus other similar algorithms is that CATS tries to identify the critical tasks dynamically, without building the dependency graph before the execution and profiling the different tasks.

To properly identify the critical tasks, a complete profiling of application is required to determine the execution time of each task as well as the DAG associated to the application to identify the dependencies between tasks. This approach, even feasible, requires a previous perfect knowledge of the application and a previous pre-processing of the data recorded, making it not practical in scenarios that requires an on-the-fly scheduling strategy. To reduce this burden, CATS identifies the critical tasks as those that belong to the longest path on the dynamic DAG of the application. Although this method does not identify the critical tasks properly, it has been proved to be valid for AMPs architectures [37, 38].

To do so, each task in the DAG is dynamically annotated with an integer (called priority), meaning the length of the longest path from that tasks to a leaf node in the dependency graph. This number is updated every time the DAG changes (i.e., a new task is created and inserted into the graph). Note that, opposite to other models, CATS focuses in runtimes that do not calculate the DAG before the execution but it is built on runtime as the different tasks are finished and new ones are created (as NANOS++ does), being needed to calculate and update this distance dynamically. Every time a new task is created and added to the DAG, all of its predecessor nodes in the graph are updated if the longest path between each task and a leaf node increases. Proceeding this way, the longest distance between each task and a leaf node is always updated. Figure 3.3a shows the critical path determined by CATS for a Cholesky factorization of 8×8 blocks (i.e., $s = 8$). Note that those tasks identified by CATS as critical might not be real critical tasks, which would only be identified by means of a previous profiling of the whole application. Nevertheless, identifying the tasks belonging to the longest path as critical tasks avoids the need of profiling, reducing the total overhead introduced and still improving the performance of the applications as shown in [37].

When a task becomes ready for execution, it is classified as *critical* or *non-critical* based on the priority annotated by the scheduler before (the length of the maximum path between the task and a leaf node at the moment): if it belongs to the longest known path, it is stored as a *critical* task. Each set of ready tasks (critical and non-critical) is dynamically sorted by the annotated priority, prioritizing those with longest distances. Targeting AMPs, CATS



(a) Critical path detected by CATS in a Cholesky factorization of 8×8 blocks ($s = 8$). Source: [37].

(b) Evolution of the queue sizes when executing the previous factorization in ODROID.

Figure 3.3: Critical tasks detection by CATS, and ready queue size evolution in ODROID.

assigns the critical tasks to the big cores, and the non-critical tasks to the LITTLE ones. If no critical tasks are ready, big cores can be used to execute non-critical tasks too. On the contrary, LITTLE cores are not supposed to execute critical-tasks as it can delay the overall execution time; however, as it can be useful in some applications, users can enable this behaviour on demand.

In OmpSs the CATS implementation is called BOTLEV (*Bottom level-aware scheduler*), and it has been used as a starting point for the strategies described in the next section. Internally, ready critical and ready non-critical tasks are stored in two different priority queues sorted by their annotated distances. When a core becomes idle, its worker thread retrieves a ready task depending on the kind of core it is tied to: big cores execute ready tasks stored in the critical queue, and LITTLE cores retrieve tasks from the non-critical queue. BOTLEV enables work stealing for big cores by default, allowing big cores to execute non-critical tasks if the critical-queue is empty. Optionally, work stealing can be activated in a bi-directional fashion. Figure 3.3b shows the evolution of the size of the ready queues when executing a Cholesky factorization configured with 8×8 blocks on the ODROID (see Section A.1) platform.

The previous strategy has demonstrated to be a valid approach to increase the performance on asymmetric architectures. However, no considerations were done in terms of energy consumption nor energy efficiency. In the next sections we propose a set of different policies built on top of BOTLEV to increase the energy efficiency of the running applications targeting AMPs.

3.2. Energy-aware policies based on frequency scaling (FS)

We introduce two different general approaches that pursue an improvement in the energy efficiency of task-parallel codes on asymmetric architectures. The first group of policies, de-

scribed in this section and named as FS (standing for *Frequency Scaling* policies), is based on the dynamic application of DVFS techniques at runtime. The goal is to integrate these techniques on an asymmetry-aware scheduler, and to reduce energy consumption by modifying the frequency of one of the clusters based on the internal state of the scheduler, without further modifications on the scheduling algorithm. Pursuing the same goal, the second group of policies, described in Section 3.3 and named TS (standing for *Task Scheduling* policies), implements different asymmetry-aware scheduling algorithmic variations on existing task schedulers.

In the following, all policies are based on the BOTLEV scheduling algorithm previously described, that is, tasks identified as critical are executed only in big cores, meanwhile non-critical tasks are executed in LITTLE cores. If big cores are idle, non-critical tasks can be executed in big cores too.

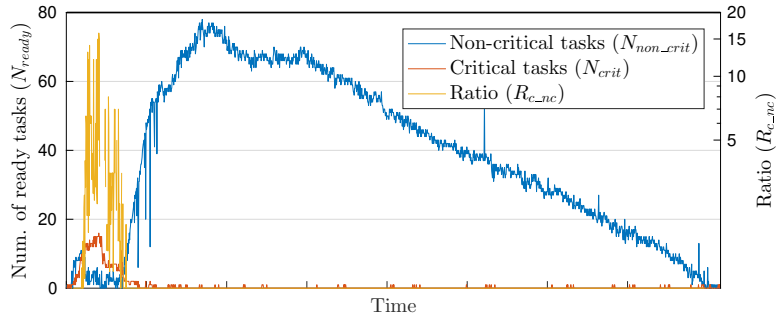
3.2.1. FS policies description

DVFS techniques represent a set of different techniques characterized by a dynamic adjustment of the operational frequency to increase the energy efficiency or decrease the instantaneous power consumption. Nowadays, this kind of techniques are of wide appeal on mobile devices, where the battery duration is a factor of vital importance to take into account. However, this technique does not limit only to this type of devices, but any other processor with dynamic frequency support. Among others, Intel supports this family of techniques through the Intel SpeedStep technology [93], or AMD through the Cool'n'Quiet technology [7] for server processors, and AMD PowerNow! [6] for mobile ones. In addition, similar techniques can be also applied to specific domain devices, as the ones implemented by AMD for its GPUs and APUs (called AMD PowerTune).

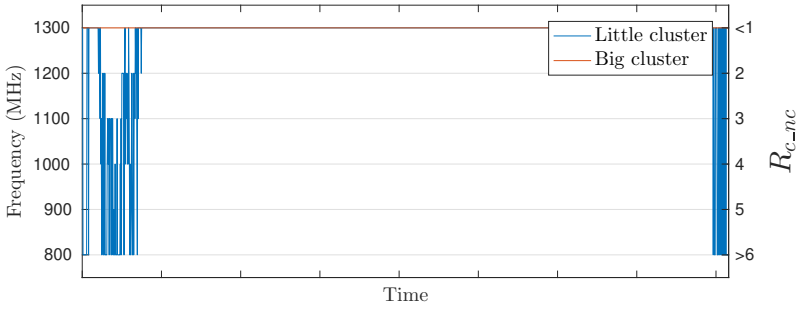
ARM big.LITTLE processors family also supports a dynamic adjustment of the running frequency. However, contrary to the other approaches, all cores in the same cluster share the same frequency domain, hence per-core frequency selection is not possible. In addition, as big.LITTLE processors are mainly designed for mobile and embedded systems, the range of frequencies each core can run is highly limited by design factors.

Applying DVFS techniques to a task-parallel problem requires three main runtime decisions to be made, namely: (I) which frequencies (among those available) to use, (II) at which moments of the parallel execution these changes need to be made, and (III) which elements of the architecture (among those that support DVFS) are affected by the voltage/frequency scaling. The set of frequencies that a processor can run at is usually defined by the architecture, so the first decision is reduced to choosing between using all the available frequencies or just a subset of them. The remaining decisions are directly related to the specific problem to tackle, and the knowledge that the task scheduler has of it.

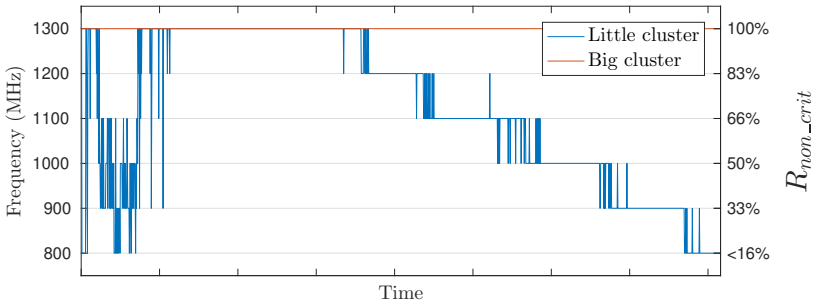
Figure 3.4a shows, for a Cholesky factorization of a 1024×1024 matrix divided in blocks of dimension 64×64 (i.e., $m = 1024$, $b = 64$, $s = 16$), the evolution in time of the amount of critical and non-critical tasks ready for execution (N_{crit} and N_{non_crit}, respectively, being $N_{ready} = N_{crit} + N_{non\_crit}$), together with the ratio between them ($R_{c\_nc} = N_{crit}/N_{non\_crit}$). In the following, we also consider N^{nc}_{max} and N_{max} as the maximum amount of ready non-critical tasks and ready tasks (critical and non-critical) observed from the beginning of the execution at each moment. Both values, N^{nc}_{max} and N_{max}, are constantly monitored and updated at runtime by the scheduler. Finally, $R_{non\_crit} = N_{non\_crit}/N^{nc}_{max}$ denotes the ratio



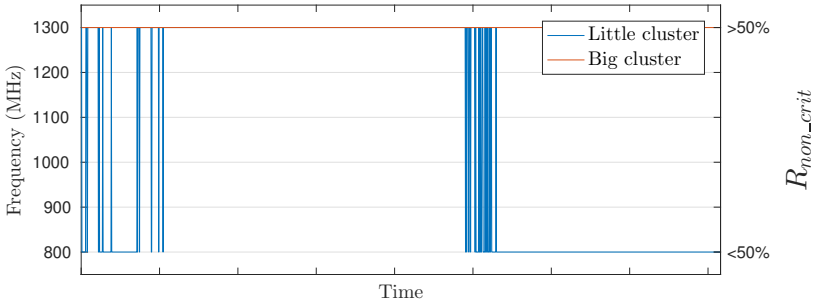
(a) Evolution of the number of critical and non-critical ready tasks.



(b) Policy FS1.



(c) Policy FS2. Notice that policy FS3 will exhibit the same behavior, but applied to the other cluster.



(d) Policy FS2'.

Figure 3.4: Behavior of each FS policy when applied to a Cholesky factorization configured as $m = 1024$, $b = 64$ on ODROID.

Abbr.	Meaning
N_{crit}	Number of ready critical tasks
N_{non_crit}	Number of ready non-critical tasks
N_{ready}	Number of ready tasks ($N_{crit} + N_{non_crit}$)
R_{c_nc}	Ratio between ready tasks (N_{crit}/N_{non_crit})
N^{nc}_{max}	Maximum amount of ready non-critical tasks from the beginning of the execution
N_{max}	Maximum amount of ready tasks from the beginning of the execution
R_{non_crit}	Ratio between non-critical ready tasks and the maxim value ($N_{non\_crit}/N^{nc}_{max}$)

Table 3.1: List of abbreviations used in this chapter.

of non-critical ready tasks compared with the maximum amount observed for this value. Table 3.1 summarizes all the abbreviations used in this chapter.

Policy FS1**. Tasks limited by the critical path**

Runtime task schedulers annotate tasks while the DAG is built and, typically, no further external information is used; thus, it is possible that multiple paths of the DAG are detected as critical at the same time. On an asymmetry-aware scheduler like BOTLEV, this fact entails moments of the execution with most of the ready tasks classified as critical, so that there are not enough tasks ready to execute on the LITTLE cores. Asymmetry-aware task schedulers alleviate these situations by allowing critical tasks to be executed by both types of cores until new non-critical tasks are ready to run. However, using LITTLE cores to execute critical tasks can slow down the execution as, despite the fact that tasks can start their execution earlier due to the greater number of available cores, running a task on a slow core can increase its execution time meaningfully.

Our approach to respond to this situation is different, as is our goal (reducing energy consumption or increasing energy efficiency): the FS1 policy leverages these moments – where the number of ready critical tasks is greater than the number of ready non-critical tasks– to reduce power consumption by decreasing the frequency of the LITTLE cluster. The side effect is that the execution time of non-critical tasks increases, but as the global execution time is limited by the critical tasks executed on the big cluster, delaying the execution of non-critical tasks on these moments should not dramatically impact the global performance of the application.

In FS1, the decision on which frequency the LITTLE cluster should run at is made by the scheduler each time the number of ready tasks changes (i.e., when a task becomes ready or a ready task is executed by an idle core), and it is based on the *relation* between the sizes of both queues (R_{c_nc}), that determines the specific frequency step that will be applied to the LITTLE cluster. For example, if $R_{c\_nc} = 2$, the LITTLE cluster will run at its second maximum available frequency; if $R_{c\_nc} = 5$, the cluster will run at its fifth maximum frequency available.

Figure 3.4b reports the instantaneous frequency applied by the task scheduler when applying FS1 on the same execution as that shown in Figure 3.4a. Observe how, when the number of ready critical tasks is higher than the number of ready non-critical tasks (e.g. at the beginning and end stages of the execution in this example), the frequency of the LITTLE cluster is scaled down, and how the frequency chosen for the cluster is directly

related with R_{c_nc}. Also, note how, when N_{non_crit} increases, the policy forces the LITTLE cores to run at a higher (even at the maximum) frequency.

Policies FS2 and FS2'. LITTLE cluster frequency scaled based on the workload

Instead of modifying the frequency based on the *ratio* between the number of both types of ready tasks, policies FS2 and FS2' modify the frequency based on the *absolute amount* of non-critical tasks at each moment, i.e., if there is a high number of non-critical tasks, the LITTLE cluster will run at a high frequency, and if the number is low, the frequency will be lower. In this policy, only non-critical tasks are taken into account as critical tasks will run on big cores, and they are not affected by these policies.

In order to determine when the number of non-critical tasks is considered high or low, N_{non_crit} is compared with N^{nc}_{max}. If higher, FS2 and FS2' will consider that the number of non-critical tasks is high, and the LITTLE cluster will run at its maximum frequency; if not, frequency is scaled down depending on the value of R_{non_crit}.

The difference between FS2 and FS2' is the set of frequencies to select: while FS2 chooses one between all the available frequency steps according to R_{non_crit} (see Figure 3.4c), FS2' only uses the highest and lowest available frequencies (see Figure 3.4d). In this case, if the current number of non-critical tasks is lower than the 50% of the maximum amount recorded (that is, if $R_{non\_crit} < 0.5$), the frequency will be the lowest available, in other case, it will be the highest.

Observing the evolution of N_{crit} and N_{non_crit} in Figure 3.4a, two different phases can be distinguished: a first phase where the number of ready non-critical tasks increases, and a second phase where it decreases. This behaviour matches with a Cholesky factorization DAG, which enlarges very fast at the beginning, and decreases slowly later. During the first phase, the maximum amount of ready non-critical tasks is growing, so the LITTLE cluster is running at its maximum frequency; during the second phase, the scheduler scales down frequency based on the amount of non-critical tasks and available frequencies.

Policy FS3. Big cluster frequency scaled based on the workload

The behavior of policy FS3 is similar to that of FS2, but, instead of modifying the frequency of the LITTLE cluster, FS3 scales the frequency of the big cluster, depending on the absolute amount of non-critical tasks at each moment.

3.2.2. Experimental results

Experimental setup

We target two different big.LITTLE platform for our experiment: an ODROID board implementing a 32-bit architecture configured with a 4+4 big and LITTLE cores respectively, and a JUNO board implementing a 64-bit architecture with a 2+4 core configuration (big and LITTLE respectively). As explained before, in both platforms core frequency is shared among all the cores of the same cluster. Table 3.2 shows the different frequency values configurable on each cluster. A more detailed description of both platforms can be found in Appendix A.

Platform	Cluster	Frequencies supported
ODROID	LITTLE	800 MHz, 900 MHz, 1.0 GHz, 1.1 GHz, 1.2 GHz, 1.3 GHz
	big	
JUNO	LITTLE	450 MHz, 575 MHz, 700 MHz, 775 MHz, 850 MHz
	big	450 MHz, 625 MHz, 800 MHz, 950 MHz, 1.1 GHz

Table 3.2: Available frequencies for each cluster on each tested platform.

In the following, we report the experimental results obtained for the Cholesky factorization described before, and the QR factorization described in Appendix A. Similar to the Cholesky factorization, the QR factorization is a highly used routine in the DLA domain, being implemented in terms of other linear algebra routines (`larft` to form a triangular factor of a block reflector, `larfb` to apply a block reflector to a general matrix, and `geqrf` to apply a QR factorization). All experiments were carried out using single precision and gathering power results from physical meters exposed in each board. Each experiment was repeated ten times, showing the average measurements in the following. In all cases, we show results for performance (in terms of GFLOPS), average power consumption (in Watts) and energy efficiency (in GFLOPS/Watt). Only results of the parallel phase are shown, omitting those measurements related with the memory reservation and matrices initialization.

For the sake of fairness, we compare our approach against an execution using the BOTLEV scheduling algorithm without any additional policy (named PBOTLEV in the following).

Detailed results

Figure 3.5 shows the results obtained when applying policies from FS1 to FS3 to different Cholesky and QR factorizations on an ODROID platform. The experiments cover a range of different matrix sizes and block dimensions. A number of general, preliminar remarks can be extracted from the results. Depending on the matrix size, the conclusions differ, namely:

- When factorizing small matrices ($m \leq 2048$) with the Cholesky decomposition, there is a considerable difference between the performance obtained when the factorization is made without any policy (PBOTLEV) and when using any of our policies. This big difference in the performance has a huge impact on the energy efficiency.
- For large matrices ($m \geq 4096$), applying our policies also implies a penalty in performance, as expected. However, energy efficiency measurements are very similar to PBOTLEV. In this case, FS3 clearly outperforms PBOTLEV in terms of energy efficiency. In addition, as a positive side effect and for this range of problem sizes, the application of any FS policy clearly reduces the average power consumption (in Watts) of the execution.
- Applying policies FS1, FS2 and FS2' have a similar impact on the energy efficiency than not applying them, however, using policy FS3 achieves improvements in energy efficiency and power consumption over using an asymmetry-aware scheduler (policy

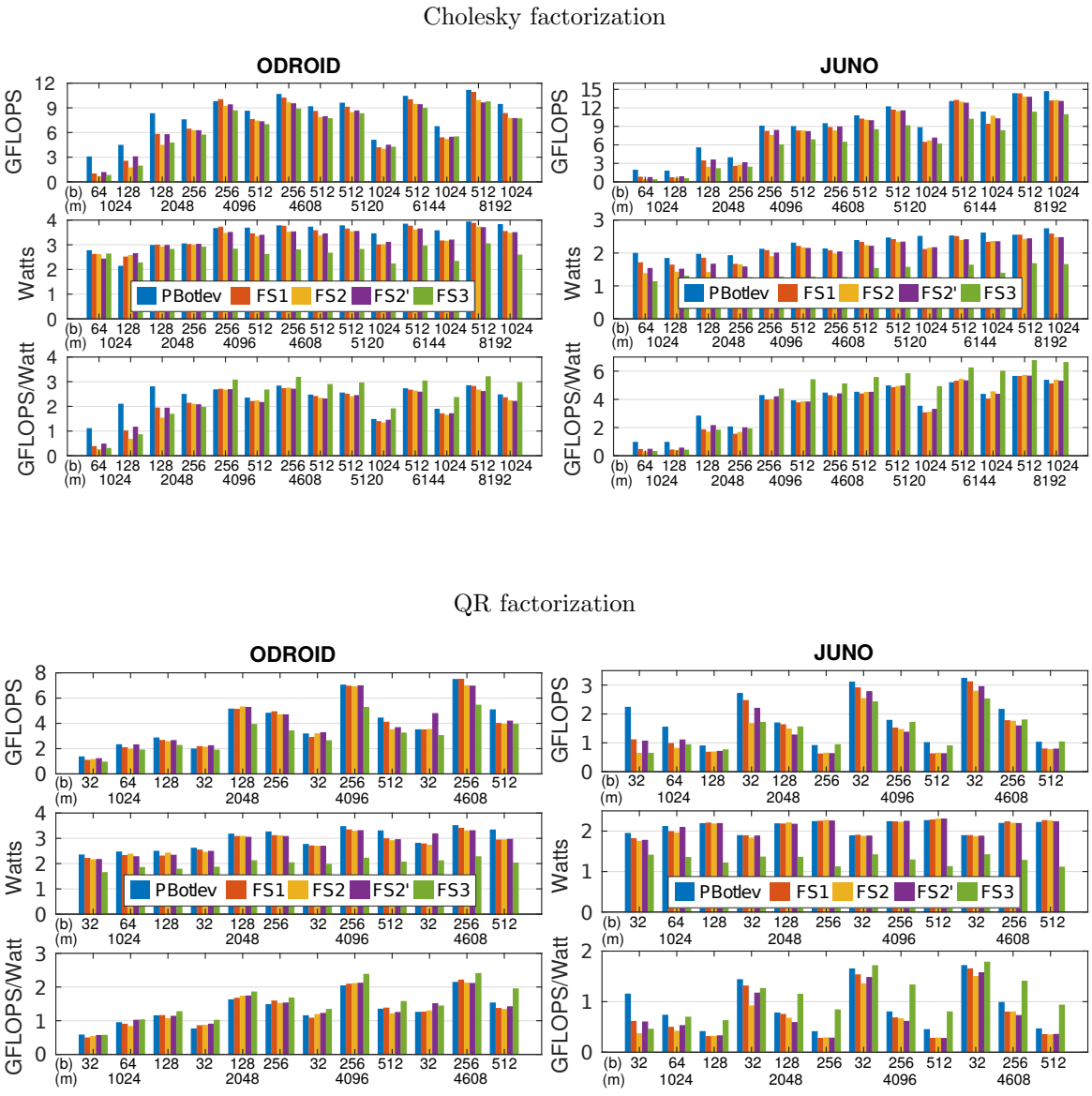


Figure 3.5: Experimental measurements for policies from FS1 to FS3 on ODROID and JUNO platforms for Cholesky and QR factorizations with different configurations. PBOTLEV stands for a normal execution using the BOTLEV scheduler without any additional policy. Tags in the horizontal axis represent the sizes of the matrix (m) and blocks (b) of each experiment.

	(m)	1024		4096		4608		5120		6144		8192	
	(b)	64	128	256	512	256	512	512	1024	512	1024	512	1024
JUNO	FS1	-0.54	-0.90	0.04	0.10	0.05	0.05	0.05	0.40	0.02	0.28	0.0	0.16
	FS2	-0.38	-0.69	0.23	0.15	0.16	0.15	0.14	0.36	0.14	0.25	0.13	0.27
	FS2'	-0.54	-0.63	0.12	0.16	0.09	0.18	0.13	0.34	0.12	0.26	0.11	0.27
	FS3	-0.13	-0.56	0.85	1.04	0.87	0.85	0.89	1.26	0.89	1.22	0.87	1.09
ODROID	FS1	-0.93	-0.30	-0.04	0.17	0.00	0.13	0.11	0.42	0.07	0.39	0.05	0.27
	FS2	-1.22	-0.46	0.15	0.27	0.22	0.30	0.21	0.41	0.22	0.40	0.19	0.34
	FS2'	-0.78	-0.09	0.13	0.22	0.20	0.23	0.20	0.33	0.18	0.35	0.22	0.31
	FS3	-1.09	-0.21	0.73	0.93	0.87	0.96	0.89	1.18	0.85	1.21	0.86	1.23

Table 3.3: Improvement of average power consumption (in Watts) for FS policies respect to PBotlev for the Cholesky factorization. Similar results are obtained for the QR factorization in both platforms.

PBotlev). The reason of this behaviour is that, while LITTLE processors are highly energy optimized, not having enough room to increase the energy-efficiency, the greater power consumption of big cores allow our solution to reduce the energy consumption enough to increase the energy efficiency.

Diving into details of average power and energy efficiency results for each policy, a number of specific insights can be extracted:

- First, the gap in performance, average power and energy efficiency between policies FS2 and FS2' is not remarkable and, similar to policy FS1, experimental results do not show any improvement in terms of energy efficiency when using these policies for these applications and platforms. However, although using these policies does not achieve any improvement in performance or energy efficiency over not using them, a decrease in the power consumption is observed, making these policies of great appeal when targeting environments where the power consumption is limited by design. Table 3.3 reports the decrease of power consumption (in Watts) achieved for each policy for the Cholesky factorization, obtaining similar results for the QR factorization. In the first set of matrices (the ones with lowest size), the power consumption increases, but, in the second group, the power consumption decreases in all matrix configurations and for all the policies, achieving a decrease up to 0.41 Watts (12.85 %) for policies FS2 and FS2', and a decrease up to 1.21 Watts (34.88 %) for policy FS3.
- Second, the penalty introduced by the application of FS1, FS2 and FS2' in terms of performance does not make up for the improvements in average power introduced by the frequency scaling in those policies. Thus, for this problem, they actually increase the energy consumption of the solution.
- Finally, from Figure 3.5 we can observe that the policy which obtains the best results is FS3, outperforming Botlev in terms of energy efficiency. Table 3.4 reports a detailed study of the energy efficiency improvement (in GFLOPS/Watt) of each policy and matrix configuration compared with a normal execution using Botlev for the Cholesky factorization. For matrices larger than 2048 elements, FS3 obtains a rise

		Cholesky factorization											
	(m)	1024		4096		4608		5120		6144		8192	
	(b)	64	128	256	512	256	512	512	1024	512	1024	512	1024
ODROID	FS1	-4.84	-3.84	0.04	-0.19	-0.12	-0.07	-0.05	-0.10	-0.06	-0.20	-0.03	-0.12
	FS2	-5.12	-4.87	-0.04	-0.16	-0.11	-0.17	-0.17	-0.16	-0.11	-0.26	-0.18	-0.25
	FS2'	-4.64	-2.49	-0.01	-0.23	-0.16	-0.19	-0.12	-0.04	-0.15	-0.19	-0.25	-0.27
	FS3	-5.01	-4.22	0.41	0.31	0.34	0.43	0.41	0.43	0.31	0.48	0.36	0.50
JUNO	FS1	-1.41	-1.96	-0.32	-0.15	-0.18	-0.12	-0.12	-0.47	0.10	-0.33	-0.01	-0.25
	FS2	-1.64	-1.20	-0.29	-0.06	-0.26	-0.01	-0.04	-0.42	0.25	0.18	0.07	0.01
	FS2'	-1.44	-1.77	-0.09	-0.08	-0.06	0.00	-0.01	-0.22	0.13	0.01	0.02	-0.06
	FS3	-1.60	-1.97	0.46	1.50	0.65	1.05	0.86	1.39	1.05	1.64	1.11	1.26

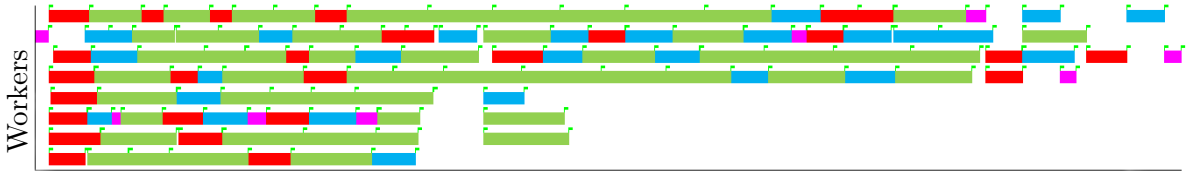
Table 3.4: Improvement of energy efficiency (in GFLOPS/Watt) for FS policies respect to PBOTLEV for the Cholesky factorization.

		QR factorization											
	(m)	1024			2048			4096			4608		
	(b)	32	64	128	32	128	256	32	256	512	32	256	512
ODROID	FS1	-0.09	-0.05	0.01	0.10	0.04	0.11	-0.08	0.05	0.03	0.01	0.07	-0.16
	FS2	-0.05	-0.11	-0.09	0.11	0.10	0.03	0.03	0.07	-0.14	0.04	-0.02	-0.19
	FS2'	-0.02	0.07	-0.01	0.14	0.11	0.05	0.07	0.08	-0.10	0.26	-0.03	-0.11
	FS3	-0.01	0.08	0.12	0.26	0.23	0.20	0.19	0.35	0.23	0.20	0.26	0.42
JUNO	FS1	-0.54	-0.24	-0.10	-0.13	-0.03	-0.13	-0.12	-0.12	-0.17	-0.07	-0.19	-0.11
	FS2	-0.79	-0.32	-0.09	-0.52	-0.10	-0.12	-0.30	-0.14	-0.17	-0.22	-0.19	-0.12
	FS2'	-0.55	-0.21	-0.09	-0.27	-0.19	-0.13	-0.17	-0.19	-0.18	-0.14	-0.26	-0.11
	FS3	-0.70	-0.04	0.22	-0.18	0.37	0.43	0.06	0.53	0.35	0.07	0.42	0.47

Table 3.5: Improvement of energy efficiency (in GFLOPS/Watt) for FS policies respect to PBOTLEV for the QR factorization.

on energy efficiency, achieving improvements from 11.7 % up to 29.3 %. Similar as before, experiments using the QR factorization (shown in Table 3.5) produce the same behaviour as the obtained by the Cholesky factorization.

In general, for policies FS1, FS2 and FS2', the reduction in power consumption is negligible respect to the decrease in performance, which implies a subsequent decrease in energy efficiency. For policy FS3, performance also decreases, but the big improvements in power consumption make the energy efficiency rise. As a side effect, one of the consequences of applying policy FS3 is that power consumption in the LITTLE cluster increases. The reason of this behavior is that, when big cluster scales down its frequency, ready tasks increase their execution time. As a result, big cores cannot execute non-critical tasks that they would have stolen if the critical tasks had been executed earlier. This behavior makes the LITTLE cores to run tasks that, in a normal execution, would be run by the big cluster, and therefore, increase the energy consumption. However, the increase in power consumption of the LITTLE cluster is traded off by the decrease on big cluster, improving the overall energy efficiency.



(a) Execution trace per core.

(b) Num. of ready tasks.

Figure 3.6: Policy TS2: task scheduling based on the number of ready tasks, for a Cholesky factorization of a square 4096×4096 elements matrix, grouped in square blocks of 512×512 elements each executed on an ODROID platform, where 0-3 cores belong to LITTLE cluster, and 4-7 cores to big cluster. Color key: red=TRSM, pink=POTRF, blue=SYRK, green=GEMM, white=IDLE.

3.3. Energy-aware policies based on task scheduling (TS)

The TS (task scheduling) policies described next are based on the same ideas as FS (frequency scaling) policies but, instead of applying DVFS techniques, they decide at runtime the phases in which both clusters are considered to execute tasks, or just one of them is used as a scheduling target. On one hand, using only one of the clusters in specific moments means that power consumption is likely to decrease, but on the other hand, performance will also be affected. Our goal is to find a trade-off between both parts, and thus to improve energy efficiency.

3.3.1. TS policies description

Policies TS1 and TS2. Making cluster unusable depending on the workload

Similar to policies FS2 and FS3, these policies track the value of N_{ready} at each moment, and determine when the amount of tasks is increasing or decreasing (comparing this value with N_{max}). If the number of ready tasks is low enough, the policy will not assign any new task to one of the clusters, making it to be in an idle state from the scheduler's perspective, and saving power consumption. If the number of tasks increases later, the cluster becomes available again and it will execute new tasks as they become available. The amount of tasks (or threshold) that determines when to disable or enable the cluster (denoted as N_{thres} in the following) is configurable and not defined by the policy; several experiments with different values for N_{thres} can be found in the next section.

The difference between policies TS1 and TS2 is that, while TS1 acts on the LITTLE cluster, TS2 acts disabling and enabling the big cluster. As TS2 disables the big cluster in some moments of the execution, critical tasks are executed on LITTLE cores until the big cluster is enabled again.

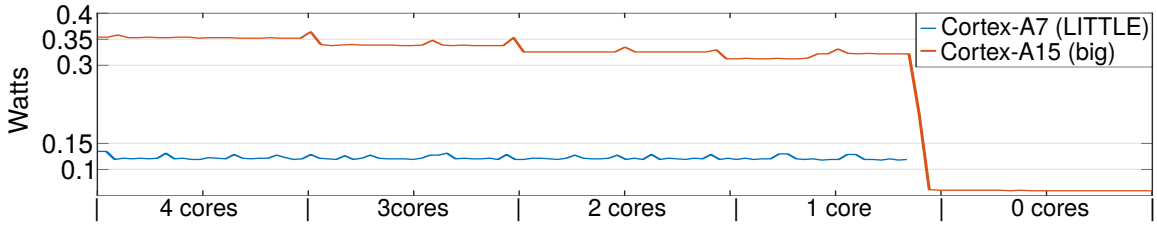


Figure 3.7: Power consumption of each cluster on idle state with different number of active cores. Linux kernel does not allow switching off the whole LITTLE cluster, thus measures could not be made for this scenario.

Figure 3.6 shows an execution of policy TS2 applied to a Cholesky factorization, where the cluster is disabled when the current number of ready tasks is under the 30% of N_{max} (that is, $N_{thres} = 30\%$). Each line in the trace corresponds to a specific core executing tasks (colored areas) or in idle state (white areas). The trace has been obtained on an ODROID platform, where cores (numbered from the top to the bottom) 0-3 belong to the LITTLE cluster, and cores 4-7 to the big cluster. The plot at the bottom shows the number of ready tasks at each moment. On the top of it, a plot with the state of each core at each moment is shown. Each color means one type of tasks (the meaning of each color can be read on the caption of the figure), and white color means that the core is idle. Observe how, at the beginning, the task scheduler assigns tasks to all the available cores, until the number of ready tasks is under 30% of maximum recorded; from that moment on, no tasks are assigned to big cluster. As there are less cores to execute ready tasks, in some moments of the execution the number of ready tasks becomes greater than N_{thres}, starting big cores to execute ready tasks until the number of ready tasks decreases again and the cluster becomes unavailable for scheduling purposes.

Policy TS3. Cluster disabled based on workload

Some platforms allow switching off one of the clusters under demand via the Operating System (OS), which entails a decrease on power consumption, as shown in Figure 3.7. This is the case for the ODROID platform, not being possible to do the same on the JUNO board. Policy TS3 is similar to policy TS2, but in addition to deactivating the big cluster to the task scheduler, it switches it off completely. As the Linux Kernel does not allow powering off the core number zero in our platform, experiments related with switching off the LITTLE cluster could not be performed.[2]

3.3.2. Experimental results

Opposite to FS policies, TS policies do not pre-define a specific moment of the execution in which a cluster is disabled. The experiments described below take into account different configurations of the policies, from disabling the cluster when the amount of ready tasks is 50% of the maximum amount recorded (that is, $N_{thres} = 50\%$), to disabling it when the

[2]Switching cores on/off under demand was done through the `/sys/devices/system/cpu/cpuN/online` mechanism offered by the Linux kernel.

	(m)	1024		4096		4608		5120		6144		8192	
	(b)	64	128	256	512	256	512	512	1024	512	1024	512	1024
JUNO	50%	69.4	45.8	43.4	50.8	39.4	38.5	40.9	42.1	39.2	35.2	40.3	48.3
	40%	68.2	31.3	29.4	33.3	30.6	32.7	32.7	30.5	32.8	30.4	30.2	37.9
	30%	63.4	34.6	21.1	32.5	20.9	24.9	25.0	28.1	23.1	25.4	21.8	33.8
	20%	20.4	17.9	11.4	31.7	12.0	17.3	18.4	21.0	14.6	19.1	13.1	26.7
	10%	23.1	15.0	5.3	20.0	4.9	11.5	9.6	15.1	7.7	10.3	5.6	12.9

Table 3.6: Amount of time when the LITTLE cluster is unusable for different configurations of policy TS1 (rows) and Cholesky factorization sizes (columns) in a JUNO platform.

amount is only at 10%. Note that disabling the cluster when the current number of ready tasks is, for example, half of the maximum amount recorded does not imply that the cluster will be unusable 50% of the execution time.

Similar as before the Cholesky and QR factorization were used to run the experiments on both platforms, achieving similar behaviour for both applications.

Policies TS1 **and** TS2

Table 3.6 shows the percentage of time in which the LITTLE cluster is unusable for policy TS1, depending on the configuration of the policy and problem dimensions.

The experiments reveal that N_{thres} has a high impact on the final performance, independently of the cluster (LITTLE in TS1 or big in TS2) which is affected by the policy. Table 3.7 shows the performance achieved by each policy for different N_{thres} values and both platforms on a specific configuration. In general, observe how, the policy TS1 achieves worse results than TS2 in the JUNO platform. Similar behaviour happens on the ODROID board, except for the first columns of the table that the behaviour is the opposite one. This difference in the JUNO platform can be explained in terms of the number of cores on each cluster: 2 cores in the big cluster, and 4 in the LITTLE. Switching off the LITTLE clusters makes only 2 big cores available to execute the ready tasks, decreasing drastically the available parallelism in the platform. On the contrary, disabling the LITTLE cluster on the ODROID board still offers 4 big cores to the applications, having a lower impact on the performance. As an additional observation, we can conclude that, although a big core on the JUNO platform obtains better performance than a LITTLE one, using the four of them outperforms the results obtained by an unique big core. In general, both policies exhibit worse energetic results than not using any policy. Whereas policy TS2 has similar energy efficiency results than PBOTLEV, the results obtained when TS1 is used are worse than when not using it.

Table 3.8 shows the improvement of GFLOPS/Watt obtained when policy TS2 is compared with a normal execution (policy PBOTLEV). Although this policy does not achieve an improvement in energy efficiency, it obtains similar energy-efficiency measurements with lower overall power consumption, making this policy, together with policies FS2 and FS2', good candidates for scenarios where the power consumption is limited.

	JUNO					ODROID				
	50%	40%	30%	20%	10%	50%	40%	30%	20%	10%
TS1	52.2%	59.6%	67.3%	78.3%	85.9%	70.5%	75.9%	82.2%	87.6%	91.4%
TS2	58.2%	63.0%	69.9%	76.1%	84.8%	62.0%	67.9%	74.6%	82.0%	90.3%

Table 3.7: Performance obtained for policies TS1 and TS2 for different N_{thres} values for a Cholesky factorization respect to the performance obtained by PBOTLEV.

	(m)	1024		4096		4608		5120		6144		8192	
	(b)	64	128	256	512	256	512	512	1024	512	1024	512	1024
ODROID	10%	-3.7	4.1	-0.5	-0.3	-0.5	-0.4	-0.3	0.1	-0.3	-0.1	-0.4	-0.2
	20%	-4.0	2.0	-0.3	-0.2	-0.5	-0.2	-0.3	0.0	-0.3	0.0	-0.3	-0.1
	30%	-4.0	-1.0	-0.2	-0.1	-0.3	-0.1	-0.1	0.0	-0.2	0.0	-0.3	-0.0
	40%	-3.7	1.6	0.1	-0.1	-0.2	0.0	0.0	0.0	-0.1	0.0	-0.2	0.0
	50%	-4.0	-1.5	0.2	0.0	0.1	0.0	0.1	0.0	0.0	0.0	-0.1	0.1
JUNO	10%	-1.5	-1.9	-0.2	0.4	-0.3	0.1	-0.0	-0.1	0.1	0.0	-0.1	0.4
	20%	-1.3	-1.6	-0.3	0.2	-0.4	0.0	-0.1	0.0	0.1	0.0	0.0	0.4
	30%	-1.4	-1.8	-0.1	0.3	-0.3	-0.1	-0.1	-0.0	0.0	0.0	-0.1	0.4
	40%	-1.4	-1.6	-0.2	0.3	-0.2	0.1	-0.1	-0.0	0.0	0.0	-0.1	0.3
	50%	-1.2	-1.5	-0.2	0.3	-0.1	-0.0	-0.0	0.0	0.1	0.1	-0.1	0.3

Table 3.8: Energy efficiency obtained by TS2 when compared with an execution without any policy (PBOTLEV) for a Cholesky factorization in both platforms.

Policies TS3

Policy TS3 does achieve an improvement in terms of energy efficiency on most of the tested configurations. Figure 3.8 shows the results obtained when this policy was applied for different problem dimensions on a Cholesky factorization. The application of the policy attains an improvement of up to 17.1 %. Table 3.9 shows the improvements for each configuration in terms of GFLOPS/Watt.

Although policies TS2 and TS3 exhibit similar behavior (policy TS2 does not use big cores meanwhile policy TS3 switches them off), the performance obtained is lower for policy TS3. This overhead is probably caused by the OS when it migrates the processes running on a big core to a LITTLE one when a complete cluster is switched off (and similarly when it is switched on again). However, due to the considerable decrease in power consumption when the cluster is off (as shown in Figure 3.7), the decrease in performance does not entail a big impact on the overall energy efficiency.

3.4. Conclusions

In this chapter we have explored a number of ways to extend an asymmetry-aware scheduler to optimize the energy efficiency of task-parallel applications, focusing on ARM big.LITTLE systems-on-chip. To do so, the described approaches base their decisions on

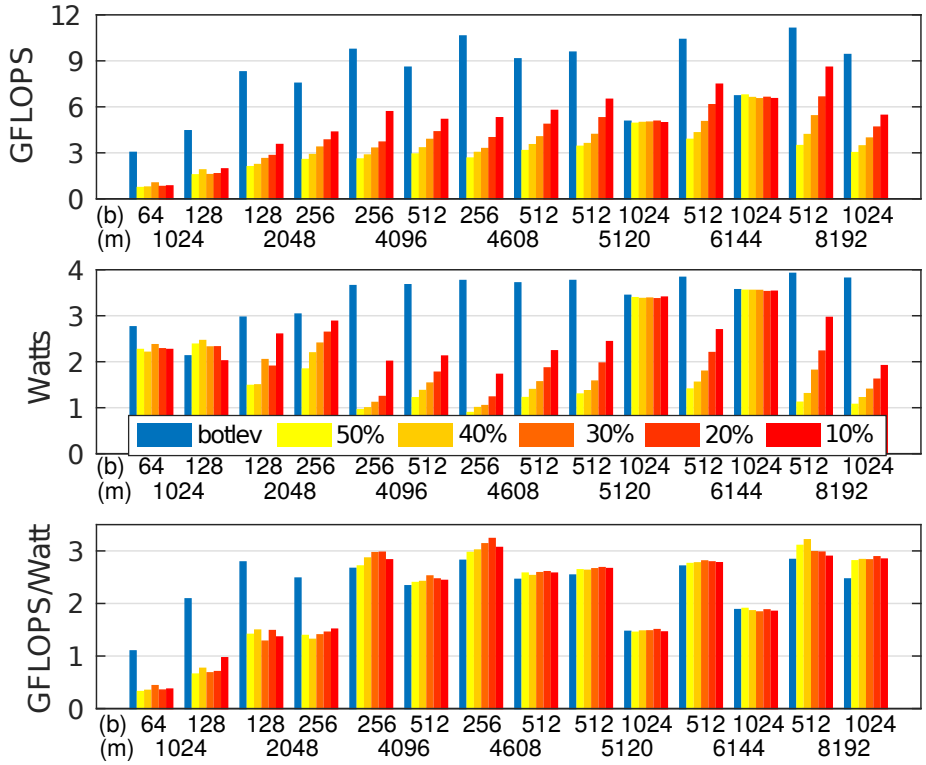


Figure 3.8: Experimental results for different TS3 configurations applied to multiple matrix sizes.

the internal status of the runtime: the classification of the different ready tasks in critical and non-critical, and their evolution during the execution time.

When testing the proposed approaches in different asymmetric architectures and different linear algebra applications, a number of insights have been extracted, namely: (I) scaling the frequency of the LITTLE cluster does not have a positive effect on the energy efficiency, but a reduction in average power consumption is constantly achieved, (II) scaling the frequency of the big cluster does achieve considerable improvements on energy efficiency, increasing it up to 29.3 %, and (III) we have demonstrated that disabling the use of one of the clusters in some moments of the execution also achieves a decrease on power consumption, but not in energy efficiency, unless the switching off of the whole cluster is supported by the hardware and OS, with improvements on energy efficiency of up to 17.1 %.

However, although the proposed policies have achieved good results for this kind of platforms, extending them to other multi-core processors is not a trivial tasks. In the next chapter we describe new strategies targeting DVFS on modern multi-core servers to improve performance on power-constrained scenarios, considering the execution of multiple parallel applications simultaneously.

(m)	1024		4096		4608		5120		6144		8192	
(b)	64	128	256	512	256	512	512	1024	512	1024	512	1024
10%	-4.98	-5.02	-0.16	-0.08	-0.02	0.01	0.02	-0.02	-0.01	0.02	0.24	0.33
20%	-4.95	-4.70	0.00	-0.05	0.03	-0.03	0.01	0.01	0.01	-0.02	0.35	0.35
30%	-4.71	-4.83	0.12	0.07	0.16	0.04	0.05	0.01	0.05	-0.05	0.13	0.33
40%	-4.92	-4.44	0.15	0.02	0.29	0.07	0.09	0.03	0.04	-0.01	0.13	0.41
50%	-4.89	-3.92	0.06	0.02	0.14	0.06	0.09	-0.01	0.04	-0.03	0.05	0.37

Table 3.9: Energy performance improvement (in GFLOPS/Watt) for different TS3 policy configurations compared with a normal execution using BOTLEV (policy PBOTLEV) for a Cholesky factorization.

4

Power budget management for runtime-based applications

In the previous chapter we demonstrated the feasibility of enriching a runtime system with a strategy aware of the internals of the scheduling state, specifically keeping track of the amount of ready tasks at each execution point, together with a criticality-based classification of tasks. Experimental results revealed it to be an effective mechanism to transparently improve the energy efficiency of applications in a big.LITTLE architecture. Specifically, we developed strategies and policies that decreased the resources granted to the application (in terms of frequency and number of cores) if the amount of work is considered low enough, increasing them otherwise under high demanding situations.

In this chapter, we address a totally different, yet complementary question: given a fixed amount of resources granted to an application (in our case, in terms of maximum power budget and number of cores), *how can runtimes and/or resource managers be extended to maximize the overall application performance without exceeding the resources assigned a priori?*

In order to answer this question, we proceed with a two-step approach:

- **BAR: Dynamic intra-application power budget re-distribution.** First, we investigate and propose an extension of a task scheduler called BAR (*Power Budget-Aware Runtime Scheduler*) to carry out a dynamic power budget redistribution between workers threads belonging to the same application, targeting performance maximization under tight power cap limits. To do so, the number of ready tasks is constantly monitored at runtime detecting periods with idle workers, and performing a re-distribution of the power budget across active workers. Ultimately, the distribution of budget allows increasing the frequency of cores where active threads are bound, while setting minimum frequency to the idle ones, improving performance without exceeding the assigned power cap.
- **BACO: Dynamic inter-application power budget re-distribution.** Second, we extend the intra-application approach by considering multiple simultaneous applications, allowing a global power budget redistribution, not only between workers

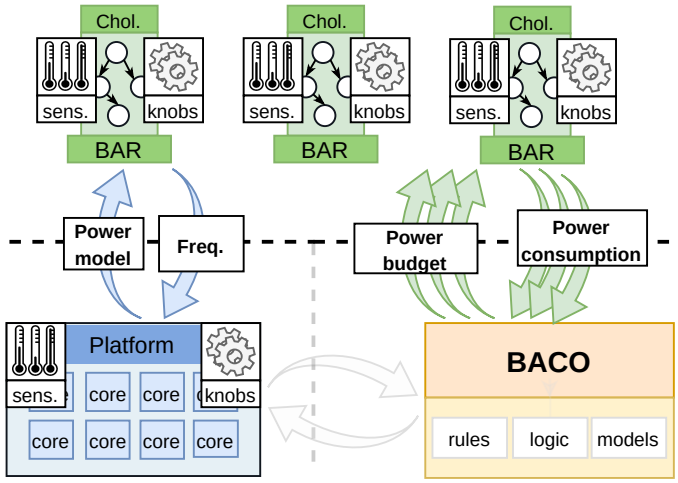


Figure 4.1: General overview of the system described in this chapter.

within the same application, but also across applications. We introduce a centralized resource manager, BACO (*Power Budget-Aware Co-scheduler*) aware of the internal status of each runtime/application running BAR, providing an optimal distribution policy of the power budget among them.

Section 4.1 motivates how a strategy aware of the number of active and idle workers can improve the performance of the applications by performing a proper distribution of the power budget. Section 4.2 shows an overview of the proposed system, while Section 4.3 and Section 4.5 show a detailed description of BAR and BACO respectively. Section 4.4 and Section 4.6 show some experimental results for BAR and BACO, when compared with other state-of-the-art approaches, as well as with an optimal approach. Finally, Section 4.7 shows some conclusions and final remarks.

Figure 4.1 shows a general overview of the relation of our proposal with the running applications and platform. As shown there, our proposal receives the metrics from the runtimes and send back its decisions, being the runtimes in charge of applying the changes accordingly to the platform (in this case by means of frequency changes).

4.1. Power budget management. Motivation and opportunities

In the previous chapter we explored how to extend the NANOS++ runtime to improve energy efficiency on big.LITTLE platforms. Our approach determines how to change the frequency or to turn specific cores off based on the number of ready tasks tracked by the runtime, and their classification between *critical* and *non-critical* tasks. Taking the decision based on the internal status of the runtime (number of ready tasks and their classification, in this case), instead of specific application metrics makes the approach perfectly valid for any application running on top of a runtime, without any modification on the user's applications.

The previous approach, specifically designed for asymmetric platforms, is mainly supported by two characteristics: (I) the frequency is the same across all the cores belonging

to the same cluster, and (II) the classification of tasks into critical and non-critical is done through the CATS (BOTLEV) algorithm. However, both premises are not valid in modern multicore servers. Indeed, modern processors support the configuration of each core with a different frequency at the same time, and CATS algorithm was specifically designed for asymmetric architectures were the behaviour of one application can change drastically if a given task is executed in one cluster or the other. In addition, to consider the number of ready tasks in the queues by its own does not represent accurately the amount of work to be performed, as it is necessary to compare it in relation with the number of workers in the system.

Targeting modern multicore processors, in this chapter we propose a new approach built on top of NANOS++ to achieve maximum performance where the instantaneous power consumption is limited to a certain level (that is, a tight *power cap* is imposed). Our approach bases its decision on the number of ready tasks *compared with the number of workers*, blocking idle workers when needed (and therefore saving energy), and waking up those blocked workers if the amount of work surpasses a given threshold. Besides, individual core frequency is modified based on the number and affinity of the active workers, increasing performance without exceeding the power cap.

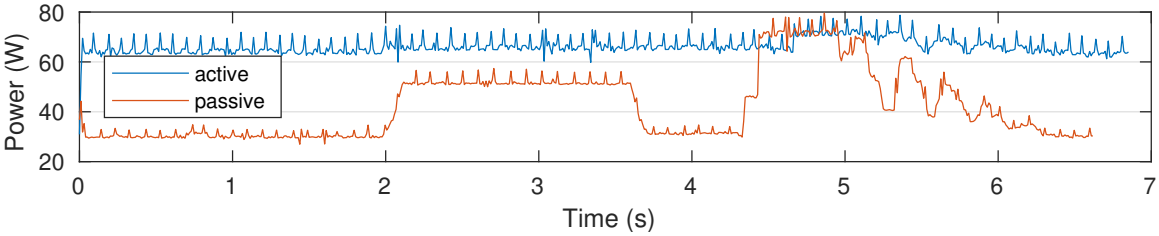
4.1.1. Idle workers management

Consider the task-based, block-oriented implementation of a Cholesky factorization described in the previous chapter, that will also be employed as a driving example throughout the chapter. As all the ideas described next affect runtimes and not applications, they can be extrapolated to any task-based application without any significant modification. In addition, they are general enough to be integrated into any other runtime with minimum effort.

The first plot in Figure 4.2 shows the execution of a Cholesky factorization of a matrix configured as: $m \approx 10000$, $b = 1024$, $s = 9$, on MAKALU (see Section A.1), using 20 worker threads running at 1.6 GHz. Each row represents the status of a worker during the execution of the application, colored in *gray* if the worker is in *idle status* (i.e., it is not executing any task), and colored in *orange* if it is *active* (i.e., executing a task). As observed, the execution can be split into three different phases: a middle phase where the factorization is carried out, being all the workers active; and two phases (at the beginning and the end) where the parallelism decreases, without enough tasks to feed all the workers. The first phase, purely sequential, occurs during the reservation of the memory for the different matrices internally used, the data initialization filling those matrices with random values, and the transformation of the original matrix into a blocked-form matrix. The last phase is a consequence of the parallel algorithm, directly related to the decreasing width (amount of potential parallelism) of the DAG associated with the problem. These phases are of special interest for us. Depending on the final goal of the system, two main strategies can be followed to manage these idle threads:

- *Active waiting (polling)*: Idle workers are kept on an infinite loop constantly checking if there is a new task to be executed. This strategy guarantees maximum performance executions (as there is no delay associated with the process of blocking and waking up the workers), at the expense of a higher energy consumption.

(a) Worker status during the execution. Each row represents a different worker, colored in gray if it is in idle status, and orange if it is active.



(b) Energy consumption when the idle threads are blocked in an active wait (blue) or passive wait (red).

Figure 4.2: Worker status and energy consumption for a Cholesky factorization with $m \approx 10000$, $b = 1024$, $s = 9$. On the top, the status of each worker during the execution (active/idle). On the bottom, instantaneous energy consumption when idle workers are blocked in an active and passive wait. On the passive wait, idle workers are blocked through the default NANOS++ mechanisms (`--enable-block` option).

- *Passive wait (blocking).* In this strategy workers are blocked by a synchronization mechanism (e.g., semaphores, mutexes or conditional variables among others). Depending on the overhead of the selected mechanism, performance can be affected and hence reduced. However, contrary to the previous approach, and depending on the number of idle workers during the execution (and therefore, depending on the application), the reduction in energy consumption is not negligible.

The plot on the bottom of Figure 4.2 reports the instantaneous power consumption when idle workers are blocked in an active and passive wait (blue and red lines, respectively). As expected, in the middle phase where all the workers are active, the power consumption is maximum and equal in both approaches, decreasing on the red line as the number of active workers decreases. Similarly, in the first phase where only one worker is active (serial phase), a considerable reduction in power is achieved (greater than $2\times$). As a side note, observe how, even only one worker is active in this phase, the instantaneous power is not constant (power increases in the interval between 2 and 3.5 seconds). This period corresponds to the matrix initialization with random values (through the `larnv` call) and the additional operations needed to ensure the matrix is definite-positive (required by the Cholesky factorization). The periods with lowest power consumption correspond to memory allocation calls (via *malloc*), and data copies between matrices to transform the original matrix into a blocked form.

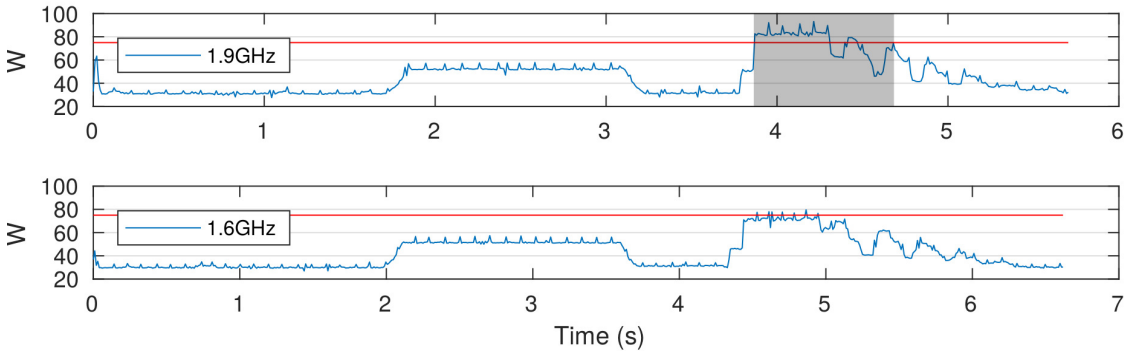


Figure 4.3: Instantaneous power consumption of a Cholesky factorization ($b = 1024$, $s = 9$, $m = 9216$) using 20 worker threads at 1.9 GHz (top) and 1.6 GHz (bottom). The red line represents the 75 W value.

In summary, blocking idle workers in a passive wait achieves a considerable reduction in power consumption. In a scenario where maximum performance is required, but a strict power cap exists, this power not consumed by idle workers can be redistributed between the active workers, increasing the operational frequency, and therefore the overall performance, and still satisfying power restrictions. This redistribution of power budget can be done between workers of within an application, or between workers of different applications running concurrently, as described next.

Power budget redistribution between workers within the same application. Figure 4.3 shows the power consumption of the Cholesky factorization described before with idle workers blocked in a passive wait when the frequency is configured at 1.9 GHz and 1.6 GHz (top and bottom, respectively). Consider a hypothetical power cap of 75 W, represented by the red line in the figure. On one hand, executing the application at 1.6 GHz guarantees an execution always below the power cap. On the other hand, an execution at 1.9 GHz achieves maximum performance, exceeding only the limit on the phase where all the workers are active (area marked in gray in the figure), being below the threshold on the other phases where the number of active workers decreases.

Combining both traces, an intelligent approach should set the maximum frequency when the number of active workers is low enough not to exceed the power limit, achieving maximum performance, and reduce the frequency as the number of active workers increases, not violating the power cap. However, this strategy requires a precise control on per-thread execution status and per-core frequency, as well as a precise estimation of the power consumption by each worker at each selectable frequency. In the next section, we propose a heuristic approach implementing these ideas to manage a dynamic distribution of the power budget and frequency between workers, achieving optimal executions in performance, without violating the different power caps.

Power budget redistribution between different simultaneous applications. Consider now an execution of two simultaneous Cholesky factorizations of different block sizes, not ex-

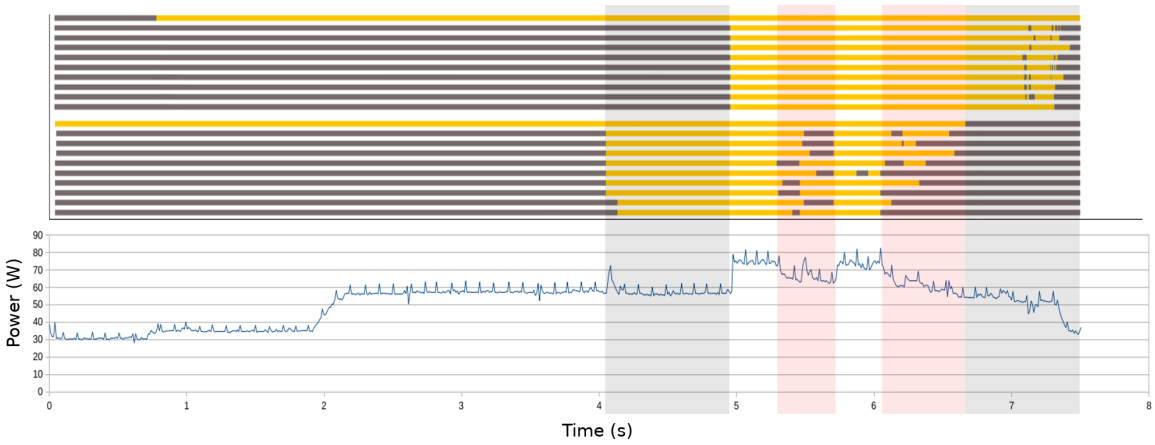


Figure 4.4: Thread status and power consumption of two Cholesky factorizations running simultaneously with 10 workers each at 1.7 GHz. The ten first rows correspond to a factorization of a matrix configured as $b = 512$, $s = 19$. The other ten rows correspond to a similar factorization configured as $b = 1024$, $s = 9$. Orange and gray colors mean active and idle status respectively.

ecuted at the same time. Figure 4.4 represents an example of this situations where two different Cholesky factorizations ($b = 512$ and $b = 1024$) are run with a 1s offset between them. The plots represent the thread status of each application, as well as the system-wide power consumption.

In this scenario, a static distribution of the power budget between applications does not guarantee the maximum performance, and therefore demanding a more advanced strategy redistributing the power budget between applications. Indeed, observe how, in the moments of the execution where an application does not have all the workers active (red region in the plot), the power consumption decreases, as it is not used by any application. In this moments, an intelligent redistribution of that budget to the other application will allow it to increase the frequency, and therefore to increase the performance, at the same time the power is not wasted, and the power cap is not exceeded. Similarly, if an application starts or finishes its execution before the other (gray zones in the plots), a redistribution of the power budget can increase the performance of the applications that can benefit from it (for example, if an application is in a parallel phase).

However, the proper redistribution of the budget is not trivial, requiring a global and intelligent approach able to know the status of each application during the whole execution, performing an online redistribution of the budget between them, as described next.

4.2. Resource management for asymmetric power budgeting: a two-level approach

Armed with the previous motivations, in the following we propose strategies and actual implementations of resource management schemes embedded into an application-based runtime scheduler (Nanos++) and an ad-hoc co-scheduler (see Appendix B). Our solutions

target a common scenario in today's HPC or data-center nodes, in which overall computing resources need to be –symmetrically or asymmetrically– distributed across applications, and afterwards individually exploited in an optimal manner by them. In our case, power budget is the resource to be globally distributed, while individual core number and frequency are the knobs to be dynamically managed at application level.

It seems natural, hence, pursuing the target by following a two-level approach, with a common goal but different specific strategies depending on the specific level.

Application-wide level Given a granted amount of resources (in our case, in the form of a tight limit in terms of power budget and number of cores), an application should ideally apply internal policies to optimally exploit the granted resources. Task-parallel applications are usually characterized by the existence of phases with different amount of potential task parallelism. In this scenario, opportunities for power budget redistribution arise in those phases in which parallelism is scarce, and a full leverage of the underlying resources is not possible, mainly due to insufficient core occupation. Other situations, not studied in this thesis, could include bandwidth occupation or LLC partitioning, for example, following similar approaches.

Moreover, this type of policies should be applied in a way to be *portable* across systems and *transparent* (non-intrusive) for the user. As task-parallel applications usually rely on a lower-level runtime task scheduler, it seems logical to integrate them within the runtime.

System-wide level At a higher level, system-wide restrictions on power consumption to meet SLA policies, for example, can also be applied. Under circumstances in which the amount of resources granted to individual applications/users is asymmetric, or granted resources are wasted at a given execution point for an application, an intelligent power budget distribution also becomes mandatory.

Following a similar idea as that for application-wide requirements, a piece of software (usually a centralized resource manager or co-scheduler) should be in charge of *autonomously* and *dynamically* applying policies and heuristics that assure a proper distribution of the available budget.

4.2.1. BAR + BACO: an overview

Our solution to the problem is based on a two-level strategy, and actually implemented within two isolated (while highly coupled) pieces of software, namely:

BAR**: Power Budget-Aware Runtime Scheduler**, is an extension of the NANOS++ runtime task scheduler, that embeds strategies and techniques for autonomous and dynamic power reduction and budget redistribution among worker threads. The solution is based on an intelligent idle worker thread management and core frequency selection based on power models (see Section 4.3).

BACO**: Power Budget-Aware Co-Scheduler**. An ad-hoc development that manages resources at a higher level (system-wide level), receiving application registration requests and dynamically offering power-budget shares between running applications based on the information dynamically received from them.

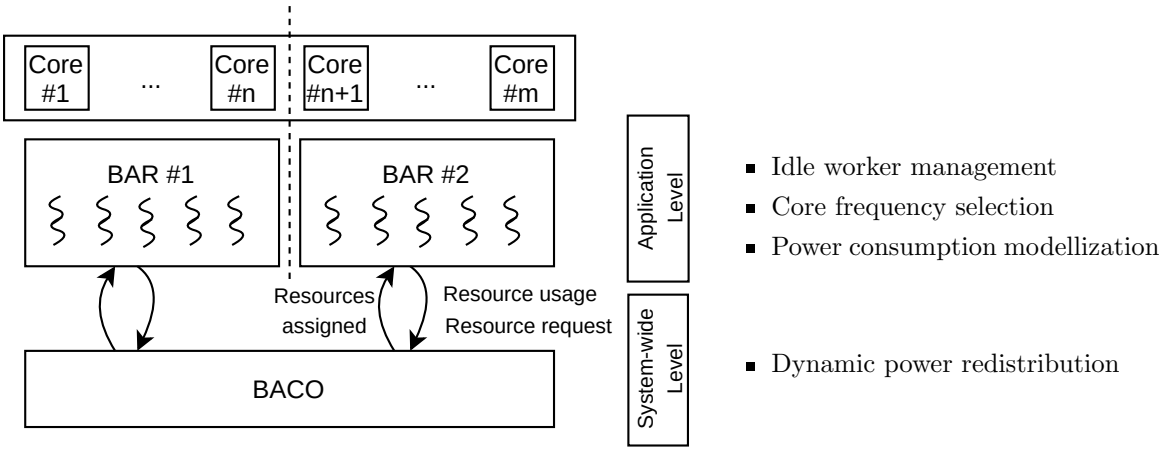


Figure 4.5: Diagram showing the coexistence of two BAR applications interacting with a BACO instance.

Figure 4.5 shows a general overview of the BAR+BACO tandem deployed on a hypothetical DVFS-capable multi-core server with two managed applications. Note that each element is responsible of different (while complementary) tasks, and information flows bidirectionally between BACO and each BAR application. Specifically, each registered application will constantly monitor the status of its worker threads, modifying the core frequency and communicating the new power needs to the BACO instance, while the co-scheduler will be in charge of responding to these interactions by redistributing the available power budget between those applications that can benefit from it.

In the following, we offer more details regarding each element of the infrastructure.

4.3. BAR. Runtime support for intra-application power budget management

4.3.1. Budget re-distribution strategy

BAR introduces a new approach to achieve high performance executions on a power-constrained environment. Our approach is based on the ideas described above: a precise control on the mechanism to detect and block idle workers, a fast and effective way to wake up blocked workers, and an estimation of the power consumption of each workers at each possible frequency, as well as at idle state. As soon as a worker becomes idle, it is blocked in a passive wait, estimating the saved power consumption, and increasing the frequency of the rest of the active workers accordingly. As soon as new tasks are created, idle workers are woken up, decreasing the frequency of active workers appropriately without exceeding the cap. The approach is described next in terms of the NANOS++ runtime, but the presented ideas are generic enough to be implemented in any other task-based runtime with minimum effort.

As described in Section 3.1.1, task-based runtimes are built on top of a pool of threads called *workers*, each one typically bound to a different core. Workers execute an infinite loop

until all tasks are completed and the application finishes. In this loop, each worker checks if a task is ready to be executed, and runs it. If there are no ready tasks available, the worker becomes idle. Upon task finalization, its output dependencies are released; when a new task is ready to be executed (i.e., all its input dependencies have been already satisfied), the task is inserted into a queue (in the following, *readyQueue*) to be executed as soon as a worker thread becomes available. The order in which ready tasks are executed differs between different scheduling strategies, although our approach is perfectly valid for all of them, and hence agnostic of the scheduling policy in place.

All the BAR logic is incorporated in this loop. Specifically, Algorithm 4.1 shows a pseudocode of our approach, which is based on four main ideas: (I) when and how idle workers wake up (Section 4.3.2), (II) how to determine if new tasks are ready to be executed or not (Section 4.3.3), (III) when and how an idle thread needs to be blocked (Section 4.3.4), and (IV) how to properly modify the frequency not to exceed the power cap (Section 4.3.5). This routine is invoked by each worker every time it finishes the execution of a task, and returns the new task to execute by the worker (if it exists), or blocks the worker (if no tasks are ready). At this point, all the output dependencies have been released and all the new ready tasks have been inserted into *readyQueue*.

Algorithm 4.1: BAR logic. Main worker loop.

```
1  begin
       /* Wake up idle workers?                                          */
2      if ( nReadyTasks > 1 and nActiveWorkers < nWorkers) then
3          maxWorkers ← getMaxThreads(Budget) - nActiveThreads
4          wakeUp_threads( min( nReadyTasks - 1, maxWorkers));
5      end
       /* Try to get a ready task to execute                             */
6      task ← readyQueue.pop()
7      spins ← maxSpins // User defined.  For example, 60000
8      while (task = none and spins > 0) do
9          task ← readyQueue.pop()
10         spins ← spins - 1
11     end
       /* Block worker?                                                  */
12     if (task = none) then sleepThread(thread.self) ;
13     return wd
14 end
```

4.3.2. Waking up idle workers

Upon termination of a task, and before it fetches a new task to execute, a worker determines if new workers should be woken up. To wake up an idle worker, two conditions need to be met simultaneously: there are at least 2 tasks ready to be executed, and of course, there is at least one blocked worker (line 2). Assuring at least two ready tasks allows both workers (the current one, and the one to be woken up) to execute a task, and therefore, avoids to wake up a thread and block it again immediately, reducing overhead. The number

of workers to be woken up is determined by the number of ready tasks in *readyQueue* and the maximum number of workers that can run concurrently without exceeding the established power cap.

The following equation is used to determine the maximum number of simultaneous active workers (line 3) for a specific power budget:

$$\text{Budget} \geq \text{nActive} \cdot \text{Power}_{\text{f\_min}} + \text{nIdle} \cdot \text{Power}_{\text{idle}} \implies \tag{4.1}$$

$$\text{nActive} = \left\lfloor \frac{\text{Budget} - \text{nWorkers} \cdot \text{Power}_{\text{idle}}}{\text{Power}_{\text{f\_min}} - \text{Power}_{\text{idle}}} \right\rfloor \tag{4.2}$$

where *Budget* is the current power budget assigned to the application, *nActive* and *nIdle* are the number of active and idle workers respectively, and $\text{Power}_{\text{f\_min}}$ and $\text{Power}_{\text{idle}}$ are the estimation of the power consumption of an unique thread at minimum frequency and idle state respectively. This equation determines the maximum number of workers that can run simultaneously, considering they run at a minimum frequency. Experimentally, we have determined that, for our tested applications and scenarios, having a high number of workers running at a low frequency achieves better results than a lower number of workers at a higher frequency. However, this policy can be modified if there is any other goal in the system (for instance, to minimize the number of active cores). Note that this function determines the maximum number of workers able to run concurrently without exceeding the power cap, but not the optimal frequency, which is calculated at a later step, described next. At the same time idle workers are woken up, the new frequency is calculated and set accordingly to all cores running active workers (see Section 4.3.5).

4.3.3. Fetching a new ready task

Once the additional workers have been woken up and the frequency has been properly set, the worker tries to fetch a new ready task to execute. Lines 5–11 show the strategy followed. Note that this code can be executed by multiple workers simultaneously, being possible for a worker to fetch the only task in the queue, and therefore, leaving other workers without tasks to run. In this case, the worker spins around a loop to check if new tasks become ready (due to any other worker finishes a task that makes new tasks ready). This loop prevents workers to become idle immediately, avoiding the overhead associated to sleeping the thread.

4.3.4. Blocking idle threads

In the case no ready task is fetched by the worker after the spin loop, the worker becomes idle and blocks itself into a passive wait (line 12). Blocking the worker itself, instead of being blocked by another worker, reduces the time other workers spend without executing a task. Once the worker is declared as idle, and before blocking itself, the worker sets the frequency of its core to the minimum available frequency, and increases the frequency of the other active cores appropriately.

4.3.5. Core frequency selection and Power modelling

Our approach relies on an internal table storing an estimation of the power consumption of one worker running at each available frequency, completed with power consumption

when idle. This estimation is performed through a previous profiling of the different tasks, storing the maximum consumption of all of them. Storing the highest consumption of all tasks makes the approach conservative, contributing to never violate the power cap. In addition, considering the maximum power consumption makes the table valid for other DLA applications without need of additional profiling. However, other approaches can be taken.

Every time the number of active threads varies (i.e., a worker is blocked or woken up), the running frequency of the active workers is modified properly, choosing the one which maximizes power consumption (i.e., the highest one), without exceeding the power cap as follows:

$$\arg\max_{f \in Freqs} \left\{ P = \text{nActive} \cdot \text{Power}_f + \text{nIdle} \cdot \text{Power}_{\text{idle}} \;\middle|\; |\text{Pcap} - \text{P}| \leq \text{offset} \right\}$$

Note that the previous formulation allows to exceed the power cap by a certain value (*offset*). This value, tunable by the user or system administrator, offers a mechanism to finely approximate the actual power consumption of the application to the power cap, and trading off possible errors in the power estimation.

This internal table storing all the estimations for the different frequencies is filled through a previous profiling of the different tasks of the application. To do that, all the different tasks are executed individually while power consumption is recorded at all the different available frequencies, as well as the power consumption of a worker blocked at minimum frequency. As the final goal of the approach is never violate the power cap, the maximum consumption among all the tasks is selected. On one hand, this ensures executions always below the power cap, but on the other hand, it is possible to overestimate the power consumption, and therefore not achieve the maximum performance. If the power cap restriction is relaxed, other alternatives can be considered as taking the average consumption of all the tasks, or even the minimum one. In addition, previous works have explored the idea of doing the estimation based on the current tasks being executed at each moment. For example, authors in [145] profile and store the execution times of different versions of the same tasks, estimating the best version to execute at each moment.

Once this table is filled, power consumption can be estimated as

$$P = \text{nActive} \cdot \text{Power}_{\text{currFreq}} + \text{nIdle} \cdot \text{Power}_{\text{idle}}$$

where *nActive* and *nIdle* represents the number of active and idle workers respectively, and $\text{Power}_{currFreq}$ and $\text{Power}_{\text{idle}}$ are the power consumption of a thread running at *currFreq* or idle state respectively. Observe that, although our approach runs all the active workers at the same frequency, this estimation can be easily adapted to other formulations with different workers running at different frequencies.

Although this model is perfectly valid, and achieves high-quality estimations (as shown later), power consumption of turbo frequencies on modern processors cannot be estimated through it. When turbo is enabled, power consumption, as well as effective frequency, does not depend only on the current frequency set, but also on the number of simultaneous active cores, being impossible to make an estimation of each core individually. Nevertheless, our approach is model-agnostic, so it is still valid for a model that can consider turbo frequencies.

4.4. Experimental results for BAR

To test our approach, we run BAR on two different scenarios, comparing it with comparable state-of-the-art alternatives. On the first scenario, only one application was run with different power caps, deploying as many workers as physical cores exist. On the second scenario, two applications were run simultaneously, each with a different power cap assigned, to show how our approach is able to deal with *asymmetric power-budget scenarios*. In addition, different application configurations were tested to proof the validity of our approach in different scenarios with different amount of idle workers.

The experimental results include comparisons between the following strategies:

BAR: This is our proposal. Given a strict power cap, the system dynamically modifies the frequency of the active cores based on the number of idle workers at each execution point. It also performs the process of detecting if there are enough tasks for all the workers, sleeping and waking up idle workers as needed.

Breadth First (BF-FREQ): This is the default strategy implemented in NANOS++ and used as our comparison baseline. Tasks are executed in a First-In-First-Out (FIFO) fashion, with idle workers blocked in an *active wait (polling)*. For the sake of fairness in the comparison, given a specific power cap, all cores were set to the maximum frequency that guarantees the maximum power cap is not exceeded.

RAPL with active wait (RAPL-ACT): This strategy implements a hardware-based power capping strategy on top of the NANOS++ runtime. Similar to the previous strategy, it implements a breadth-first scheduling policy, but frequency is set to the maximum (i.e., 1.9 GHz) before execution. In this case, RAPL is used to dynamically modify the frequency and not exceed the power cap. RAPL is a mechanism introduced by Intel on the latest architectures to limit the power consumption via hardware through an automated DVFS mechanism.

RAPL with passive wait (RAPL-PASS): This is considered the *optimal strategy* and is used to check how far our proposal is from an *optimal* solution. Similar to the previous strategy, it implements the RAPL hardware-based power-capping mechanism on top of the NANOS++ runtime and the breath-first scheduling policy. In this case, however, idle workers are *blocked* in a passive wait using the internal mechanism offered by NANOS++.

Note that RAPL-based solutions are not always available (power capping capabilities need to be provided by the manufacturer). Hence, demonstrating that our approach is close to (or even mimics) RAPL-based solutions is mandatory and a demonstration of the feasibility of our software-based approach.

4.4.1. Experimental setup

The described strategy has been implemented in MAKALU (see Section A.1), a real server comprising a modern 20-core Intel CPU. Hyperthreading technology was disabled for all the experiments. The NANOS++ runtime was chosen to implement BAR as it offers a mechanism to easily extend the scheduling policy and implement alternative strategies.

Freq (GHz)	*idle*	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9
Power (W)	1.47	2.97	3.09	3.22	3.35	3.43	3.56	3.79	3.97	4.24	4.45

Table 4.1: Power estimation for one worker at different frequencies on MAKALU.

Our mechanism was implemented on top of the breath-first policy (the default policy in NANOS++). POSIX unnamed semaphores were used to block idle workers in a passive wait. The same task-based parallel Cholesky implementation as in Chapter 3 was chosen to run the experiments, as it is a representative example of other linear algebra operations. In addition, the blocked parallelization of the Cholesky factorization is carried out in terms of other linear algebra operations widely used in most of the scientific applications (for instance, a matrix-matrix multiplication, *gemm*, or a symmetric rank-b matrix update, *syrk*). Frequencies from 1.0 GHz to 1.9 GHz were selected to run the experiments, as greater (*turbo*) frequencies are not handled by the used model, as previously described. The estimated power values are shown in Table 4.1. Experimentally, we have observed a maximum difference of 4.46 W for the whole socket when measuring the power consumption through RAPL and our estimations (TDP=125 W). *offset* parameter on the power model was experimentally set to 0.8.

Each experiment was repeated 5 times, showing average values in the following. The results are reported in terms of the execution time of the sequential and parallel phase, global speed up and power and energy consumption. Power consumption is reported in terms of a moving average, in order to reduce noise in power measurements.

All the experiments were carried out over a matrix randomly initialized of size $m \approx 10000$. Three different block sizes were used covering all the possible scenarios: $b = 512$, $s = 19$ ensures enough tasks for all the threads during most of the execution time, $b = 1024$, $s = 9$ produces enough tasks to feed all workers in the most demanding period, decreasing the number of tasks as the factorization is completed, and $b = 2048$, $s = 4$ does not generate enough tasks for all the workers at any moment of the execution.

4.4.2. Preliminar analysis of BAR performance

Figure 4.6 shows a detailed trace of our approach when executing a Cholesky factorization configured as $s = 9$ and $b = 1024$, with a power cap of 63 W ($\approx 50\%$ TDP), using 20 worker threads (i.e., all the available cores of the processor). The first plot represents the number of workers set active by our approach, while the second one shows the frequency configured on those workers at each moment of the execution. The third plot shows the instantaneous power of the socket measured through an external library (see Section A.2).

As observed, our approach is able to detect the sequential phase at the beginning of the execution, having only one thread active and setting its associated frequency at maximum (1.9 GHz) to speed up this phase and achieve maximum performance. Once this phase is finished, and the parallel phase starts, all the workers are woken up as there are enough tasks to feed all of them. At the same time workers are woken up, frequency is reduced to not violate the power restrictions. Contrary to the previous chapter, having a power model to estimate the power consumption allows our strategy to set the frequency to 1.1 GHz in this scenario, instead of minimum frequency. As the number of ready tasks decreases, idle

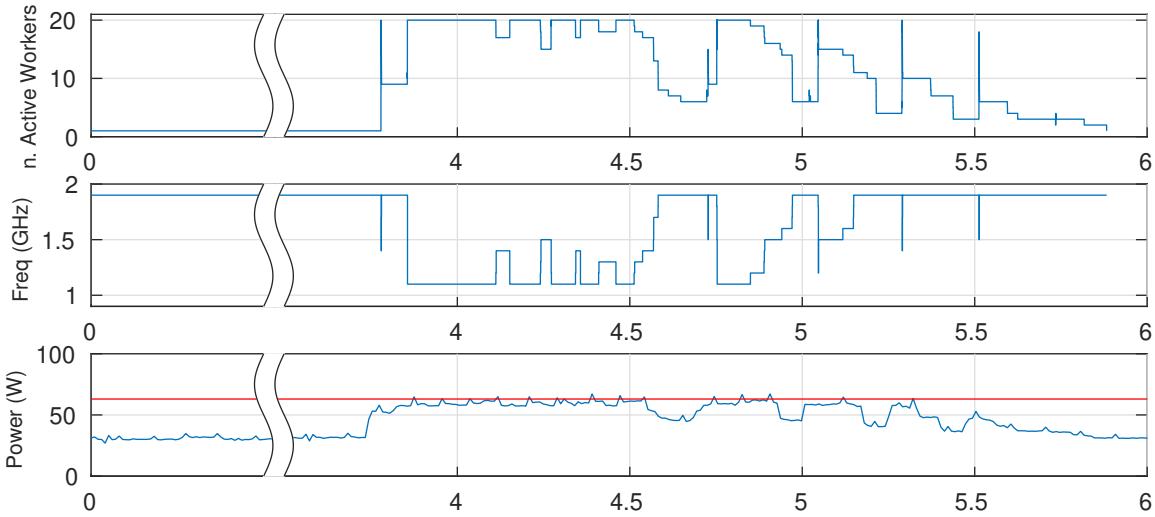


Figure 4.6: Behaviour of BAR when executing a Cholesky factorization configured as $m \approx 10000$, $b = 1024$, $s = 9$, using 20 worker threads and a power cap of 63 W (50% of the TDP). From top to bottom: number of active workers across execution, frequency set by our approach at each moment, and power consumption measured through RAPL.

workers are blocked, increasing the frequency of the active ones properly. Observe how our approach is able to always set the maximum frequency as possible, never exceeding the power cap set.

4.4.3. Scenario I: Power capping for one application

Table 4.2 shows the experimental results of our approach when compared with the other three approaches previously described, in terms of execution time of the sequential and parallel phases, and average power consumption. The experiments covered all the possible values of power caps and frequencies (the table only reports the maximum, minimum and intermediate values). For the sake of fairness, for each tested power cap, the frequency chosen for BF-FREQ was the maximum (between 1.0 GHz to 1.9 GHz) that does not exceed the power cap in our model.

In terms of performance, all approaches attain similar qualitative numbers, obtaining the better results as block size decreases, and hence, potential task parallelism increases accordingly. As described before, large block sizes do not expose enough parallelism to keep all the workers active (20 workers in the experiments). Indeed observe how, for all the approaches, reducing the block size obtains better performance than increasing the power cap (for instance, $b = 512$ and 60 W, compared with $b = 2048$ and 90 W). However, in those scenarios with a high number of idle workers, redistributing the power budget between active cores can improve performance greatly, as shown later.

b	PCAP	Freq	Time (s) - seq. phase				Time (s) - parallel phase				Power (avg.)			
			BAR	BF FREQ	RAPL ACT	RAPL PASS	BAR	BF FREQ	RAPL ACT	RAPL PASS	BAR	BF FREQ	RAPL ACT	RAPL PASS
512	60	1000	3.8	6.7	5.4	3.8	2.1	2.3	2.0	1.9	44.2	51.8	57.0	44.3
	66	1300	3.8	5.4	4.9	3.8	1.8	1.7	1.6	1.6	44.8	58.1	61.9	45.4
	74	1600	3.8	4.6	4.1	3.8	1.5	1.4	1.4	1.4	46.0	66.0	71.5	46.3
	90	1900	3.8	4.0	4.0	3.8	1.2	1.2	1.2	1.2	47.3	74.5	74.6	47.6
1024	60	1000	3.7	6.7	5.4	3.7	2.4	3.6	3.2	2.4	43.3	52.2	57.1	43.5
	66	1300	3.7	5.3	4.8	3.7	2.2	2.7	2.6	2.2	44.0	58.2	62.0	44.0
	74	1600	3.7	4.5	4.0	3.7	2.1	2.2	2.0	2.0	44.1	66.3	71.9	44.4
	90	1900	3.7	3.9	4.0	3.7	1.9	1.9	1.9	1.9	44.7	74.7	74.3	45.1
2048	60	1000	3.6	6.6	5.3	3.7	5.2	9.7	7.7	5.2	37.9	51.2	57.2	38.6
	66	1300	3.7	5.3	4.8	3.6	5.2	7.5	6.7	5.2	37.7	57.1	61.6	38.4
	74	1600	3.7	4.4	4.0	3.6	5.2	6.1	5.3	5.2	37.8	65.4	72.0	38.7
	90	1900	3.6	3.9	3.9	3.6	5.2	5.2	5.2	5.2	38.0	73.5	73.4	38.7

Table 4.2: Execution time of the sequential and parallel phase, and energy consumption of all the approaches when executing different Cholesky factorization configurations under different power caps. Only representative experiments are shown in the table, but similar behaviour applies to the rest of the experiments.

From a quantitative perspective, however, some comparative remarks should be noted. The reported results show how BAR is a perfectly valid and competitive approach, outperforming the BF-FREQ approach (the default policy implemented in NANOS++, and used as our base line) in all the tested configurations, with an average speed up of 1.2×, 1.3 × and 1.4× for $b = 512$, $b = 1024$, $b = 2048$, respectively, and a maximum speed up of 1.9× (for $b = 2048$, pcap = 60). RAPL-ACT achieves better results than NANOS++, as it is able to redistribute the power budget across the active threads. However, as idle threads are blocked in an active wait, most of the power budget is still consumed by those threads, not achieving optimal executions in terms of power. Compared with RAPL-ACT, BAR achieves average speed ups of 1.1×, 1.2 × and 1.2× for $b = 512$, $b = 1024$, $b = 2048$ respectively. In addition, BAR is the best solution in most of the configurations, with a maximum penalty of 3% in performance in the worst case, as detailed next.

A more detailed analysis of the results yield a number of interesting insights, namely

- For the experiments where the power cap is high enough to run all the cores at maximum frequency during the whole executions (that is, 90 W, 1.9 GHz), all the approaches obtain similar execution times. This scenario, equivalent to a situation with no power capping, shows how the blocking mechanism chosen (both in BAR and in NANOS++), as well as the process used to detect when a worker becomes idle and active, does not introduce any remarkable overhead in terms of execution times.
- When blocking idle workers in an active wait (BF-FREQ and RAPL-ACT), power consumption keeps constant independently of the number of idle workers (i.e., block size). However, when idle workers are blocked in a passive wait, the reduction in power consumption is not negligible. Indeed, just blocking workers in a passive wait and not

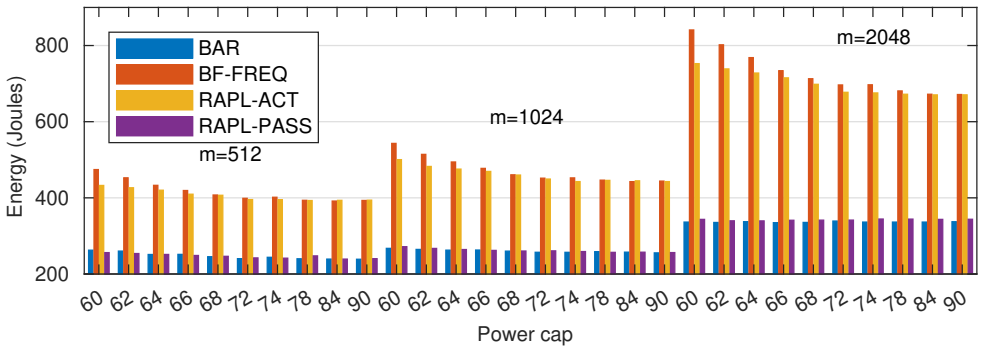


Figure 4.7: Energy consumption (Joules) for all the different configuration tested.

changing the frequency (e.g., when the power cap is 90 W), reduces power consumption up to 36 % when the number of idle workers is low ($b = 512$), and up to 40 % and 48 % when the number of idle workers increases ($b = 1024$ and $b = 2048$ respectively).

- When comparing BAR with an optimal solution (RAPL-PASS), our approach obtains similar results in performance. Specifically, for block sizes $b = 1024$ and $b = 2048$, the measured times are exactly the same, while for $b = 512$, the execution time of the parallel phase is slightly greater (both for RAPL-PASS and RAPL-ACT). In the case of executions with a high number of active workers, the introduction of a small overhead in the process can, hence, introduce a small increase factor in the attained execution time.

  As discussed before, no overhead was introduced in the process of managing idle workers, meaning the difference in time is merely caused by the frequency management process. Indeed, as RAPL bases its decisions on real measurements of the system and not in power models (as BAR does), the selected frequency at each moment guarantees the optimal execution. On one hand, basing the decisions on real measurements of the system allows to have optimal executions without exceeding the power cap. However, on the other hand, power measurements are usually carried out at the processor level (not at the core level), being impossible to run scenarios where different power caps are assigned to different applications.

  Focusing on energy efficiency (GFLOPS/Watt), both approaches achieve similar values, verifying our assumption that there is no overhead introduced by our approach, and an optimal (more sophisticated) power model would yield optimal executions by BAR.

- In any case, our approach obtains similar results to the optimal solution in most of the cases, with a minimum speed up of 0.974× in the worst case ($b = 512$, pcap=60 W).

As described before, the execution time increases as the block sizes increase due to the absence of enough parallelism to feed all the workers during most part of the execution. Similarly, as shown in Table 4.2, the average power consumption decreases too as not all the

workers are active. The amount of power saved compared with the increase in execution time will ultimately determine if reducing the block size produces savings in energy consumption. Figure 4.7 shows the total energy consumed by all the approaches in all tested configurations (different block sizes and power caps). The results show how, in general, reducing the block sizes achieves the best savings in energy consumption and fastest executions. Same as before, BAR outperforms the default configuration of NANOS++ (BF-FREQ) in all cases, with an average and maximum savings of 46% and 60%, respectively. In particular, observe how BAR achieves better results in the least favorable scenario ($b = 2048$, $pcap = 60$) than BF-FREQ for the best configuration ($b = 512$, $pcap = 90$). Similarly to the previous results, BAR obtains similar results than the optimal approach in terms of performance.

As a side note, observe how in this platform, power caps greater than 78 W do not have any impact on energy consumption, meaning that for those power consumption levels (and frequencies associated to those power levels), the increase in performance is linear to the increase in power consumption.

In summary, our proposal is able to detect where workers are idle in the execution, blocking those idle and redistributing the power budget between the others active workers (in form of frequency increase), improving performance and energy consumption, never violating the preset power cap. In particular, BAR outperforms the default policy implemented in NANOS++, obtaining an average speed up of 1.3× (with a maximum speed up of 1.9×), and an average reduction of 46% on energy consumption. In addition, BAR achieves similar results than an optimal approach, where energy measurements, as well as frequency managements are done autonomously via hardware mechanisms.

4.4.4. Scenario II: Multiple applications with different power caps

The previous scenario explored a configuration where only one application was running in the system, deploying a worker thread per available physical core. Under this situation, we showed how BAR is able to outperform the default strategy followed by NANOS++, and to obtain similar results to an optimal approach using RAPL to dynamically modify the frequency accordingly without exceeding the power cap set.

However, although RAPL achieves the optimal results thanks to the access to online power measurements, modern platforms only offer power measurements at the processor level, making RAPL approach not useful in those scenarios where not all the cores are used, or the power budget is not distributed equally across workers. On the contrary, approaches based on power models, as the one embedded in BAR, offer the possibility of estimating the power consumption *per core or per application*, allowing to manage scenarios where not all the cores are used, or not all the power is proportionally shared between them.

To proof the validity of BAR in these scenarios, we run a second round of experiments where two similar applications (same block size and 10 workers each) share computing resources, each one with *a different power cap*. Specifically, a first configuration where both applications are power constrained (33 W and 40 W, respectively), and a second configuration where one of the applications has a power restriction while the second one is not power-limited (35 W and 45 W)[1].

[1]Note that, in our platform, a power cap greater than 45 W for 10 workers is equivalent to not setting any power cap at all.

Pcap1	Pcap2	b	Time app1 (s)			Time app2 (s)			Energy (J)	
			BAR	BF FREQ	OPT.	BAR	BF FREQ	OPT.	BAR	BF FREQ
33W 1.2GHz	40W 1.7GHz	512	6.9	9.1	6.8	6.2	6.5	6.0	351.8	440.2
		1024	6.9	9.5	7.0	6.4	6.8	6.4	348.8	446.2
		2048	8.8	13.6	8.8	8.8	9.7	8.8	390.8	554.0
35W 1.5GHz	45W 1.9GHz	512	6.6	7.4	6.6	6.0	6.0	6.0	345.4	388.7
		1024	6.6	7.8	6.7	6.3	6.3	6.3	342.7	389.0
		2048	8.8	11.0	8.8	8.8	8.8	8.8	391.8	468.7

Table 4.3: Execution times and energy consumption of two simultaneous Cholesky factorizations with different power caps assigned. For each experiment, both factorizations are configured with the same block size, and 10 workers each.

Table 4.3 shows the results obtained for BAR and BF-FREQ, together with the results obtained for an hypothetical optimal approach done through RAPL (OPT. on the table). To calculate the optimal measurements, two applications were run simultaneously using RAPL-ACT, with double the assigned power cap. Assuming a proportional distribution of the power cap is done, each application is executed with half of the power cap assigned. However, note that RAPL cannot provide different power caps to each application, being only possible to set a global power cap for the whole system.

Similar as before, experimental results reveal how BAR outperforms the baseline implementation (BF-FREQ) both in terms of execution time and energy consumption. In addition, results show how BAR is able to apply different power caps to each application, thanks to the use of a power model, and therefore, to apply different policies to each application. This approach, as described before, cannot be implemented by system-wide approaches (like RAPL), which base their decisions on global measurement, and tune the system (changing the frequency in this case) in a system-wide mode, instead of per-application or per-core, leveraging internal application-specific information.

Applying a different power cap to each application can be useful in scenarios where different requirements exist for each application, in terms of external constraints, or in terms of resource requirements. However, if no special requirements exist, a centralized approach considering the power cap as a dynamic resource, instead of performing a static distribution of it between applications, can increase the overall performance of the system. For example, consider now the same scenario for RAPL-PASS, but instead of having a different power cap for each application, consider an unique power cap gathering both (as the sum of each individual power cap). Similar to the previous scenario, this definition also guarantees that the maximum power cap is not violated at any moment of the execution. However, thanks to consider the power budget as a global resource, a proper and dynamic redistribution of the budget can be done between applications depending on their resource needs, increasing the overall performance. Indeed, Table 4.4 shows the results obtained when the power cap is considered globally in the previous scenarios, and executed using the RAPL-PASS approach. Although is not fully comparable, the total execution time is reduced respect to the one obtained by BAR. In general, considering the power limits globally can improve the overall

Pcap	b	RAPL-PASS		
		ExecTime (s)	Energy (J)	Max power (W)
73W (33 + 40)W	512	6.4	350.5	69.1
	1024	6.6	355.1	69.8
	2048	8.8	407.6	60.2
80W (35 + 45)W	512	6.0	346.2	77.6
	1024	6.3	344.4	77.2
	2048	8.8	403.9	60.0

Table 4.4: Optimal values for the same scenario considering the power cap globally, instead of assigning an individual power cap to each application.

performance of the applications, as well as the energy consumption. In the following sections we will extend BAR to consider a global power cap, performing a dynamic distribution of the budget between the applications based on the internal status of each runtime.

4.5. BACO. Runtime support for inter-application power budget redistribution

In scenarios where multiple applications run concurrently, a distribution of the power budget across applications is required to not exceed the global power cap. As applications do not have knowledge about others, granting a fixed amount of budget to each one ensures that the power limit is never jointly exceeded. In the previous sections we have shown how a static and asymmetric distribution of the budget between applications can be done to not exceed the power restrictions imposed. Internally, BAR manages the assigned budget so that it can be freely redistributed between workers of the runtime to achieve highly efficient executions.

However, there are situations in which the assigned budget is not fully used by all the workers within an application (for example, due to a very low number of active workers), or the distribution done a priori is not the perfect one and needs to be modified at runtime. In these situations, a considerable amount of power budget can be wasted. Nevertheless, a dynamic distribution of the power budget between applications can increase the overall performance of the system, redistributing the power budget not used by an application to those which can benefit more from it. In this scenario, a centralized resource manager, aware of the power consumption of each application, is required to properly redistribute the power budget between the different running applications. We target this problem next, and introduce BACO, a centralized resource manager able to perform this kind of power budget redistribution across registered applications.

Next, we propose a two-layer approach to address those scenarios where the power budget of the system is dynamically distributed between simultaneous applications to increase the overall performance of the system without exceeding the power cap. The first layer comprises a centralized resource manager (BACO) aware of the status of each running application; the second comprises all the different running applications, each one armed

with a BAR runtime. BACO is responsible of distributing the budget dynamically across applications, and each application is responsible of using the assigned budget optimally.

Decoupling the process of redistributing the budget to the process of using it, provides an effective method to run multiple applications with different power usage policies. BACO is based on a continuous and bidirectional communication between the resource manager and the applications, where applications inform the resource manager about the current power consumption and desired power budget, and the resource manager communicates to the applications the assigned power budget at each moment.

4.5.1. Resource manager layer

BACO (the centralized resource manager) is responsible for, based on the knowledge of each application at each moment, redistributing the unused power budget between those applications that can benefit most from it. In particular, our proposal is based on two main ideas: (I) *Applications cannot steal power budget from others without permission.* This implies that, if a specific power budget has been assigned to an application, it cannot be reassigned to other application. Contrary, under our deployment, applications will be responsible to *release and notify* to BACO the unused/excessive power budget at runtime, and (II) *Applications are not delayed.* This means that applications are executed as soon as they are launched. If there is not enough budget to run the application (even with one worker at the minimum frequency), it is possible to *steal* the assigned budget from other applications. This ensures that, independently from the overall performance, the individual execution time of each application is not extended.

To properly distribute the power budget between applications, BACO internally stores, for each registered application, a tuple containing the power budget assigned (*assigned*), the current power consumption of the application (*used*), and the power budget requested (*desired*), as well as the power budget not assigned to any application (*available*). The *assigned* and *available* power budget are directly controlled BACO, while the other two (*desired* and *used*) are continuously communicated by each application. Every time an event occurs in one application (e.g., the beginning or the end of the execution, the power consumption changes, etc.), the system will use the information stored to redistribute the power budget between applications.

Based on these parameters, applications are classified in three categories, namely:

1. *Demanding applications:* Applications that can benefit from receiving more budget, as they have workers not running at maximum frequency, or even not enough power to wake up idle workers. This situation arises when *desired*>*assigned*. Note that, as applications cannot use more budget that that assigned, these applications need to guarantee that *used*=*assigned*.

2. *Neutral applications:* Applications that are using all the assigned budget, but do not benefit from having more budget. These applications satisfy that *desired*=*assigned*, and *used*=*assigned*.

3. *Donor applications:* Applications that are using less budget than that assigned. In this case, the exceeding power budget can be redistributed between the demanding applications to increase the overall performance of the system. These applications are characterized by *used*<*assigned* and *used*=*desired*.

Every time an event occurs in one application, a communication with BACO is generated, sending a tuple {used, desired}, and a response from BACO is triggered including the assigned power budget. In order to determine the new power budget assigned to each application, the previous classification is used. If the application is classified as *neutral*, no changes are done in the budget. If the application is classified as a *donor*, the assigned budget is decreased to the actual power consumption, keeping the unused budget as *available*. In the case the application is classified as a demanding application, a new budget is assigned to it based on the *desired* budget, and the *available* budget in the system.

Algorithm 4.2 shows the pseudocode of the proposed policy.

Algorithm 4.2: Budget redistribution

```
1  begin
2      if assigned - used > δ then                                  // Donor app
3          availableBudget ← availableBudget + (assigned - used);
4          assigned ← used;
5          notifyNewBudget(clientId, used);
6      else if |desired-assigned| ≤ δ then                          // Neutral app
7          notifyNewBudget(clientId, used);
8      else if required - assigned > δ then                         // Demanding app
9          extraBudget ← min(available, required - assigned);
10         available ← available - extraBudget;
11         assigned ← assigned + extraBudget;
12         notifyNewBudget(clientId, assigned);
13     end
14 end
```

When an application is launched, an initial power budget is assigned by BACO based on the amount of budget requested initially by the application and the available budget unassigned to any application. If not enough available budget exists to run the application, a redistribution of the budget between running applications is carried out. In the latter situation, each registered application is requested to free a predefined amount of budget so that it can be assigned to the new application. The amount of budget released by each application is calculated supposing an equal distribution of the budget across applications. If the number of running applications is *Napps* (including the new application), the amount of budget released by each application is:

$$\frac{Pcap}{Napps \cdot (Napps - 1)} = \left(\frac{Pcap}{Napps - 1} - \frac{Pcap}{Napps} \right) \tag{4.3}$$

4.5.2. Application layer

The second layer is composed by all the applications running on the system, each with a specific power budget assigned ensuring the power cap is not exceeded globally. These applications are considered to be task-parallel, and are equipped with a Power-Budget-Aware runtime (e.g. BAR). However, how the power budget is used by each application is

not specified by the resource manager, letting each application to use it as desired. In other words, BACO is agnostic to the internal per-application power budget management policy. Additionally, in order to support a dynamic distribution of the power budget, applications have to constantly communicate with BACO to inform about the power usage and the desired budget, as well as to receive the power budget to use at each moment.

Satisfying the previous conditions, we propose the extension of the policy described in the previous section. On one side, we have demonstrated how it can achieve optimal executions never exceeding the dictated power budget. On the other side, targeting runtimes to deal with power consumption and communications with the centralized resource manager allows us to run any application without any modification.

Similar to the previous section, the communication with BACO can be integrated into the running loop of each worker, so it is considered as an extension of the BAR runtime. Every time a worker completes the execution of a task, it checks with BACO if a new budget has been assigned, modifying the frequency of the active workers, if needed, to the maximum frequency that guarantees not to exceed the newly assigned budget. Similarly, every time an idle worker is blocked or woken up, the worker sends to BACO the *used* and *desired* budget.

The *used* power is determined by the number of active workers running at the moment and the frequency set $used = \text{nIdle} \cdot \text{Power}_{\text{idle}} + \text{nActive} \cdot \text{Power}_{\text{currFreq}}$. The *desired* power is the prediction of the power used by the same number of active workers, but running at maximum frequency $desired = \text{nIdle} \cdot \text{Power}_{\text{idle}} + \text{nActive} \cdot \text{Power}_{\text{maxFreq}}$. Those values heavily depend on the number of active workers, making that, on those moments where the amount of active workers is low, the unused power can be redistributed to other applications.

4.6. Experimental results for BACO

We present next the results measured in two different realistic scenarios, and compare them against an optimal state-of-the-art approach. The first scenario comprises executions of two simultaneous Cholesky factorizations with different block sizes and different power caps. The second scenario extends it by executing the applications at different arrival times, overlapping their sequential and parallel phases. For the sake of completeness, we compare our approach with the RAPL state-of-the-art approach, which applies an homogeneous DVFS setup across all cores to never exceed the preset power cap. Finally, the third scenario simulates a realistic scenario running multiple simultaneous batches of different applications, each configured with different block sizes and spread over time following a normal distribution,

4.6.1. Preliminar analysis of BACO performance

Consider the execution trace represented in Figure 4.8, that reports the behaviour BACO when running two simultaneous Cholesky factorizations ($b = 512$ and $b = 1024$) under a 70 W power cap. The first two plots represent the number of active workers at each moment, and the power budget requested by the first application (equipped with a BAR runtime) and the budget assigned by BACO, respectively. The next two plots show the same information for the second application. The last plot shows the instantaneous power

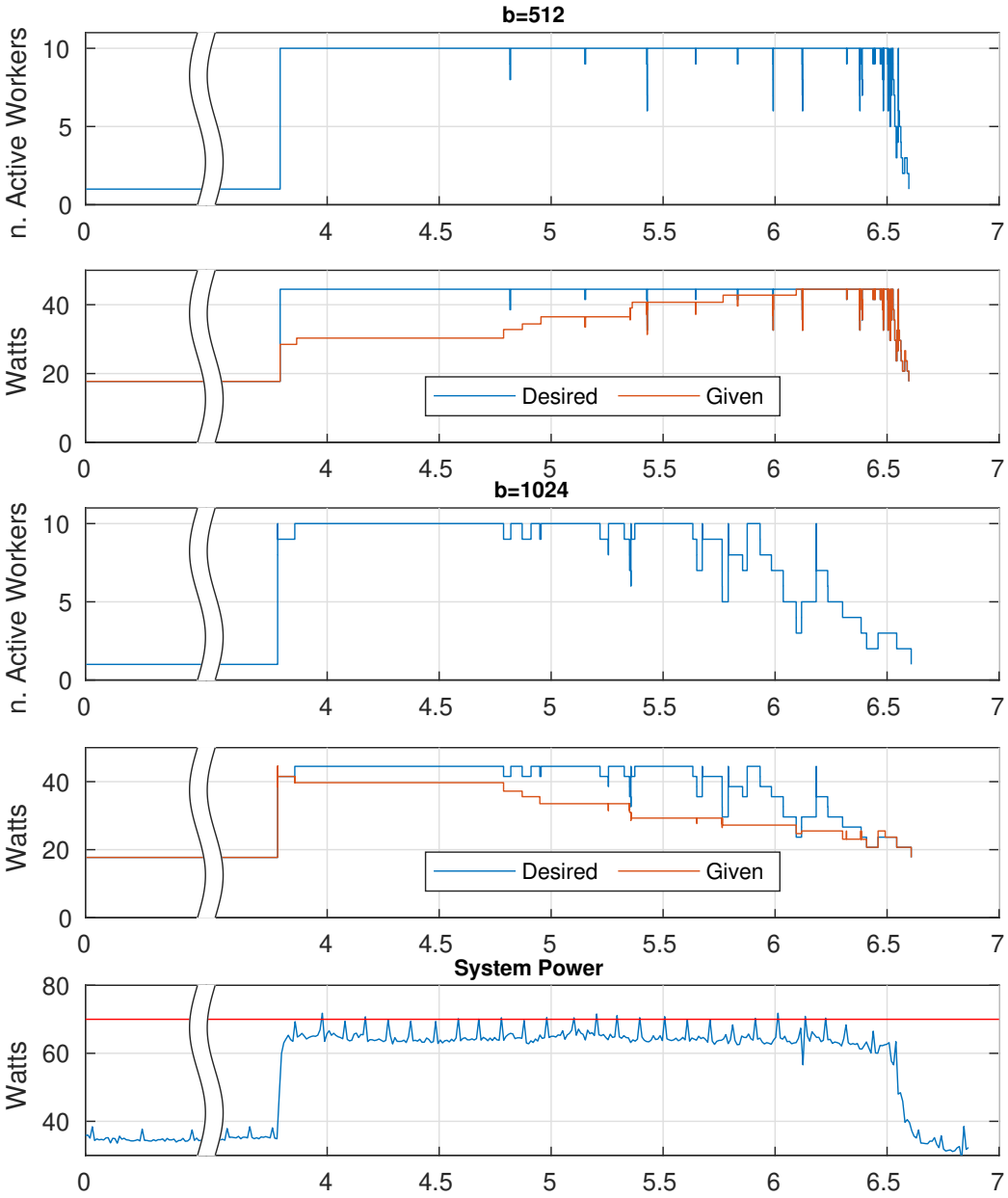


Figure 4.8: Detailed behaviour of BACO when running two simultaneous Cholesky factorizations ($b = 512$ and $b = 1024$) under a 70 W power cap. The first two plots represent, for the first application, the number of active threads and the power budget *desired* by the runtime and the budget *assigned* by the resource management. The next two plots show the behaviour of the second application. The plot on the bottom shows the power consumption of the whole processor measured through an external component (see Section A.2).

consumption of the whole processor, measured by an external library (i.e., real measurements are shown here).

Focusing on the reported instantaneous power consumption (bottom plot), we can observe how the BACO + BAR implementation is able to dynamically distribute the power budget between the applications, never exceeding the power cap. Also, observe how the distribution of the budget between applications entails the use of all the available power during the whole execution, meaning no power was assigned to an application and never used afterwards. The remaining four plots illustrate the actual behaviour of the applications and proposed system in terms of the number of active threads and the requested budget to BACO, and the budget assigned by it to each application. These traces show the general strategy followed by BACO: granting as much available budget as is requested to each application, redistributing only the unused budget by any application and not that already assigned.

Indeed, observe how during the sequential phase, the budget requested and assigned to each application is the minimum necessary to run one active worker at maximum frequency, keeping the rest of the budget unassigned to any application, and classified as *available* by BACO. As soon as the parallel phase starts, and the applications request more budget, the available budget is redistributed across applications. In this case, as the second application arrives first to this phase, the BACO grants all the requested budget to it. Contrary, as the available budget is lower than that requested by the first application (because it was assigned to the other application before), only the available budget is assigned, forcing the application not to run all the workers at the maximum frequency. As the number of active workers decreases in the second application (and therefore, the desired budget), the system dynamically redistributes the budget not used to the first application, allowing the runtime to increase the frequency of the active workers.

As a conclusion, this behaviour shows how BACO is able to dynamically (re)distribute the power budget based on the current status of the applications, never exceeding the preestablished power cap, and producing an optimal assignation of if (no unused power budget is observed).

4.6.2. Scenario I: Different block sizes

In this scenario, we explore the behaviour of BACO when running simultaneously two Cholesky factorizations of different block sizes under different power caps. Each factorization was run with ten workers, mapped to different physical cores (i.e., half of the available cores). The power caps tested ranges a wide of different values, from 90 W (a value high enough to run all the cores at maximum frequency during the whole execution) to 60 W (less than 50% of the Thermal Design Power (TDP)), and intermediate values of 80 W and 70 W ($\approx$ 65% and 56% of the TDP respectively).

Table 4.5 reports the results obtained by our approach (BACO + BAR) and RAPL, in terms of the execution time of each application (serial and parallel phases combined), the global energy consumption, the energy efficiency (in terms of GFLOPS/W), and the speed up of BACO + BAR w.r.t. RAPL. As both applications are run simultaneously, the total execution time is determined by the slowest factorization (i.e., the one with largest block size for this specific problem).

Pcap	b_1	b_2	Time app1		Time app2		Energy (J)		GFLOPS/W		SpeedUp
			PBACS+ BAR	RAPL	PBACS+ BAR	RAPL	PBACS+ BAR	RAPL	PBACS+ BAR	RAPL	
90W	512	1024	6.0	6.0	6.2	6.3	347.3	341.2	**0.99**	0.97	**1.01**
	512	2048	6.0	6.0	8.9	8.8	412.9	409.1	0.82	0.82	**1.00**
	1024	2048	6.3	6.3	8.9	8.8	409.4	402.9	**0.83**	0.82	**1.00**
80W	512	1024	6.1	6.0	6.6	6.3	350.3	346.3	0.97	0.97	0.96
	512	2048	6.0	6.0	8.9	8.8	418.1	406.4	**0.83**	0.81	0.99
	1024	2048	6.2	6.3	9.0	8.8	408.3	409.1	0.82	0.82	0.98
70W	512	1024	6.6	6.4	6.9	6.7	358.7	355.1	**0.79**	0.80	0.97
	512	2048	6.0	6.0	9.4	8.9	419.8	427.1	**0.94**	0.93	0.94
	1024	2048	6.2	6.3	9.5	8.9	412.7	427.3	**0.80**	0.81	0.94
60W	512	1024	6.9	7.4	7.7	7.7	384.5	384.0	0.88	0.88	**1.00**
	512	2048	6.1	6.6	10.9	9.4	431.1	478.8	**0.70**	0.78	**0.87**
	1024	2048	6.4	6.9	10.1	9.5	423.4	442.7	**0.76**	0.79	0.94

Table 4.5: Output metrics for BACO + BAR and RAPL when executing two simultaneous Cholesky factorizations with different block sizes, under different power caps.

In general, BACO + BAR achieves optimal results in terms of energy efficiency and performance, when the power cap is high, decreasing both as the power cap decreases. Nevertheless, a performance loss greater than 5% occurs only when the power cap is lower than 70 W (56% of the TDP), being a low enough value not to be used on a realistic scenario. Besides that, the average speed up achieved by our solution is 0.97 compared with an optimal solution, with a maximum loss of 13% in time. Similar as in the previous sections, when no power cap is applied (i.e., pcap$\geq$90 W), both approaches achieve the exact same values, showing no overhead is introduced by the BACO or BAR integration.

Although both approaches achieve high performance executions without exceeding the power cap set, observe how the behaviour of both approaches differs, as the execution time of each individual application varies between approaches. While BACO does not redistribute an assigned budget until the applications release it, RAPL redistributes the budget uniformly between all the cores, setting them to the same frequency. For this scenario, both approaches have demonstrated to be perfectly valid, achieving similar results in both cases. However, in scenarios where different policies need to be applied individually to each application (for example, they have different priorities), approaches like RAPL, that base their decisions on system-wide changes, cannot be utilized.

Targeting energy consumption, the experiments can be classified in two different sets: experiments where BACO has a slightly greater consumption than RAPL, and the contrary. The first set comprises the experiments with the highest power caps, and correspond to those scenarios with an speed up close to 1.0. On the contrary, when the energy consumption of BACO is lower than the consumption for RAPL, performance also diminishes. This correlation between energy consumption and speed up, and its relation with the different power caps, lead us to conclude that our energy model does not estimate perfectly the power consumption of the lowest frequencies, and a better energy model would produce executions closer to the optimal one. In addition, the energy efficiency follows the same behaviour as

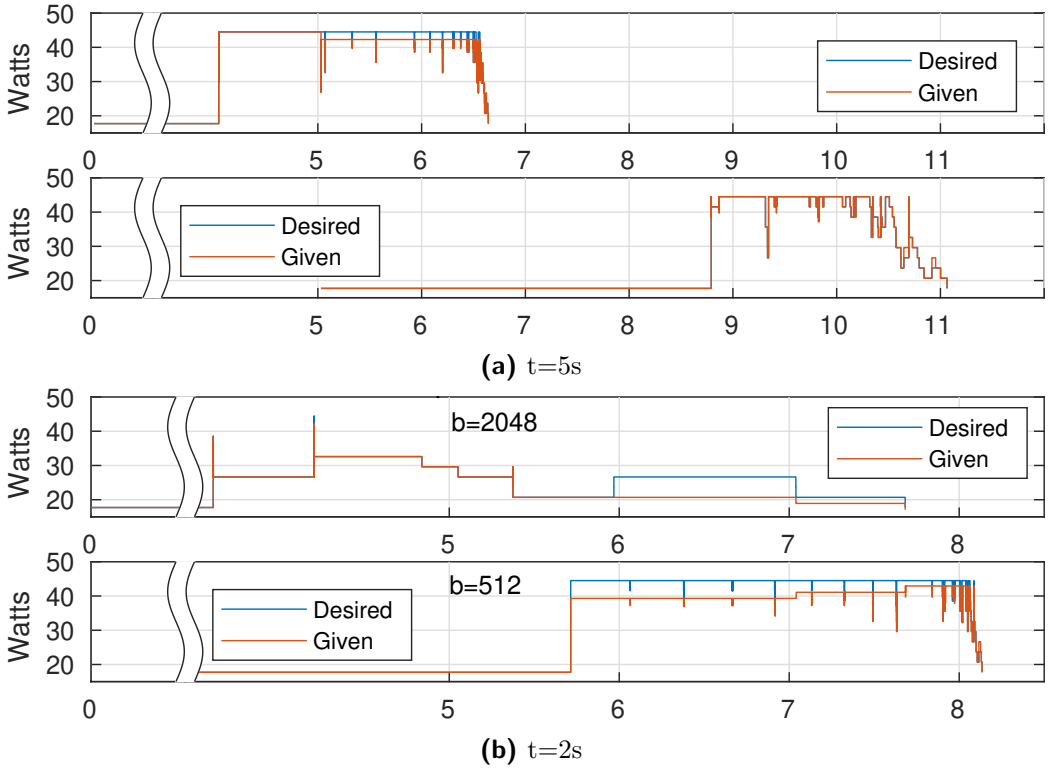


Figure 4.9: Assigned and desired budget of two different Cholesky factorizations when executed concurrently with a 5 (top) and 2 (bottom) seconds delay between them and a power cap of 60 W. Note that for the first application, the *desired* budget is almost identical to the *given* budget as explained in the text.

the speed up and energy consumption. For the highest power caps, our approach is able to achieve the same energy efficiency as RAPL, even greater values in some experiments, increasing the difference when the power cap decreases.

4.6.3. Scenario II: Different application arrival rates

This scenario explores a more realistic situation in which applications do not start simultaneously, but simulates an environment where multiple users can launch them at different time points. Contrary to the previous scenarios where there was always enough budget available to launch both applications (as serial phases matched in time), it can be the case for BACO to perform a redistribution of the budget, removing some assigned budget to one application to assign it to a new applications waiting to run. For example, if the first application is in the parallel phase using all the available budget, and a new application is launched, a redistribution of the budget between applications has to be performed, as explained before (see Equation 4.3).

In particular, in this scenario we explore two different points of times at which applications can start, that cover all the possible cases: $t = 2s$ and $t = 5s$. For $t = 2$, it ensures that the second application starts while the first one is still on the sequential phase, and therefore, there is enough available budget to start the second one. Starting the second ap-

plication at $t = 5$, guarantees that the second application starts when the first factorization is on its parallel phase, so that it is possible for the resource manager not to have enough available budget, and needing to *steal* some budget from the running application to assign it to the second one and start its execution. Figure 4.9 shows the *desired* and *assigned* budget for an scenario where two different factorizations ($b = 512$ and $b = 1024$) are executed concurrently, with a 5 and 2 seconds gap between them (top and bottom respectively) and a power cap of 60 W. In this scenario, all the power budget is assigned to the first application as it is started before. As explained before, because there is not *available* power budget for the second applications, the required budget to run the second application is removed from the first one, assigning it to the second, and therefore, avoiding to delay the execution of the second.

Similar as for the previous scenario, we have explored all the different combinations of block sizes, and the same power cap values. For all cases, the total execution time of our approach is equal to the one obtained by RAPL (maximum difference between both approaches is less than 1%). The same situation happens for the energy efficiency or the total energy consumption.

On the contrary, as explained before, the execution time of each individual application differs between approaches. While RAPL does a homogeneous distribution of the budget and frequency between applications, BACO does not redistribute it until it is not going to be used any more by the application. From the results, we can conclude that both approaches are perfectly valid, and useful in all the scenarios. Table 4.6 shows the results of different experiments in terms of execution time of each application, and the speed up between both approaches (Δ).

4.6.4. Scenario III: A realistic simulation

For the sake of completeness, we have run our system on a realistic scenario, where different batches of Cholesky factorizations were run simultaneously, simulating a real system where multiple applications are run concurrently, and the workload varies over time. On our experiments, each batch was composed by 100 different Cholesky factorizations spread over time. The delay between the end of a factorization and the beginning of the next one was randomly chosen following a normal distribution. Specifically, we have tested our approach running 2 and 4 simultaneous batches, assigning 10 and 5 physical cores to each batch respectively. To increase the variability of the system, each batch was configured to run with a different block size, using $b = 512$ and $b = 1024$ when only 2 batches were run simultaneously, and $2 \times b = 512$ and $2 \times b = 1024$ when 4 batches were executed. Similar to the other experiments of the chapter, the matrix size was configured as $m \approx 10000$. For each configuration, four different power caps where tested.

For the shake of clearness, consider first the experiment where only two application batches were run simultaneously. Going in depth in the behaviour of the system, we can clearly differentiate three different states the system can be:

ⓐ Moments of the execution where two applications are executed simultaneously and both expose enough parallelism (in form of ready tasks) to utilize all the available resources.

Pcap	b_1	b_2	$t = 2s$					$t = 5s$				
			PBACS+ BAR		RAPL		Δ	PBACS+ BAR		RAPL		Δ
			t1	t2	t1	t2		t1	t2	t1	t2	
90W	512	512	5.97	6.18	6.10	6.22	1.01	5.99	5.99	5.98	5.99	1.00
	2048	512	8.84	6.17	8.94	6.20	1.01	8.82	6.08	8.82	6.13	1.01
	2048	1024	8.83	6.38	8.89	6.45	1.01	8.83	6.32	8.83	6.32	1.00
80W	512	1024	5.98	6.42	6.05	6.47	1.01	5.98	6.22	5.98	6.22	1.00
	512	2048	5.98	9.00	6.03	9.04	1.01	5.98	8.84	5.98	8.83	1.00
	1024	1024	6.20	6.43	6.33	6.49	1.01	6.22	6.24	6.24	6.25	1.00
70W	512	512	5.97	6.17	6.06	6.22	1.01	5.98	5.99	5.99	5.99	1.00
	512	2048	5.98	9.02	5.98	9.04	1.00	5.98	8.82	5.96	8.83	1.00
	1024	1024	6.22	6.43	6.26	6.48	1.01	6.19	6.24	6.22	6.22	1.00
60W	1024	512	6.29	6.25	6.43	6.31	1.01	6.29	5.98	6.25	5.98	1.00
	2048	512	9.37	6.40	9.13	6.47	0.98	8.81	6.07	8.83	6.11	1.01
	2048	1024	9.24	6.59	9.23	6.82	1.00	8.82	6.30	8.82	6.31	1.00

Table 4.6: Output metrics for BACO + BAR and RAPL when executing two simultaneous Cholesky factorizations with different block sizes and different start points (2 and 5 seconds), under different power caps.

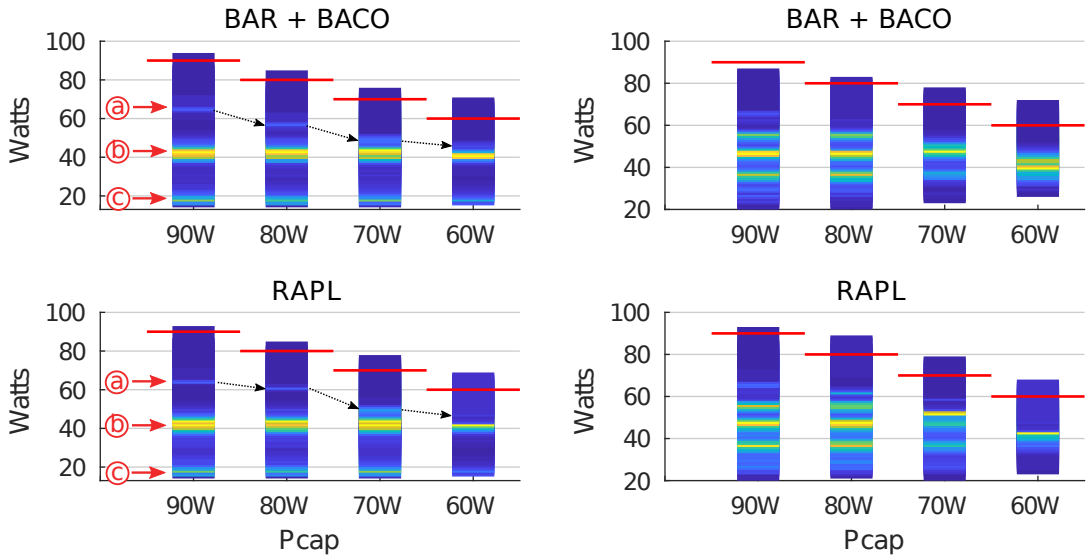
ⓑ Moments of the execution where two applications are running simultaneously, but not all the resources are used. For example, this is the case where one application is in the serial phase (as only one worker is active), or both applications are in the parallel phase, but there is not enough parallelism to use all the resources. Clearly, this is the most frequent state.

ⓒ Moments of the execution where no application is run in any of the batches, being the whole system in idle state.

Those states are clearly represented on the left plot of Figure 4.10, showing the histograms of the instantaneous power consumption measured during the experiments for 2 (left) and 4 batches (right). Each bar represents a different power cap tested (indicated by the horizontal red line and the labels on the x-axis). Lightest colors represent the most common power consumption measured over time (in a scale from yellow to dark blue). Darkest points correspond to punctual measurements related with the noisy nature of the date and can be ignored.

As observed, in each experiment three different power levels can be distinguished as the most frequent ones (those marked as *a*, *b* and *c*) corresponding each to the previously states mentioned. As expected, the plots confirm the state *b* as the most common one.

Focusing on the evolution of the three regions when the power cap decreases we can determine how our approaches affects the behaviour of the system. While states *b* and *c* are not affected in any situation as the power consumption of these states are far from the maximum one, our system needs to limit the power consumption of the system when



(a) 2 lines **(b)** 4 lines

Figure 4.10: Power consumption histograms for different power caps and approaches. A lighter color means a most common measurement of that power. Darkest blue points correspond to noisy measurements and can be ignored.

it is in state *a*, i.e., when all the resources are used. The plots also shows how the RAPL approach has a similar behaviour in all the cases. When 4 different batches are run, a greater number of states appears as the number of combinations between phases of the applications of different batches can occur, resulting in a greater number of phases visible in the traces shown in the left of Figure 4.10. Nevertheless, the overall behaviour of the approaches is similar to the one described before.

Table 4.7 shows the output metrics of all the experiments in terms of total execution time (in seconds) and average power consumption (in watts). Similar to the other experiments in the chapter, the results confirm that our approach achieves optimal results, obtaining the same results as RAPL when launching 2 simultaneous batches, and slightly better results when 4 are launched. Results for 4 simultaneous batches run under 60 W are not shown as our approach is not able to run all the applications under the power cap the whole time, violating the power cap the 4.69 % of the time. Nevertheless, a power cap of 60 W is less than the 50 % of the TDP, being a low enough limit to not be used in any real system.

4.7. Conclusions

In this Chapter we have explored how traditional resource managers can be extended to achieve high performance executions in modern platforms where the instantaneous power consumption is strictly limited. Our approach is based on a light and effective process to detect idle workers, a passive blocking mechanism with no overhead, and a proper redistribution of the saved power between the active workers in form of an increase in the frequency. This redistribution of the power budget and the calculation of the frequency to set is sup-

	Time (s)		Power avg.	
	PBACS+ BAR	RAPL	PBACS+ BAR	RAPL
90 W	717.8	720.8	53.3	52.8
80 W	726.4	721.8	52.7	52.8
70 W	738.2	732.0	52.3	52.3
60 W	755.0	764.6	51.3	51.1
2 simultaneous batches				

	Time (s)		Power avg.	
	PBACS+ BAR	RAPL	PBACS+ BAR	RAPL
90 W	755.0	820.2	59.1	59.3
80 W	821.5	821.7	58.6	59.3
70 W	829.9	841.8	59.1	58.0
4 simultaneous batches				

Table 4.7: Output metrics when running 2 and 4 simultaneous application batches

ported by a conservative power model created from a simple profiling, but proved to be perfectly valid for the tested scenario. In addition, the profiling of the different tasks makes the model perfectly valid for other DLA operations. Nevertheless, the proposed strategy do not actually depend on the model used, being valid for other power models.

The proposed approach has been tested on a real platform, outperforming the default policy implemented in NANOS++, obtaining a speed-up up to 1.9 (average value: 1.3), and a reduction in energy consumption of 46% in average. In addition, our approach achieves the optimal results in performance and energy consumption, similar to RAPL, which bases its decisions on a real-time power measurements, and a system-wide policy. Additionally, we have shown how our approach is able to do an asymmetric distribution of the budget between applications, contrary to RAPL, which has a proportional distribution of the frequency between all the cores.

Considering a more realistic scenario where not all the applications are similar, or they are not executed simultaneously, we described how to incorporate a centralized resource manager to the previous ideas to perform a dynamic distribution of the power budget between the running applications, assigning the budget not used by one application to those applications which can benefit from it. This approach have been proved to be perfectly functional, achieving the same results than the ones produced by RAPL, both in terms of execution time and energy consumption.

Finally, in the last two Chapters we have shown how the resource management process can benefit from having knowledge of the status of the runtime to improve different metrics (execution time, energy consumption, energy efficiency, power capping, etc.), and how this can be useful for scenarios where multiple applications are running, and how a centralized resource manager can be useful to orchestrate the different applications. In the next chapters we extend these ideas to consider not only metrics of the runtime, but also specific metrics of the running applications, offering a plethora of new opportunities to optimize. However, this new opportunities makes the decision space hardly manageable by traditional approaches, needing to incorporate machine learning techniques to handle it.

Part II

Application-aware Resource Management. A Machine-Learning based approach

5

Resource Management for QoS-aware applications

In Chapters 3 and 4 we have demonstrated how resource managers can include the knowledge of the internal status of the running applications in general, and of the status of the runtime task scheduler supporting the execution of the applications in particular. This approach makes it possible to extend and complement the knowledge of the environment and system status, as traditional resource managers usually do. In particular, this extended knowledge allows resource managers not only to exhibit a more precise control of the environment and to achieve a better exploitation of all the resources exposed by the underlying architecture, but also to optimize the execution based on different metrics offered by runtime systems (e.g. progress, number of ready tasks, number of idle worker threads or task-execution-time estimation, among others). In addition, this knowledge can also be leveraged to perform better decisions on traditional objectives like energy consumption, execution time or the efficient use of the available resources, among others.

However, although resource managers can potentially exploit information from multiple sources to optimize both the application execution phases and to improve resource usage, there is still a considerable number of optimization opportunities that are not usually considered, mainly due to the absence of a deeper and more specific knowledge of each application and its internal status.

In this chapter, we propose extending resource managers to consider specific knowledge of the running applications both in terms of metrics to optimize, and application knobs that can be potentially exposed and dynamically tuned. Obviously, the exposition of these new available metrics and knobs naturally entails a huge decision space that can be hardly managed by traditional heuristic-based approaches. The proposal in this chapters orbits around the integration of ML approaches (specifically a RL approach) in order to tackle the increasing complexity in the decision process for resource management. Figure 5.1 shows an overview of the systems, describing the relation between platform, applications and resource manager.

The goal of the chapter is two-fold. First, to provide a general overview of a prototypical QoS-aware application that exposes dynamically tunable knobs and pursues specific output

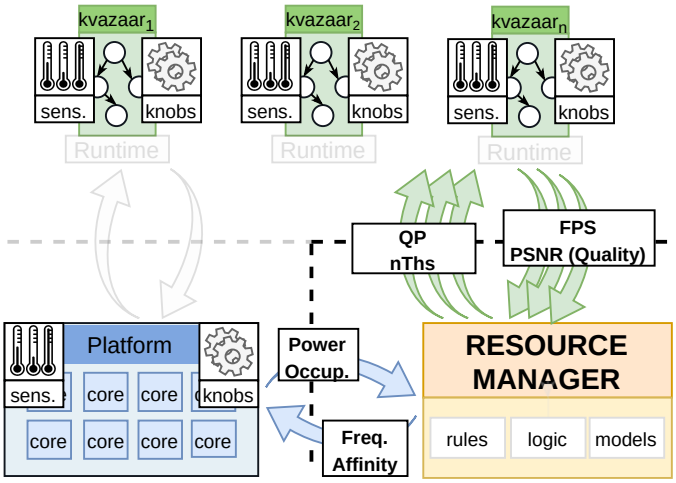


Figure 5.1: General overview of the resource manager formulation proposed in this and the following chapters.

metric values, mainly related with QoS fulfilling. Second, motivating the use of Machine Learning approaches in order to alleviate the massive decision space generated from malleable applications. As a motivating example, we will employ from now on a motivational use-case in terms of a QoS-aware highly malleable application: multi-user video transcoding. This chapter develops a discussion regarding the impact of a proper selection of values for the exposed knobs for the aforementioned application that will be useful for the development of Chapters 6, 7 and 8. Specifically, in this chapter:

1. We propose an extended design for resource managers that considers not only information from the system, but also metrics obtained from the running applications.

2. We consider malleable applications that expose a number of internal knobs that can be dynamically tuned, each with a direct impact on a specific system- or application-wide metric.

3. We demonstrate how this new scenario offers a plethora of new knob combinations that can be hardly managed by traditional approaches, and describe how a ML approach can reduce the complexity by means of unsupervised learning. This approach is able to find autonomously the relations between metrics and configuration parameters.

4. We describe the internals of Q-Learning (QL), and how a generic resource manager can be formulated to incorporate the Q-Learning approach in its decision process for generic QoS-driven applications.

Section 5.1 extends the definition of *malleable applications*, showing how an external resource manager can handle different application knobs based on specific application metrics. We will refer to this approach as QoS-aware resource management. In addition, we discuss how, in co-scheduling scenarios, applications can benefit from delegating the tuning of its internal knobs to a centralized resource manager. Section 5.2 provides deep insights of an example of a malleable and QoS-aware application: HEVC online video transcoding, that

will be used through the development of this and the following chapters; the section also performs an analysis of the internal and external parameters (*knobs*) that affect its execution, and how they are related with different internal and external metrics. It also presents a detailed motivation for dynamic resource and knob management, arguing why traditional approaches cannot efficiently handle this scenario. Section 5.3 describes the mathematical model of a Markov Decision Problem (MDP), and how a resource manager can be formulated in terms of it. Also, it describes how a Q-Learning algorithm works and how it is able to find an optimal policy to solve a MDP problem based on a dynamic programming approach. Finally, Section 5.4 closes the chapter with the main contributions and motivation for the following chapters.

5.1. Exposing application internals: metrics & knobs

The increase in the number of cores per processor has been a trend of paramount importance in the evolution of modern processing architectures, becoming the current tendency (together with heterogeneity) to increase performance in current processors [83]. This rise in the number of cores allows both the exploitation of *intra-application* parallelism, and also the execution of multiple concurrent applications in the same node, exploiting *inter-application* parallelism and hence increasing the overall system throughput. In order to boost performance, traditional resource managers usually distribute the available cores *statically* across applications, by means of sophisticated heuristics based, for example, on a priori profiling, trying to avoid the co-existence of multiple application instances competing for the same resources, and hence reducing contention [22].

As we saw in Chapter 4, the behavior of an application is far from being uniform in resource requirements during its life cycle; even more, even two consecutive instances of the same application could differ in behavior depending on their input data. Anyway, in phases where an application does not utilize all the (pre-)assigned resources, those can be shared with or granted to other co-existing applications, increasing the performance of the latter without affecting the execution of the former. A clear example arises when an application (or a thread within) is blocked in Input/Output (I/O) operations [155], communication points [70], or due to a coupled-application behaviour [204, 203] (typically producer-consumer relationships between workloads) among others, and has been widely studied in the past. This behavior also appears on parallel applications with variable workloads, and hence needing different amount of resources during their execution, and therefore, benefiting from a *dynamic* resource distribution. This scenario has been studied in Chapter 4, showing how a centralized resource manager aware of the internal status of the task scheduler executing each application can handle this scenario.

In the following, we will use the term *QoS-aware applications* to describe a class of applications that do not necessarily need to achieve maximum performance, but to fulfill a minimum level (quality) in a given metric or set of metrics during the whole execution. We will refer to these application specific metrics as *internal metrics* (e.g., quality), and to those more traditional system-wide metrics as *external metrics* (e.g., power consumption). These applications, by their nature, do not need to use all the available resources (even if they exhibit enough workload to fully use them). For example, video streaming seeds to provide 24 FPS and not more.

These resource requirements vary between applications, and range from real-time requirements in terms of execution time, to energy consumption limits (power capping) or maximum response time, among others. These applications are of special interest nowadays in cloud environments, where multiple application instances (equal or different between them) are simultaneously executed in the same computing node [136]. In this scenario, the simultaneous quest of an optimal use of resources and a minimum quality fulfilling in terms of Service Level Agreements (SLAs) has to be provided by the cloud provider. Usually, the actual use of resources and the value of QoS metrics is highly sensitive, and can be modified by tuning application specific parameters (called *internal knobs*), or system-wide parameters (called *external knobs*). Each knob can affect one or multiple metrics, and each metric can be affected by one or multiple knobs. Determining the best value for those knobs at each execution point is, obviously, far from being a trivial task.

A naïve yet effective way to handle this situation consists of a static distribution of all resources across applications on a pre-execution fashion, together with a proper configuration of all the application-level knobs in accordance with the resource (pre-)assignment. Once this distribution is completed, each application will be in charge of utilizing its assigned resources accordingly to satisfy the established QoS requirements. However, although this approach is perfectly valid, it presents two major drawbacks, namely:

(a) *Resource imbalance.* Typically, different applications require different amount of resources. If the resource distribution is not properly performed, some applications will waste assigned resources, while others will not achieve the required performance due to the absence of them.

(b) *Resource rigidity.* As resources are distributed a priori, and the knobs are statically configured, they need to be set in a conservative fashion, so that the QoS requirements are fulfilled at the most demanding stage of the application lifetime. If the amount of work varies, this can imply that stages where the assigned resources will not be used can arise, wasting resources that could be assigned to other applications.

On the contrary, as we described in Chapter 4, if a *dynamic* resource distribution is in place (i.e., not determined a priori, but elastic enough to be varied during execution, under request), the aforementioned problems can be alleviated. Assigning resources upon application request does not only avoids the problem of wasting assigned resources, but also allows the redistribution of resource to other applications that can potentially need them.

However, in order to support dynamic resource distribution, traditional resource managers have to be extended to be aware of the internal status of all the running applications and their internal metrics. This extension does not only affect resource managers, but also applications that need to inform about the current values of the observed metrics. In addition, if some of those metrics are affected only by internal knobs, applications should also expose those knobs to the resource managers, so they will be able to externally modify them together with other system knobs (or *external knobs*).

The class of applications that can modify their internal knobs dynamically during their lifetime are usually named *malleable applications* in the literature. Strictly, the term malleability usually restricts the dynamic nature of applications to only one knob: the number

of active threads in the application [117]. Thread malleability, also referred as elasticity in the cloud computing arena, has been of wide appeal in the last years. It has been deeply studied mainly in terms of dynamic thread variation in the fields of High Performance Computing (HPC) [53] and multi-cluster systems and cloud computing [31, 33]. Thread malleability however, can be extended and generalized to other *dynamic application knobs*. In the following we extend the definition of malleable application to not consider only the applications that are able to tune themselves, but also to those that expose their internal knobs and metrics to be tuned by an external resource manager. The ability of a resource manager to access to internal application metrics as well as to knobs that directly affect these metrics allows us to: (I) tune the internal and external knobs by the same resource manager, being able to discover and learn the relations between knobs and metrics, and (II) execute multiple malleable applications at the same time, being able to perform a proper resource distribution and knobs value selection to avoid situations with multiple applications competing for the same resources and or wasting resources.

However, the addition of all these new metrics and knobs to the resource managers, together with the more traditional knob and optimization goals, result in a considerable amount of knobs to tune and a plethora of scenarios to optimize; its implementation and efficiency is, hence, a daunting task. Fortunately, Artificial Intelligence (AI) techniques can be of great appeal to face this complexity burden. In the rest of the chapter we discuss how a centralized resource manager can be formulated to incorporate AI techniques to tackle all these problems, dealing with QoS-oriented malleable applications.

5.2. A motivational example: multi-user video transcoding

As explained in Section 1.2, online video streaming services account for 57% of the global Internet share, and it is supposed to surpass the 80% downstream threshold by the end of 2020 [160]. The increase in the visualization of media contents via streaming, specially the so called Video On Demand (VOD), has entailed the emergence of new problems for video providers. As a result of the increasing diversity in video formats, users' devices (and thus, media resolution), and available bandwidth, the original media has to be adapted to the different users via a transcoding process, typically a high resource-intensive process [191, 116].

The typical solution to alleviate this two-stage process used nowadays, consists of storing multiple versions of the same video in different formats, serving the most suitable to users on demand. However, given the amount of new video content daily uploaded to data centers, implies that ad-hoc storage is a costly and inefficient solution. A transcoding process is a two-phase process, in which the original video is decoded to an intermediate format to be encoded again with the requested features. In addition, in multi-user scenarios, video providers' servers receive multiple simultaneous transcoding requests, each with different quality or throughput requirements. A promising alternative is real-time video transcoding, which re-encodes the original video on the fly depending on user's requests.

High Efficiency Video Coding (HEVC) has emerged as a feasible solution to alleviate the exponential growth in network traffic originated both for live streaming and Video

On Demand, because it provides up to 50 % better compression compared with their predecessors keeping video quality [144]. This dramatic reduction in bandwidth, however, entails a proportional increase in computational complexity, shifting the wall from network performance to video providers' servers in terms of both *performance* and *energy consumption*. Regarding *performance*, the efficiency with which servers can handle the increase and variety in user requests, dynamically managing internal resources to meet quality requirements without unnecessarily wasting computing resources, is key to maintain a proper user QoE [142] under a reasonable resource usage. Regarding *energy consumption*, meeting restrictive per-server power capping thresholds to maintain data center energy consumption under control is also of capital importance for providers [192]. While mechanisms exists that force tight power capping thresholds (e.g. Intel RAPL on Intel Xeon servers) [138], the use of application-aware power capping mechanisms can greatly improve the *efficiency* –performance delivered under the cap– and *timeliness* –response time after a new cap is enforced– [202], as we show in the previous chapter.

Many modern video transcoding standards, including HEVC, expose a number of dynamically tunable knobs with different implications in resource usage and attainable Quality of Service, converting these applications into *QoS-aware* and *malleable* applications. Similarly, modern techniques exposed by hardware also have considerable impact on application throughput and power consumption. Hence, a proper selection of application-side and system-level parameters to simultaneously fulfill QoS requirements of clients while optimizing resource usage becomes a challenging task to increase the productivity of the underlying computing platforms. Focusing on HEVC as the de-facto standard for video encoding, the bottleneck for achieving real-time transcoding is the encoder complexity, which is approximately 100× higher than that of the decoder [29]. Moreover, the numerous parameters available for adjusting the output quality and throughput add extra complexity. Finally, dealing with multi-user environments, where multiple different encoding requests have to be fulfilled simultaneously, and dynamic video contents poses other challenges on video providers' servers. Thus, we exclusively focus on resource management to HEVC encoder in order to optimize performance and energy of the servers while providing the required per-user QoS and QoE. However, the development of a holistic, autonomous resource management scheme to simultaneously adapt application- and system-wide knobs with real-time requirements is far from being a trivial task, as shown next.

All in all, multi-user video transcoding is a real application of wide appeal to develop and integrate our ML-based resource-management proposals, mainly due to the following characteristics:

Real-life application: Multi-user video transcoding is not a synthetic benchmark, but a real-life application widely used nowadays worldwide, and with a good perspective to be further implemented in the future.

QoS awareness: Tight limits in application-level metrics –throughput, quality or compression ratio (see Section 5.2.1)– require a careful selection of application-level knobs, and hence derive complex resource management schemes.

Dynamic application-level knobs: Modern video transcoders usually include a number of dynamically modifiable parameters (knobs) that can be selected and modified autonomously or externally by resource managers to adapt QoS with direct implications on application metrics and system-wide resource usage.

Dynamic system-level knobs: Modern servers in which video transcoding processes are usually deployed also exhibit a number of architectural knobs that can be modified at runtime to adapt or limit application- and system-wide metric limits (e.g. throughput or power consumption, respectively); per-core Dynamic Frequency-Voltage Scaling (DVFS) or power capping mechanisms are just two examples of such system-level knobs.

Restrictions at system-level metrics: as for application-level metric limits (to fulfill QoS), the imposition of system-wide limits shared by all applications is usually in place and imposes an additional difficulty in resource management.

5.2.1. Output metrics, Application- and System-wide knobs, and QoS in HEVC

A generic HEVC encoder (and typically any modern encoder) is built on top of different processing blocks, each of which has multiple input tunable parameters or *knobs*, and is characterized by different metrics related to both the output file and the performance obtained during the encoding process. A modification on any of those parameters affects the behavior of the encoder in terms of one or multiple output metrics, and features system-wide collateral effects (e.g., chip power consumption [97] or socket occupation, among others).

Application-level metrics:

Among all the set of different metrics available, those that best represent the behaviour of the encoding process, and therefore, the most studied in the literature are:

Frames per second (FPS). Measures the *throughput* of the encoding process, and represents the overall performance of the system. An encoding process is considered real-time if the process is able to encode above 24 FPS (defined in the NTSC standard [129]), or 60 FPS (in the latest standards) [85]. The throughput of the process is highly affected by the overall performance of the physical system, but also by the desired quality of the output sequence and the contents of sequence being encoded (texture and motion).

Peak signal-to-noise ratio (PSNR). Relation between the original data and the noise introduced in the encoding process, measures the *quality* of reconstruction. A value between 30 and 50 dB for lossy compression codecs (as HEVC) represents an acceptable quality for the human vision, being the greater the best [194, 78]. Peak Signal-to-Noise Ratio (PSNR) is measured on a frame-to-frame basis.

Bitrate. Measured in *Mbits/s* determines the *compression level* of the output sequence. For streaming purposes, this value should be lower than the bandwidth offered by the standard technologies used nowadays (3G bandwidth ranges from 2 Mbits/s to 6 Mbits/s,

while 4G technology offers a bandwidth from 7 Mbits/s to 17 Mbits/s) [21].

Video format and, most importantly, specific video contents play a key role in the attained *throughput*, *video quality* and *compression*, and also in the instantaneous power consumption. Since video contents may differ on a frame-to-frame basis without any predefined pattern, encoding parameters should be regularly adapted (ideally, frame by frame) for Quality of Service (QoS) optimization under a limited power budget.

System-level metrics:

Systems offer a plethora of different metrics to consider and optimize, some of them related with the running applications (e.g., number of cores in use limits the parallelism of the applications), and some of them independent (e.g., temperature). The most important metrics, broadly studied in the literature are:

Power consumption. Of special interest in embedded systems [42] and cloud environments [4]. Power consumption is directly related with the resource usage and the frequency processors are running. Modern hardware exposes power measurements to the running software [13] allowing resource managers to tune it via software-defined strategies (e.g. *powersave* governor), or hardware-based policies (e.g., RAPL [50]) as described in the previous chapter.

Core occupation. This metric represents the number of cores in used at each moment. This metric can be considered as an optimization goal (to minimize the resource usage), or can be used by resource managers to take better decisions as other metrics and knobs are directly related with it (for example, power consumption is highly influenced by the number of active cores, or the effective frequency when *turbo mode* is enabled is also determined by this metric).

Temperature. Multiple works, like [49], [96] or [97], consider this metric as an optimization objective, trying to achieve maximum performance without exceeding a thermal limit. Although this metric is not considered in the following, the ideas presented in this thesis are generic enough to incorporate temperature to the problem formulation without additional efforts.

Application-level knobs:

Since its introduction in 2012, several implementations of the HEVC encoding standard have been proposed, from the non-real-time HM Test Model [27] as the reference software, to Kvazaar [186] and x265 [8], both of which are able to provide real-time HEVC encoding through parallel processing and architectural-aware adaptations at different levels. In this thesis, we consider the Kvazaar open source encoder as the baseline of our investigation, as it achieves faster multi-threaded transcoder compared to x265 [109] and it is a clear example of a highly optimized and dynamically tunable HEVC encoder. Each basic building block in Kvazaar (similarly to other HEVC encoders) is parameterized by means of input *knobs* with application-wide impact (e.g., throughput, encoding quality or compression among others)

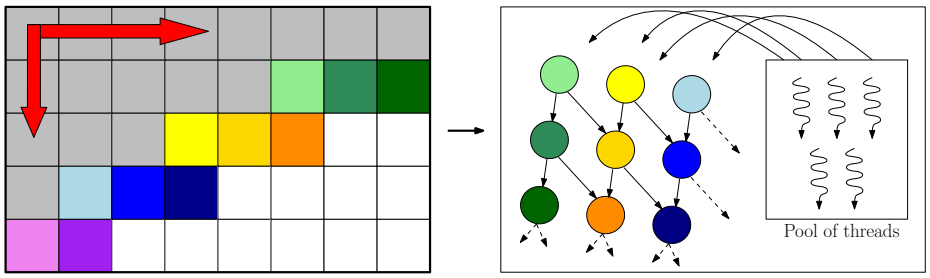


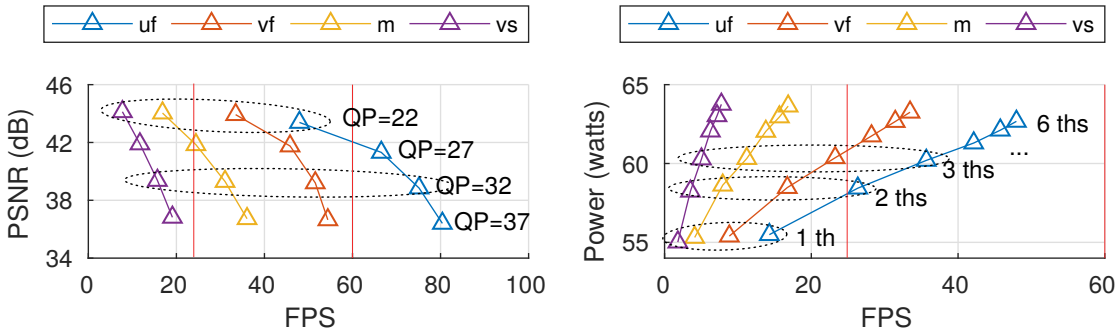
Figure 5.2: Wavefront parallel processing order, and its associated task dependency graph.

and/or system-wide effects (e.g., power consumption). Some of them must be decided and statically fixed a priori with no option for runtime modification (for example the *preset* selection); others can be modified, with different granularity, at runtime (e.g., Quantization Parameter (QP), or Number of Threads used to codify each frame[1]). We will refer to them as *static* and *dynamic knobs*, respectively, following the idea proposed in [87]. Moreover, not all dynamic knobs have the same impact on the encoding process, so that it is not necessary or feasible to tune all the available ones, but only those with a highest impact on the output Frames Per Second (FPS), quality, bitrate and/or power consumption. After an exploration of all the different knobs, and the impact of each of them on the considered metrics, we have limited the scope of our proposal to those with the largest impact on the output quality and performance. However, our proposal can be mapped to other knobs with minimal changes.

Quantization Parameter (QP). Dynamic knob in charge of the quantization degree per frame [169]. QP plays a significant role in output quality and can be tuned on a frame-to-frame basis at runtime [97, 95, 137]. Traditional approaches, as Huang et al. [89], Biatek et al. [23], or Czuni et al. [48] among others, have worked on proper QP estimation and selection of the ideal QP value of each frame, mainly based on the differences of content between frames.
Quantization Parameter (QP) does not only affect quality, but also has a significant effect on FPS and bitrate. Decreasing the QP value increases the obtained quality at expense of a more demanding codification process, and therefore, affecting negatively to FPS and bitrate. QP values of 22, 27, 32 and 37 are suggested by JCT-VC [28] to yield desirable quality. However, even with this knowledge, the optimal value is still unknown to meet the required PSNR, bitrate, FPS, and power budget at runtime. The optimal QP varies between frames due to inter- and intra-video variations, and its adaptive selection is still a subject of detailed study in the literature [82, 207].

Thread parallelism in HEVC. A key feature of Kvazaar that enables real-time encoding is Wavefront Parallel Processing (WPP) [169]. WPP divides each frame into different rows and blocks which are processed in raster-scan order (i.e., for processing one block, the blocks above and the block on its left have to be processed in advance). Kvazaar implements this technique following a task-based parallel paradigm, building an implicit graph with all the

[1]The capability of modifying the amount of threads on a frame-to-frame basis has been added in the framework of this thesis and is not part of the standard Kvazaar release.



(a) Preset and QP impact on FPS and quality using 6 threads

(b) Preset and N. Threads impact on FPS and power with QP=22

Figure 5.3: Preset, QP and number of threads impact when encoding the Quarterback 1080p sequence (see Table 6.2). All the experiments are encoded at 2.0 GHz. The legend represents different values for the preset knob: uf=ultrafast, vf=veryfast, m=medium, vs=veryslow.

dependencies between the blocks. When a block is processed, its output dependencies are released, and possibly new blocks can be processed as soon as their input dependencies are satisfied.
In order to provide runtime support to the execution of tasks and to exploit the potential data-flow parallelism, a pool of *worker threads* is deployed upon initialization. When a thread finishes to process a block, it checks whether there is a new block with no dependencies to be processed. If there is none, it remains in idle status until new blocks are released. To support dynamic number of threads, Kvazaar implementation was modified to block or wake up the number of threads dictated by the dynamic knob before the codification of each frame, offering a dynamic number of threads at each frame with no overhead. Figure 5.2 summarizes this process.

Preset. Present in many HEVC encoders, ultimately sets the value of internal knobs *before execution*. Kvazaar, for example, offers 10 different preset values, ranging from *ultrafast*, to *medium* or *ultraslow* configurations. The selection of a preset ultimately means the configuration of 24 different knobs (both static and dynamic) before the execution, offering different trades-off between quality, throughput and compression.
Given that the preset selection determines the value of the *static* knobs, the preset selection, together with the possible values of the dynamic knobs, will ultimately determine the range of the output metrics. Figure 5.3 shows the range of different output metrics when a 1080p sequence (described later in Table 6.2) is encoded with different presets and dynamic-knob configurations (QP and number of threads). Observe, for example, how even PSNR is mainly determined by the QP value, although the other static knobs set by the preset have also a slightly influence on the quality obtained. Among all the presets, ultrafast provides trade-offs supporting all the interesting range of FPS and PSNR at a lower computational cost. So, we will use it in the following.

System-level knobs:

Similar to application-level knobs exposed by HEVC encoders, modern multi-core architectures on which they typically run also expose a number of *system-level* dynamic knobs, ranging from traditional frequency scaling techniques, to newer power control mechanisms like processor turbo mode or dynamic power capping. Its proper selection offers trade-offs between power consumption and performance, and a proper adaptation to variable workloads with imposed resource limits. Similarly to HEVC knobs, we have limited out study to those dynamic knobs with the highest impact on the encoding process, but additional knobs can be incorporated into the formulation with minimal changes. These system-wide knobs include:

Dynamic Voltage-Frequency Scaling (DVFS) [32]. Mechanism to dynamically select the operating frequency of the processor, typically done through voltage modifications at the hardware level, and supported by specialized drivers at the operating system level. This technique is commonly used for reducing the dynamic power consumption [63, 9]. On modern processors, the selection of the frequency is done at the core level, allowing to have different cores running at different frequencies on the same processor. Usually, changing the frequency from the operating system has a negligible overhead of $\approx$10 µs on a modern server-oriented processor. Changing the frequency has direct implications on performance but also on power consumption [107].
However, the selection of the proper frequency at each frame is not a trivial task, and heavily depends on a correct and precise estimation of the performance and energy consumption of each frequency and the content of the frame being encoded. Previous attempts as [131, 132] have tried to model this trade-offs based on previous profiling of the architecture. However, due to the complexity of creating the model, ML techniques have arisen as a promising solution [181].

Dynamic power capping. On modern processors, the instantaneous energy consumption can be estimated based on internal hardware events and read by the running software. Having access to the underlying energy consumption allows to consider the power consumption as a knob that can be tuned dynamically. Modern Intel processors offer automatic power capping done by the RAPL mechanism, which allows to set a maximum energy consumption in short and large term. Internally, the capping is done by the hardware using Dynamic Voltage-Frequency Scaling techniques. However, in platforms where this mechanism is not present, or an application-specific power capping is needed, the limitation of the power consumption has to be done via software as shown in Chapter 4. Traditionally, power capping has been done using detailed models [16, 98] (as we explored on previous chapters), or reactive heuristics (like [75, 79]). In this thesis, we explore a third alternative based on the extraction of those models and rules through an automatic machine learning process.

TurboBoost management. Modern Intel processors (including all the i3 - i9 and Xeon families manufactured since 2008) incorporate the TurboBoost feature that allows to raise operating frequency on a per-core basis when demanding tasks are running, boosting performance under specific application requirements [158]. The frequency is accelerated above the normal operational frequency and is limited by processor's power, thermal limits,

		Num. active cores					
	Freq. Base	1-2	3-4	5-8	9-12	13-16	17-20
Normal	2.0	3.7	3.5	3.4	3.2	2.9	2.7
AVX2	1.6	3.6	3.4	3.2	2.7	2.5	2.3
AVX512	1.3	3.5	3.3	2.7	2.3	2.0	1.9

Table 5.1: Effective TurboBoost frequency on MAKALU (see Section A.1). The Turbo frequency is ultimately determined by the number of cores in use and the instruction set used by the applications.

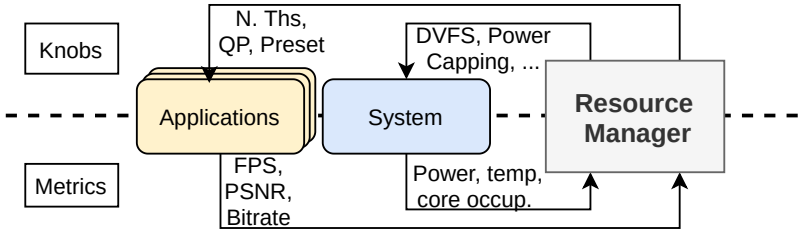


Figure 5.4: Our proposal: A centralized QoS-aware resource manager for malleable applications.

number of cores in use, and the type of vector instructions delivered [94]. Table 5.1 shows the effective frequency of TurboBoost on an Intel Xeon Gold 6138 CPU[2], (the one used later in the experiments), and how it varies depending on the number of active cores and the instruction set in use.
As the resulting frequency will depend on the status of all the socket, an individual application cannot predict the effective value of the turbo frequency by its own, so the management of the turbo frequency by a centralized manager aware of the status of the whole socket is needed. In addition, the activation of turbo frequency by an individual application can entail a negative impact on other applications relying on turbo frequency. Thus, applications need to use turbo frequencies in a restricted and intelligent way, considering a full view of the system usage. This situation adds an extra challenge in runtime frequency management, and shows how a global resource manager can be useful to solve this difficulties.

Figure 5.4 shows a general diagram of our approach, showing how a centralized resource manager is fed with metrics from applications and system status, and tunes the appropriate system- and application- knobs.

5.2.2. Motivation for dynamic resource and knob management

The development of a proactive, self-adaptive policy for resource management in a multi-user environment with multiple video requests is motivated by two main intrinsic characteristics of this kind of application, namely: (I) *intra-video requirement variations* (due to

[2] `https://en.wikichip.org/wiki/intel/xeon_gold/6138`. Last visit: Sep 2020

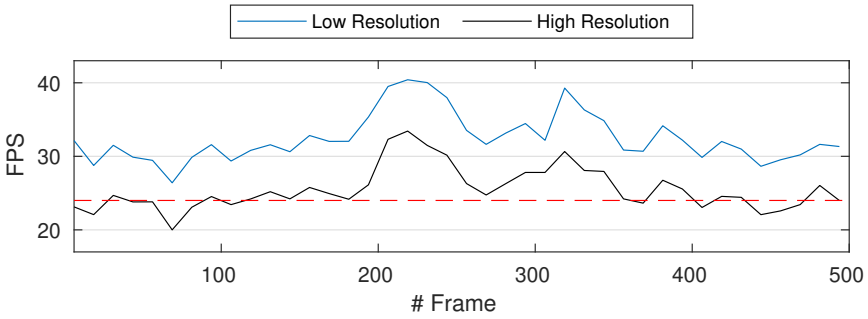


Figure 5.5: Timelines representing the FPS of the same sequence (*QuarterBackSneak*) with different resolutions: High in black (1280 × 720) and Low in blue (832 × 480), when encoding with the same knobs (3 threads at 1.5 GHz, setting QP=22 and preset=ultrafast). The discontinuous red line represents the real-time threshold (24 FPS).

changes in the content), and (II) *inter-video interactions* (both in terms of different video requirements and the variable number of running applications).

Let us illustrate this fact in terms of actual throughput (reported as Frames Per Second), extracted on MAKALU (see Section A.1), a real platform later used for the experiments, for different scenarios (varying resolution and number of concurrent encoding processes) using a static resource assignment.

Intra-video requirements variability (due to content variation). The black line in Figure 5.5 reports a timeline of the observed FPS of a complete transcoding process of a High Resolution (HR) video (*QuarterBackSneak*, see Table 6.2) using static computing resources (3 threads at 1.5 GHz, QP=22 and preset=ultrafast). Considering a target throughput of 24 FPS [129] (red discontinuous line on the plot), the video contents ultimately determine the instantaneous FPS attained. Consider, for example, the timeline between frames 200 and 350, in which the computing resources are sufficient (or even wasted) to fulfill real-time encoding. However, between frames 0-100, or 400-500, the same resources are not enough, resulting in frequent QoS violations. This is a very common situation, and demonstrates how intra-video computing requirements vary depending on the specific video contents or complexity, and how a strategy trying to determine a priori the resources needed by an encoding process is not feasible.

Inter-video requirements variability (due to different resolution). Figure 5.5 shows the FPS obtained for the same video sequence with two different resolutions: High and Low resolution (black and blue lines, respectively) using the same values of knobs -number of threads, frequency and QP-. While the low resolution video transcoding process can achieve real time transcoding (i.e., 32.5 FPS on average), the high resolution video suffers from frequent QoS violations (i.e., 25.5 FPS).

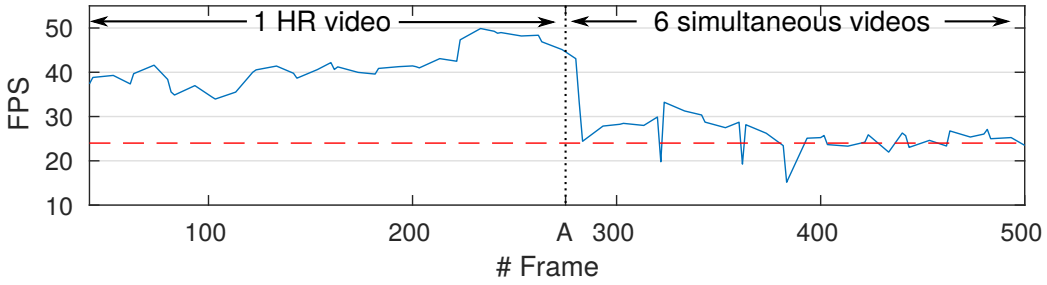


Figure 5.6: Timeline representing the instantaneous FPS of one High Resolution video (*QuarterBackSneak*) running with 3 threads and QP=22 at turbo frequency. Until point A, the video is encoded alone. From that point on, it is encoded simultaneously with other 5 videos (3 threads each). Each video is encoded on independent physical cores.

Inter-video resource contention (due to changing number of requests). Figure 5.6 simulates a situation in which an on-going isolated transcoding process with fixed assigned resources has its FPS diminished upon the appearance of other independent transcoding processes (point labelled as A in the Figure). The same sequence as that in Figure 5.5 is encoded with the same knobs, except that in this case turbo frequency is used. When the process is executed in isolation, the FPS is much higher than required; but as soon as other transcoding processes appear in the server, and although each process is mapped to independent cores in the experiment, the achieved FPS is dramatically reduced. The attained throughput is thus highly sensitive to both general video characteristics, associated assigned resources and *machine occupation*. This type of scenario results in wasted resources when the process runs alone and constant quality of service violations when the server is loaded. Obviously, some of the resources could have been distributed to accommodate all six processes. In this situation, an increase in QP, for example, could balance throughput and quality. This combined decision, however, is not trivial and must be taken in an holistic/integral way.

5.2.3. Necessity of Machine Learning for multi-user video transcoding

Once motivated the plethora of different decisions and factors that can affect throughput and quality in our scenario, we illustrate next the necessity of a machine learning-based solution to tackle the complexity and size of the decision space. Figure 5.7 gives a quantitative and qualitative overview of the amplitude of the design space considering QP, number of threads and frequency for the encoding of a single video (*FourPeople*, see Table 6.2). Each point represents an execution of the video with a different knob configuration (determined by the three axis). The average FPS of each encoding process is represented by the color the point is filled. The different plots shown in the Figure represent a different encoding scenario under a specific power cap and throughput constraints. Each colored point in the figure represents a valid solution, with different throughput (from 24 FPS in dark blue to more than 80 FPS in red), power and quality; colorless points represent solutions where power or throughput constraints are violated. Observe how areas in which both QoS and

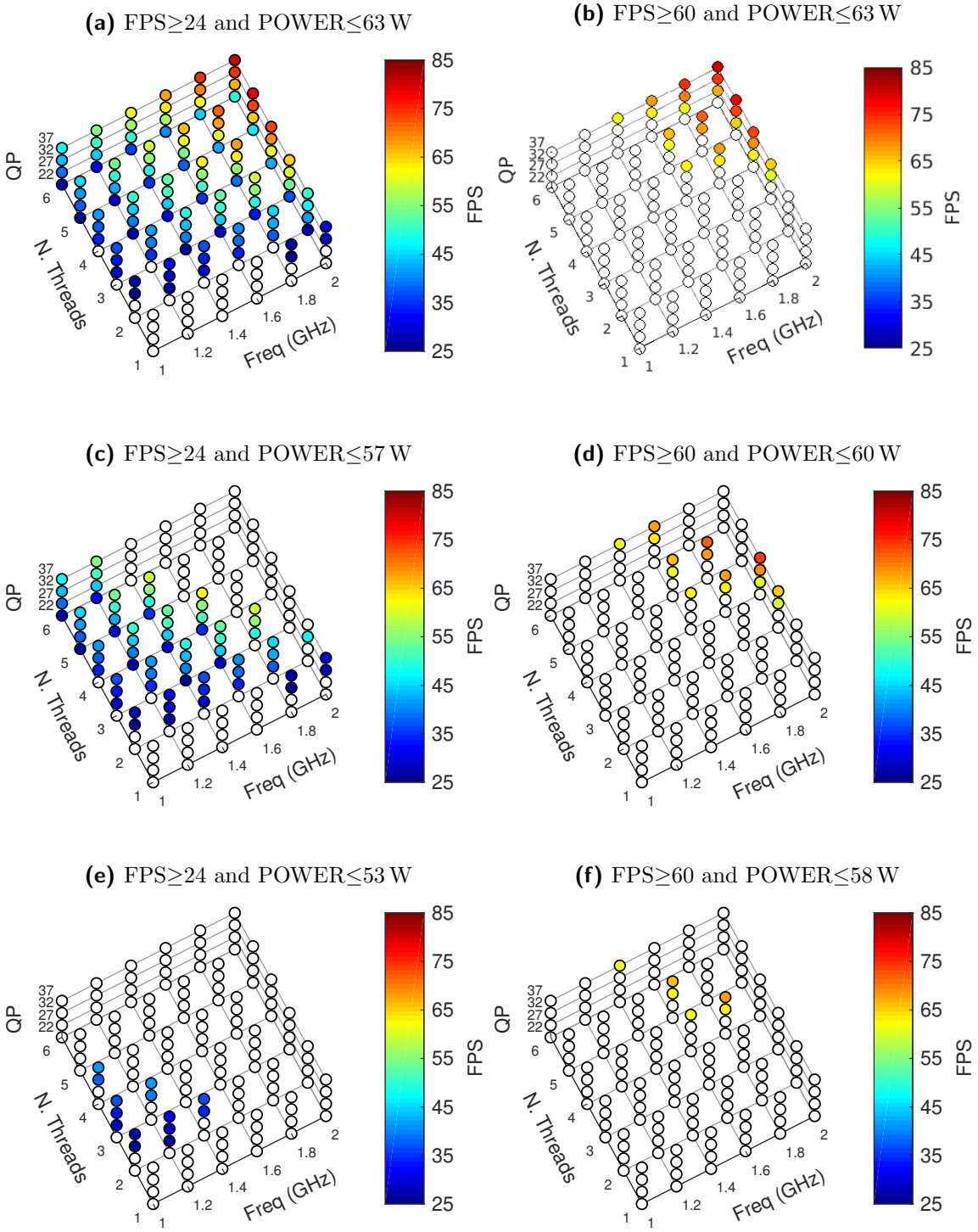


Figure 5.7: Average FPS for all combinations of QP, number of threads and frequency for three different values of power capping, when encoding the HR sequence *"FourPeople"*. Colorless points represent knob combinations not satisfying the constraints. In the experiment, idle cores were set at 2.0 GHz.

	QP	N. ths	Freq (GHz)	FPS	Power (W)	PSNR
≈24 FPS	37	1	1.8	25.9	54.2	36.4
	32	2	1.0	24.0	49.3	38.9
	27	2	1.2	24.9	52.6	41.3
	22	3	1.4	25.3	55.0	43.4
	22	5	1.0	25.1	54.3	43.4
≈60 FPS	32	3	2.0	60.2	58.1	38.8
	27	4	2.0	60.8	59.5	41.3
	32	5	1.6	60.7	57.6	38.8
	37	6	1.4	60.8	57.3	36.4

Table 5.2: Feasible knob combinations producing near-24 FPS (top) and near-60 FPS (bottom) encoding on average, and impact on the other output metrics.

power restrictions are met (FPS $\geq$ *threshold* and *power* $<$ *power_cap*) grow larger as power capping is relaxed, yielding more potential knob combinations.

Consider the left side plots in Figure 5.7, which correspond to different executions with a 24 FPS throughput constraint and different power caps requirements. The dark blue points represent, between all the valid configuration that satisfy both constraints, those which are closer to 24 FPS. Table 5.2 summarizes different assigned resources and output metrics (averaged for the complete execution). Thus, for the best quality the chosen knobs would be $QP = 22$, 5 threads and 1 GHz, while the lowest power solution would be $QP = 32$, 2 threads and 1 GHz. Plots on the right shows the solutions when the throughput constraint is set to 60 FPS (the standard value used nowadays) [85] and the power requirement varies. Observe how, when the requirements change, the optimal configurations differ considerably from previous one: the maximum quality is obtained when using 4 threads, $QP = 27$, and 2.0 GHz, while the execution with the lowest power consumption uses 6 threads, $QP = 37$ and 1.4 GHz. Observe how for the same problem, different solutions that satisfy both constraints exists, each of them optimizing different optimization goals (e.g., minimize power consumption or maximize quality).

The results above change for the different frames of the video, depending on video contents. Let us assume an encoding process with support for N_{enc} different encoding knobs values and N_{sys} different system parameters (e.g., frequency), which can be tuned at runtime, at a frame granularity. Consider now the encoding of a t-second video at a frame rate of F_r: in order to find the best encoding configuration for that frame, $N_{enc} \times N_{sys} \times t \times F_r$ profiles should be statically obtained. For instance, 4 QP values, 5 number of threads values, and 10 DVFS values, for a 10-second video at a frame rate of 24 FPS, would yield 48 000 combinations to obtain the optimal encoding configuration and DVFS settings for the video, which gives a hint of the complexity of the brute-force approach.

In summary, simultaneously dealing with: (I) performance restrictions, (II) energy consumption thresholds, (III) the increasing complexity of video encoders, and (IV) the dynamic nature of multimedia material and user requests is, obviously, a challenge for video

providers that needs to be addressed. In addition, configuring the encoding parameters is dramatically more challenging when considering multiple concurrent videos running on a server, in a so-called *multi-user video transcoding*. First, the inter-video resource contention impacts the throughput obtained by the independent encoding processes. Second, the power constraint may not let all encoding processes run at their own optimal configuration simultaneously.
Tackling these problems in a holistic, combined and automatic fashion will become mandatory in the next years. A few previous works, such as [104] and [3], have modeled the output and complexity of an HEVC encoder as a function of a few encoding parameters by exhaustive profiling of the application. However, these models only take into account a reduced number of output metrics, ignoring the impact on QoS and output throughput [97]. Moreover, since the contents of other videos may be completely different, such an exhaustive static profiling needs to be performed or each video instance in order to obtain the best combination of video quality and compression. The approach is, thus, neither practical nor feasible for real-time video encoding.

Machine learning techniques are able to consider complex, and sometimes hidden, interrelations of encoding parameters on any arbitrary platform along with video contents. This, however, requires a model-free learning algorithm, as provided by Reinforcement Learning (RL) in general, and the Q-Learning (QL) algorithm in particular, as described in the following section.

5.3. Reinforcement Learning-based formulation for Resource Management

The resource management scenario described in the previous sections can be mathematically modeled as a Markov Decision Problem (MDP). A Markov Decision Problem [20, 5] is a mathematical framework for modelling decision making problems in stochastic environments. A MDP$= (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ is defined by a finite set of states the system can be at each moment of the execution, $\mathcal{S}$, a finite set of actions, $\mathcal{A}$, that can be applied to the system and can produce a change in the state the system is, a set of probability functions $\mathcal{P} = \{P_a(s, s') : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]\}$ which determine the probability of moving from one state (s) to another (s') after taking an action (a), and a set of reward functions $\mathcal{R} = \{R_a(s, s') : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}\}$ which define how good or bad was the transition $s \rightarrow s'$ due to action a.

At each step of the process, the system in a specific state s chooses any action a that is available in state s. On the next time step, the process moves randomly to a new state s', and gives the system the corresponding reward associated to that transition $R_a(s, s')$. The destination state s' is determined by a random factor (dependent on the environment) modeled by the probability functions $\mathcal{P}$ that comprises the formulation. Observe that the destination state s' depends only on the probability functions, the current state s, and the chosen action a, but *never on previous events*. It means that a MDP makes the decisions based on the information of the current and future states, but never based on the previous states it has visited (i.e., a MDP satisfies the *Markov Property*). Although the mathematical formulation allows to have a different set of actions at each state ($\mathcal{A}_s$),

traditional approaches assume that all actions are available in all states ($\mathcal{A}_s = \mathcal{A},\ \forall s \in \mathcal{S}$), giving very low rewards to those actions which cannot be applied in specific states, meaning that those pairs *state/action* cannot be explored.

Translating this formulation into a Resource Management context, every time an event occurs in the system and the resource manager needs to act (called *step* in the original formulation), the resource manager has to decide which are the best values for all the knobs (*actions*), based on the values of the different metrics measured at each moment (*state* of the system). Once the resource manager takes an action, the output metrics of the application and system will be affected by the new knob values, moving the system to a new state. The new state will be ultimately determined by the knob values but also by the stochastic behaviour of the application and system, having to formulate these transactions through a set of probability functions ($\mathcal{P}$) depending on the current state and knob values, similar to the MDP formulation. However, the definition of states and actions based on the output metrics, as well as the definition of reward functions based on the optimization goals of the resource manager is not a trivial task.

The final goal of a Markov Decision Problem (MDP) is to find an optimal policy $\pi(s)$ which maximizes the expected accumulated reward when transitioning through the system. For a resource manager, the policy which better meets the optimization goals. Once this policy $\pi(s)$ has been found, the system will move through the different states taking the actions dictated by $\pi(s)$. Note that this definition entails that if the system is provided with different reward functions, the obtained policies will be different, even if the state and action definition are kept constant. Theoretically, it is possible for any system to find the optimal policy $\pi(s)$ by an infinite-time process. However, due to the infinite nature of the formulation, different solutions have been developed to find policies $\pi'(s)$ as close as possible to the optimal one in a finite number of steps. In addition, in most real problems, determining the probabilities or rewards that define a specific MDP is not an easy task, being in most of the cases unknown, or estimated from noisy observations. To find the policy $\pi'(s)$, multiple approaches have been proposed in the past, from Dynamic Programming [157] to Reinforcement Learning [161] approaches. In the following, we will focus on Q-Learning, as a Reinforcement Learning method to solve the problem.

5.3.1. Reinforcement Learning: Q-Learning

Reinforcement Learning (RL) is an area of Machine Learning (ML) which can tackle the problem of finding an optimal policy $\pi(s)$ for a MDP where the probabilities or rewards functions are unknown. Unlike other Machine Learning approaches, Reinforcement Learning is characterized by not needing labelled input/output pairs. Instead, the learning mechanism is based on an exploration of the solution space, building the solution progressively, while the different possible pairs *state/action* are classified from *exploration* states (the learning process is still learning the policy for that pair) to *exploitation* (when there is a final policy for that pair). Due to the exploration nature of RL, it makes all the RL algorithm family ideal for problems where the only way to collect information about the environment is to interact with it, and not to read it from a dataset.

Specifically, Q-Learning (QL) [159] has been proved as a valid Reinforcement Learning algorithm able to find the optimal policy following dynamic programming techniques [171]. Q-Learning is a model-free algorithm which can handle problems with unknown stochastic transitions based on an infinite-time exploration of the transitions between states when different actions are applied based on a partially-random policy of choosing actions. In addition, it is exploration-insensitive, thus, more suitable for practical problems [100]. However, due to the infinite-nature of the formulation of the algorithm, the optimality of the obtained policy $\pi'(s)$ will be ultimately based on the time the algorithm has been exploring the system and, in essence, the distribution of the number of times each pair *state/action* has been explored. In real-time problems, where the definition of the state comes from real measurements of the environment, this exploration time depends on the frequency the system provides the different metrics, producing long time training sessions.

A (mono-agent) Q-Learning model is composed of an agent (*learner*) able to select and take actions from a finite action set, $\mathcal{A}$, and capable of observing (*sensing*) its current state from a finite state space, $\mathcal{S}$. The agent applies actions starting from an initial state and moving to a new one. Applying particular actions in particular states is encouraged or discouraged based on a *reward* received after moving to the new state. Starting from an usually random policy to select actions, the agent is ultimately able to follow a learned policy, π', which is a mapping from the state space to the action set. This mapping simply implies if action a_t in state s_t is worth to be applied.

Learning process – method

In a Q-Learning process, a policy is determined by a table called *Q-table*. To learn the best policy, the agent maximizes the reward by storing in its *Q-table* a *Q-value* per state/action pair ($Q^\pi(s,a)$), indicating the quality of applying action a in state s. In other words, the *Q-value* represents the most probable long-term reward, provided the agent starts from state s, applies action a, and follows the policy π. Once the table is created and completely filled, the agent will take in each state the action with maximum *Q-value* (i.e., the one which maximizes the expected reward).

To learn a policy, the agent starts the process with an empty *Q-table*, which is updated at the same time the different states and actions are explored. Each time a pair *state/action* is visited, the corresponding *Q-value* ($Q^\pi(s,a)$) is updated following the Equation 5.1 [171]:

$$Q_{t+1}(s_t,a_t) = Q_t(s_t,a_t) + \alpha(s_t,a_t)\left[R_{t+1} + \gamma \max_a Q_t(s_{t+1},a) - Q_t(s_t,a_t)\right] = \tag{5.1}$$

$$= [1-\alpha(s_t,a_t)] \times Q_t(s_t,a_t) + \tag{5.2}$$

$$+ \quad \alpha(s_t,a_t) \times [R_{t+1} + \gamma \max_a Q_t(s_{t+1},a)] \tag{5.3}$$

where $Q_t(s_t,a_t)$ and $Q_{t+1}(s_t,a_t)$ are, respectively, the current and updated *Q-values* corresponding to the current taken action a_t at the current state s_t and R_{t+1} is the immediate reward after next state s_{t+1} is observed.

Gamma ($\gamma \in (0,1]$) is the discount factor and controls the significance of the history of the *Q-values* against the recently obtained reward. If γ is closer to zero, the agent will tend

to consider only immediate rewards. On the contrary, if γ is closer to one, the agent will consider future rewards with greater weight, willing to delay the reward.

$\alpha(s_t, a_t)$ determines the *learning rate*, and it is defined for each pair state/action. So, the learning can vary from one pair to another depending on the number of times each pair has been visited by the agent. Through the learning rate definition, we ensure that a specific state/action pair has been observed a sufficiently large number of times.

In stochastic environments where applying action a_t at state s_t does not always result in a particular next state s_{t+1}, the learning rate is critical to ensure a fast and flawless learning phase. If the learning rate is assumed constant and set to 1.0, the previous reinforced information is overridden every time the pair s_t/a_t is observed (Equation 5.2 values zero, setting the new value based only on the destination state by Equation 5.3). If the learning rate is constant and set to zero, there is no learning process (Equation 5.3 is not considered, and the destination state is not taken into account to determine the new value). For fully deterministic environments, $\alpha(s_t, a_t) = 1.0$ provides optimal learning.

However, for stochastic problems, a decreasing-to-zero function for learning rate [171] is able to provide optimal learning phase. A common definition for such a learning rate function is presented in Equation 5.4 [59]:

$$\alpha(s_t, a_t) = \frac{\beta}{Num(s_t, a_t)} \tag{5.4}$$

where β is a pre-defined constant ($\beta \in (0, 1]$) and $Num(s_t, a_t)$ is the number of observation of the state/action pair (s_t, a_t).

At the beginning of the exploration, when a pair is explored for the first times, $Num(s_t, a_t)$ is close to 1, (and therefore $\alpha \approx \beta$), the learned values are more influenced by the newest explored transitions than the previous ones (the coefficient in Equation 5.2 will be smaller than the coefficient in Equation 5.3). As a pair *state/action* is explored, $Num(s_t, a_t)$ value increases, and therefore, the learning rate decreases down to zero, stopping the learning rate (Equation 5.3 will be close to zero) and defining the final policy. A greater β value signifies a larger learning time, but the resulting policy will be closer to the optimal. It is responsibility of the designer of the experiment to set the β value properly, finding the trade-off between the quality of the final policy and the learning time. As explained next, a pair state/action is considered learned (and therefore, no more updated) when the learning rate α decreases under a certain value.

Learning process – phases

The previous definition of the learning rate makes the learning process an infinite process. However, as a pair *state/action* is explored, the learning rate decreases making at each exploration step smaller changes in the corresponding *Q-value*. A threshold α_{th2} is set to stop the learning process. Once the learning rate decreases below that threshold ($\alpha < \alpha_{th2}$), the corresponding *Q-value* is not updated anymore, and therefore, that pair is considered as learned. Similar to the definition of β, the value of this threshold will determine ultimately the relation between the speed of the learning process and the quality of the obtained policy.

Following the ideas presented in the literature [49], we consider three phases for the learning process: *exploration*, *exploration-exploitation* and *exploitation*, each one delimited

by two different thresholds (α_{th1} and α_{th2}). Ultimately, this phases will determine the behaviour of the agent:

$$\text{Learning Phase} = \begin{cases} 1.0 \geq \alpha > \alpha_{th1} & \text{Exploration} \\ \alpha_{th1} \geq \alpha > \alpha_{th2} & \text{Exploration-Exploitation} \\ \alpha_{th2} \geq \alpha > 0 & \text{Exploitation} \end{cases}$$

When an agent is in a specific state, the next action it will take is determined by the phase it is:

1. *Exploration phase* ($1.0 \geq \alpha > \alpha_{th1}$): At the beginning, the agent explores as many different actions as possible. In this phase, the actions are taken randomly trying to explore all the states and actions as quickly as possible. To explore all the actions uniformly, the selection is done randomly between those actions that have been visited a smaller number of times.

2. *Exploration-Exploitation phase* ($\alpha_{th1} \geq \alpha > \alpha_{th2}$): Once a state is explored enough number of times, the agent will take the action that maximizes the Q-values learned in the previous phase trying to maximize the expected reward. The learning process has not finished yet, so the different Q-values are still updated as described before.

3. *Exploitation phase* ($\alpha_{th2} \geq \alpha > 0$): In this phase we consider that the agent has completely learned, and therefore, it has obtained the final policy π'. In this case, the agent will take the action that maximizes the Q-values similar to the previous phase ($a = \arg\max_{a \in \mathcal{A}} Q_t^{\pi}(s_t, a)$). The Q-table is not updated anymore.

Having a different learning rate for each pair *state/action* allows to have a more precise control of the learning progress of each state. However, it is possible to have for a specific state, different actions in different phases of the learning process. In this cases, it is not clear which action should an agent choose, as it will depend on the learning phase. To solve this issue, we have followed a two step approach: (I) In the first step, the agent chooses the phase it is going to explore, but not the specific action. This is because the method to choose the next action will depend on the phase selected (for example, in exploration it will choose an action randomly while in exploitation it will choose the action which maximizes the expected reward), and (II) In the second step, the agent chooses, between the actions belonging to the phase selected on the previous step, the next action following the selection rules of each phase described before.

Algorithm 5.1 shows the pseudo-code followed by an agent to choose the next phase to explore (step I). If all the actions belongs to the same phase, the decision of which phase select next is clear (1–5). However, in the case where actions belong to different phases, the agent will try to progress the learning process exploring those actions that are still in the *exploration* phase first. To do that, the agent will choose randomly the next phase to explore giving the double of probability to the *exploration* phase than the other two phases, trying to choose the exploration phase over the other phases, and therefore, trying to avoid to keep actions in exploration phase too much time. To do that, in the case there is at least one pair in exploration phase, the agent will give a probability of 1/2 to select that phase, and a probability of 1/4 to the other phases (lines 5 to 10). The decision between *exploration-exploitation* and *exploitation* phases are done with the same probabilities.

Algorithm 5.1: Algorithm to determine the phase of the next action to choose.

Data: number of actions at each phase (*nExploration, nExplor_Exploi, ...*) and number of available actions ($|\mathcal{A}|$).
Result: choosen ← phase of the next action to choose.

```
1  begin
       /* =A=:  All the pairs are in the same phase:                    */
2      if nExploration = |A| then choosen ← exploration;
3      else if nExplor_Exploi = |A| then choosen ← exploration_exploiation;
4      else if nExploitation = |A| then choosen ← exploitation;
       /* =B=:  Pairs mixed with different phases                       */
5      else if nExploration > 0 then    // If ∃ exploration, set double prob.
6          if rand() ≥ 0.5 then                                  // P=0.50
7              choosen ← exploration;
8          else                                                  // P=0.25
9              choosen ← randomSel(exploration_exploiation, exploitation);
10         end
11     else      // if ∄ exploration, same prob.  to explorExploi & exploi
12         choosen ← randomSel(exploration_exploiation, exploitation);
               // P=0.50
13     end
14 end
15 Function randomSel (phase1, phase2) is:
16     Selects randomly between both phases.
17     Checks if there is at least one action in each phase before choosing it.
18 end
```

5.3.2. Mapping a generic QoS-aware application to a Q-Learning formulation

Similarly to a Markov Decision Problem where the ultimate goal of the system is to transit between states that maximize the expected accumulated reward, the ultimate goal of a resource manager is to tune the system and applications properly during the whole execution to satisfy the optimization goals as much as possible at any given time. To solve the problem, a MDP formulation is based on a set of *states* in which the system can be, a set of *actions* the system can take and ultimately makes the system to transit between states, and a set of *rewards* that define how good or bad is the transition between two states caused by the taking of a specific action.

Following the same formulation, a resource manager can be defined in terms of the values the different metrics report at each moment (*states*), the different knob configurations the system can set (*actions*), and how much the optimization goals are satisfied at each moment of the execution (*rewards*).

Formulating a resource management scenario as a Q-Learning scenario, rather than a more traditional heuristic-based scenario has a considerable number of advantages, namely:

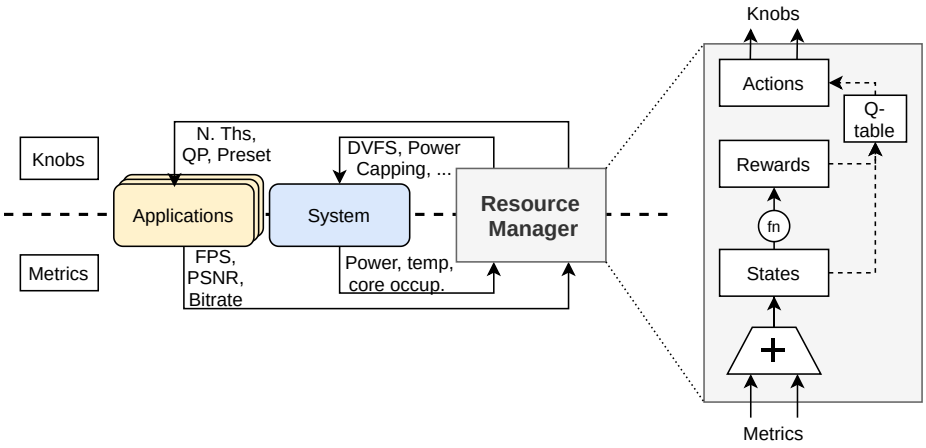


Figure 5.8: Application- and system-metrics transformation into QL states and rewards.

- Supposing the states, actions and rewards are properly defined, the policy extraction is done automatically by the agents; in this case, expert knowledge of the problem is unnecessary or reduced, differently from a heuristic approach in which knob tuning needs to be performed manually.
- Due to the exploration nature of the process, Q-Learning agents are able to learn by their own the transitions and probabilities relating different states and actions ($\mathcal{P}$), which are usually difficult to identify by a human expert.
- Although Q-Learning takes the decisions based only on the current state, the learning process is formulated to take into account both the current and destination states, making the decision process aware of the implications of the current actions on future states, and therefore, producing better long-term policies.

Figure 5.8 shows a detailed view of how a resource manager transforms the application- and system- metrics into internal states assigning a reward value to these states, later used to determine the next actions to take (i.e., how to tune the different knobs). However, the quality of the obtained policy and the learning time, is ultimately determined by the correct definition of these states, actions and rewards. This is still far from being a trivial task, and in this decision, expert knowledge is still needed. This makes the process of defining a resource manager in terms of Q-Learning a hard problem, usually tackled by a tune-and-test process, where the obtained policy has to be tested and states and rewards have to be tuned based on the observed results. In particular, this methodology can be summarized as:

1. In a first step, a definition of states, rewards and actions has to be done based on the available knobs, metrics and expert knowledge of the problem.
2. Second, the system is trained with a predefined set of inputs, obtaining a policy for the system (i.e., a *Q-table* with all the pairs in *exploitation* phase).
3. This policy is later applied to another predefined set of input different from the one used to train the system.

4. Based on the results observed for this input set, states and rewards are tuned, repeating all the process from step one again.

Although this is a complex and large process, it can be alleviated by a proper initial state and reward definition, and speeded up storing intermediate results as shown later in Chapter 8.

The previous QL formulation presents the learning process as an intensive exploration process where the final policy is only obtained when all state/action pairs have been explored completely a minimum number of times (i.e., all pairs are in *exploitation* phase). Indirectly, this problem formulation requires the exploration space to be finite allow the exploration of all the actions in a finite amount of time. However, although this is normally the case for the action set (usually the knobs can only be configured with a predefined set of different values), this is not the case for states, since many of the metrics that define the system report continuous values, and therefore, they have to be discretized.

If the metrics are discretized in a small number of buckets (or intervals), the learning process will be short as the agent will need to explore a low number of states. However, if the number of states is too small (i.e., each state covers a huge interval), it can lead to bad quality policies. Suppose a scenario where, after applying two different actions (modifying two different knobs), the measured values from the metrics are close enough to be classified in the same state. In this scenario, the agent will not be able to distinguish the effects of both actions, and it will consider them equivalent, ignoring optimization opportunities offered by each knob. On the contrary, if each state covers a smaller interval, it can mitigate the previous problem at the expense of a longer learning period. But, if the covered intervals are too small, in stochastic problems the effects of a specific action can be classified into two different states, producing again a bad quality policy. The trade-off between the space decomposition, the learning time, and the quality of the learned policy has to be done by a human expert, and usually is done through a trial-and-test process.

Similar to the state definition, the optimization goals of the system have to be discretized into reward functions giving different rewards to different states. Those states that best satisfy the optimization goals will have a higher reward, while those states farther away from the desired ones will receive lower rewards. However, if the reward given to two different states is similar, the system will not distinguish which state is the best, obtaining not so good policies. On the contrary, if the difference between the rewards given to two similar states are too big, the system can learn not to transit between those states, even if moving to a worse state allows the system to move to a better state in the future. The process of defining the rewards that best fit the optimization goals of the system requires expert knowledge of the problem, and usually is done via a test-and-tune process.
Traditionally, reward functions have been used to model maximization or minimization objectives, however, the use of negative or zero rewards can be used to model other behaviors like constraints, or forbidden actions. For example, in scenarios where not all the actions are available in all the states, giving a negative reward to these forbidden pairs *state/action* will make the system to learn not to chose these pairs in the future.

Together with the definition of the states, the proper definition of the rewards will ultimately determine the quality of the learned policy, and will set a trade-off between the quality and the learning time.

5.4. Conclusions

Extending resource managers to consider specific knowledge of the running applications (both in terms of metrics and tunable knobs) allow them to perform a deeper control of the system, and to consider new optimization goals derived from the internal metrics. This is of special importance for QoS-aware applications, which do not benefit from obtaining the maximum performance, but have special requirements in other metrics. A representative example of a QoS-aware application is HEVC online video transcoding, where a minimum requirements in terms of throughput, quality and compression have to be satisfied. Trying to fulfill all of these requirements, HEVC video encoders expose a set of different tunable knobs able to modify one or multiple of those metrics. However, the multiple relations between the different knobs and metrics, as well as the strong dependencies present in this kind of applications (both intra- and inter-application dependencies), makes this kind of scenarios hardly manageable by traditional heuristic-based approaches.

Obviously, the exposition of these new metrics and knobs to resource managers entails a huge decision space to consider. Machine Learning approaches have emerged as promising solutions able to handle this type of scenarios. Specifically, resource managers can be formulated in terms of a Markov Decision Problem, where Reinforcement Learning approaches can easily find policies that satisfy all the optimization goals through a dynamic programming learning process. However, even if mathematically the definition of a resource manager is similar to the definition of a MDP, the actual definition of states, actions and rewards are far away from being a trivial task.

In the next chapter, we show how an online video-transcoding scenario can be mapped into a Q-Learning approach, managing all the intra-application dependencies to obtain a real-time codification process fulfilling quality requirements at the same time. In addition, we show how the Q-Learning formulation presented in this Chapter can be extended to a stochastic multi-agent formulation, comparing our approach with a static and a heuristic approach.

6

Self-adaptive Application Execution via Reinforcement Learning

The integration of internal metrics delivered by malleable and QoS-aware applications, together with the ability to manage their internal application-specific knobs offers resource managers a new plethora of potential optimization opportunities, both in terms of metrics to be monitored and optimized, and also in terms of which and how knobs should be tuned in order to fulfill a certain level of QoS with a proper resource usage. However, as described in the previous chapter, the management of this kind of applications involves a new set of dependencies to be handled (intra- and inter-application), and generates massive decision spaces, hardly manageable by traditional heuristic-based resource management schemes. A representative example of this kind of applications is HEVC online video transcoding, as it offers a range of different metrics and knobs to tune, and at the same time, its behaviour heavily depends on the contents of the sequence being encoded, and ultimately also on other encoding processes running simultaneously in the system.

Machine Learning has emerged as a promising solution to tackle these burdens. Specifically, provided a proper formulation of the problem is given, Q-Learning promises to obtain high-quality policies through an *unsupervised exploration-based process*. In particular, following with our driving application, a reformulation in terms of a Q-Learning process mixing knobs and metrics from the application (e.g, QP knob or Frames Per Second, respectively) with external ones (e.g., processor frequency or power consumption) is an approach of wide appeal to learn complex dependencies between them, and properly apply the most suitable set of actions. However, although the canonical formulation of Q-Learning presented in the previous chapter is perfectly valid, it presents a set of drawbacks when implemented in a real system. The first one is the considerable learning time directly derived from the exploration-based nature of the algorithm. The second one is related with the stochastic nature where the learning process is usually developed, that typically introduces noisy measurements with a direct impact on the quality of the extracted policy.

In this chapter, we introduce a modified multi-agent implementation of the canonical Q-Learning formulation that addresses and solves the aforementioned problems, and we demonstrate it to be a valid solution for its integration into a real resource manager. Specifically, throughout the chapter, we:

1. Show how an HEVC encoding process can be formulated in terms of a Q-Learning algorithm, modelling real-time throughput restrictions and quality requirements.
2. Extend the previous Q-Learning formulation to: (I) consider a multi-agent formulation, reducing the learning time at the same time multiple agents co-operate to obtain a better policy, (II) incorporate the stochastic component into the decision-making process, and (III) mitigate the negative effects induced by noisy metric measurements.
3. Implement and test the described approach in a real multi-core server, comparing our proposal with a static and a heuristic-based approach in terms of throughput constraint violations and attained quality.

Section 6.1 describes how an online HEVC encoding process can be formulated in terms of a Q-Learning algorithm, mapping the different metrics, knobs and optimization goals into specific states, actions and rewards, respectively. Section 6.2 extends the previous formulation into a co-operative multi-agent formulation. It also reveals how problems derived from a stochastic environment with noisy metrics can be solved and integrated into the formulation. In Section 6.3, we compare our approach with a static approach (in which knob values are fixed a priori and are not modified throughout the encoding process) and a heuristic-based state-of-the-art strategy, both in terms of real-time constraint violations and obtained quality for a varied set of videos of different content and resolution. Finally, Section 6.5 shows some conclusions and final remarks.

6.1. A Mono-agent Q-Learning formulation for video transcoding

To show how the resource management of an online encoding scenario can be formulated in terms of a Q-Learning algorithm, we describe next a scenario in which one video is encoded in an isolated fashion in the system, under two simultaneous encoding requirements, namely: (I) applying real time throughput requirements (in our case, FPS$\geq$ 24), and (II) at the same time, maximizing quality.

In the following, our formulation responds to these requirements in throughput and quality by tuning application-wide dynamic knobs (QP and number of threads) and system-wide knobs (core frequency) while monitoring specific application and system metrics (throughput and quality, and core occupation, respectively). In Chapter 7, we extend this formulation to consider power consumption as an additional constraint. The considered scenario includes two additional conditions, namely: (I) resource usage should be *minimized*, provided THROUGHPUT is met, and (II) only if THROUGHPUT constraint is satisfied, quality should be *maximized* provided there are enough available resources. In addition, two different resolutions will be supported (High Resolution (HR) and Low Resolution (LR)), affecting how our formulation is done.

Gathering all the aforementioned restrictions, the scenario is complex enough to illustrate the potential of our RL-based resource management scheme. In Chapter 7, this

scenario will be extended to encode *multiple* videos simultaneously and therefore, to consider not only intra-application knob and metric dependencies, but also the inter-application relationships and their implications in resource management.

6.1.1. Problem mapping to a Q-Learning formulation: states, actions and rewards

As described in Section 5.3.2, mapping this problem statement into an actual Q-Learning process requires a correct definition of *states*, *actions* and *reward functions*. The *number*, *granularity* and *value distribution* of states and actions is ultimately a trade-off between learning time and control degree on the accuracy of output metrics, in which both expert knowledge and application specifics play an important role. A correct definition of the reward function will ultimately determine the success in maximizing/minimizing output metrics, optimality in resource usage and the compliance with the imposed restrictions.

State definition

Designing and training a Q-Learning system is a trial-and-error process in which the designer of the experiment modifies and tunes the state and reward function definitions until the obtained policy meets the desired behavior. Although this process can be alleviated by incorporating expert knowledge of the problem into the tuning process, it is still an extremely complex and long process. The decomposition of the states and reward spaces into multiple simpler sub-spaces can reduce the complexity of this process.

Instead of placing a unique and complete definition of state, we propose the division of the states space into different independent sub-spaces (i.e., the metrics required to build one sub-space are not employed in the creation of any other sub-space). This state definition allows a more fine-grained control on the behaviour of the system, and provides a direct way to verify how a change in the definition of the problem impacts each sub-space independently from the others. In addition, as in our formulation each output metric, measured from the application or from the system, is mapped to only one sub-space (to have independent sub-spaces), it is relatively straightforward to determine, both for the human and for an agent, the reason of a change in one sub-state, and therefore, to incorporate those changes into the learned policy:

$$s = (s_1, \ldots, s_n), \quad \text{with } s \in \mathcal{S} = \mathcal{S}_1 \times \ldots \times \mathcal{S}_n,\ s_i \in \mathcal{S}_i \tag{6.1}$$

Since our target scenario aims at attaining QoS-aware real-time transcoding, the state space is divided into three different sub-spaces in our formulation, each one related to a different monitored metric: PSNR, FPS, and occupation level of the processor (measured in terms of the number of cores in use). While the first two sub-spaces correspond to metrics directly sensed from the application, and optimization goals of the system, the last one is a system-wide metric; it will assist the resource manager to deal with turbo frequencies, as described next. The system constantly senses the output metrics on a frame-to-frame basis, discretizating the measurements and mapping each particular observation into a particular state.

QUALITY (S_{psnr}): For 8-bit-depth videos with lossy compression, the video quality should range between 30 dB and 50 dB for acceptable human vision [194]. In our setup, we divide this range into the following intervals to constitute PSNR states:

$$\mathcal{S}_{psnr} \equiv \quad \{\, \text{PSNR} \in (0,30), [30,35], (35,40], (40,45], (45,50], (50,\infty)\ dB \,\}\,.$$

Although the PSNR metric is measured every frame, the state is built based on the average value of the last frames. The amount of frames considered will depend on the frequency the agent acts, and therefore, the frequency the state is read and built.

THROUGHPUT (S_{FPS}): As our scenario aims at encoding sequences under a strict real-time limit (24 FPS), and at the same time reducing resource usage, this ultimately means to encode the sequences never under the throughput threshold and as close as possible to 24. Following this dual purpose, we divide S_{FPS} into the following states: $FPS < 24, < 28, < 32, < 35, < 40, < 50$ and ≥ 50. This division allows us to return a negative reward to those states below 24 FPS, and to distinguish the goodness of each state above the threshold.

$$\mathcal{S}_{FPS} \equiv \quad \{\, FPS \in (0,24), [24,28), [28,32), [32,35), [35,40), [40,50), [50,\infty) \,\}$$

This non-regular formulation of the states is a trade-off between quality of the obtained policy and learning time. On one side, having smaller intervals near the threshold allows the system to distinguish the effects of similar actions and to have a more precise control on the applications. On the other side, having larger intervals far from the threshold reduces learning times, with a minimum impact on the quality of the obtained policy.

However, this space decomposition is not useful by its own, and it needs to be accompanied with the correct reward function able to exploit this asymmetric state classification. Similar to PSNR, the FPS metric is measured in a frame-to-frame basis. However, due to the huge variation measured in this metric across frames (mainly due to content changes), a moving mean is used to build the state.

TURBO (S_{occ}): As detailed in Section 5.2.1, the use of turbo frequency management into our framework motivates the introduction of buckets of *level of occupation* (in terms of number of cores occupied at a given execution point by all transcoding processes co-existing in the system) into the state definition as an additional state (S_{occ}), as the actual processor frequency directly depends on this value. In our platform, we distinguish 6 different levels of occupation:

$$\mathcal{S}_{occ} \equiv \quad \{\, \textit{N. active cores} \in [1,2], [3,4], [5,8], [9,12], [13,16], [17,20] \,\}$$

This metric depends exclusively on the status of the machine, and the reported value corresponds to the current value, differently from the average value taken on the rest of the metrics. Even if this state is not useful for encoding an individual sequence, it will be of great importance when multiple videos are encoded simultaneously, using almost all the resources of the machine. This scenario (multiple sequences being encoded simultaneously) is explored later in Chapter 7.

Experimentally we have checked that the compression achieved in all of our tested sequences is below the range of users' network bandwidth around the world, both for 3G (from 6 Mbits/s to 2 Mbits/s) and 4G (from 17 Mbits/s to 7 Mbits/s) [21]. This is the reason why *bitrate* is not considered as a metric to tune (and therefore, a state to consider). However, bitrate can be incorporated into the formulation following a similar methodology applied to the other metrics.

Reward Function. Sub-reward definitions

The definition of the reward function follows a similar decomposition as that used for the state definitions: instead of designing a unique definition for it, we propose the use of multiple *sub-reward functions* (normalized to 1), each one valuing the goodness of each sub-state. Then, we combine all sub-reward functions into an unique composed reward through the use of different coefficients:

$$R(s) = R(s_1, \ldots, s_n) = \lambda_1 \cdot R_1(s_1) + \ldots + \lambda_n \cdot R_n(s_n), \quad \text{where } s_i \in \mathcal{S}_i,\ \lambda_i \in \mathbb{R} \tag{6.2}$$

Although the behaviour of the global system depends on the combination of the different sub-reward functions, each sub-reward can pursue a different sub-goal (for example, one reward can try to maximize one metric while other sub-reward tries to minimize another independent metric). Additionally to maximize or minimize one metric, a reward definition can reflect a constraint not to exceed. To do that, returning a value low enough (even a negative value) for those states will make the system to learn to avoid those combinations that moves the system to those states. In our formulation, we have determined that a reward of -4 in a sub-state is low enough to consider the global state as bad (having in mind the different λ values as well as the values reported by the rewards of the other sub-states).

In our specific problem, we propose two different reward functions showing all the aforementioned behaviors: minimizing a metric under one lower limit, and maximizing a metric between two different limits. Note that this two situations can emerge in a wide variety of applications, not exclusively in video transcoding.

QUALITY **(maximizing output metric between two bounds):** As explained in Section 5.2.1, a minimum PSNR of 30 dB guarantees an acceptable quality for the human vision. However, assuming there are available resources, quality should be maximized to improve both QoS and QoE. Hence, this (sub-)reward function assigns higher rewards to states with higher quality, decreasing the reward for those states with lower quality, as follows:

$$R(PSNR) = \begin{cases} -4.0 & \text{PSNR} < 30 \text{ or PSNR} > 50 \\ a \cdot e^{\text{PSNR}/50} - b & \text{otherwise.} \end{cases}$$

where a and b are set to return a maximum reward of 1.0 when PSNR=50, and a reward of 0 when PSNR=30. Observe how the constraint of being always above 30 dB is formulated giving a negative reward to the states below. Similarly, states with quality greater than 50 dB are penalized as quality greater than that is imperceptible for the human vision, but ultimately translates into a waste of resources.

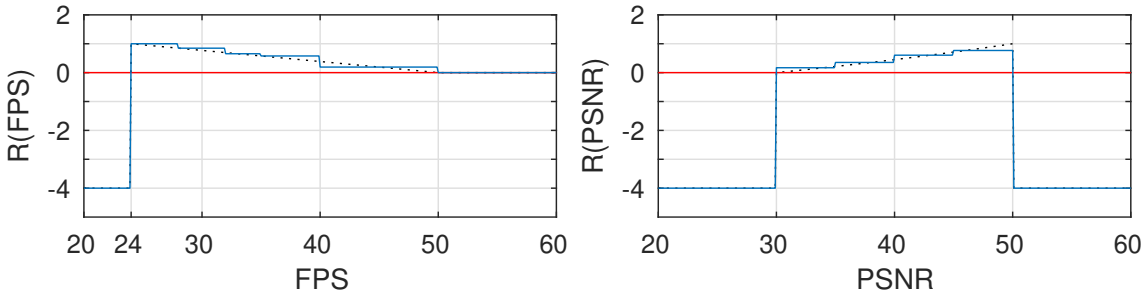


Figure 6.1: Reward functions for THROUGHPUT and PSNR. Blue and dotted black lines show the reward after and before discretization, respectively. All functions are normalized between [0,1]. Constraint violations are penalized with a negative reward of -4.0.

THROUGHPUT **(minimizing an output metric under one lower bound):** We define the following reward function based on the target frame rate (24 FPS):

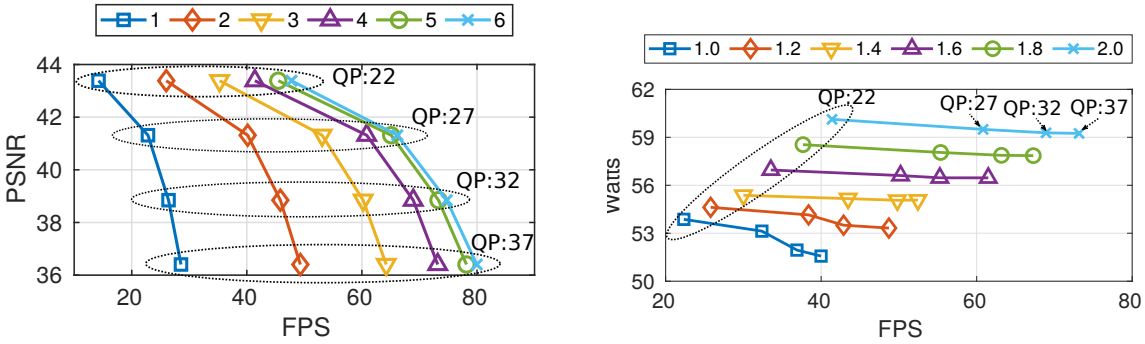
$$R(FPS) = \begin{cases} -4.0 & FPS < 24 \\ a \cdot FPS + b & 24 \leq FPS < 50 \\ 0.0 & FPS \geq 50. \end{cases}$$

This (sub-)reward function provides negative values if the throughput is smaller than the target frame rate. The a and b parameters are adjusted to produce a maximum reward of 1.0 if FPS meets the target of 24, and a decreasing reward down to 0 for larger FPS. The reason is that achieving larger FPS may result in wasting resources, which ultimately means fewer users can be served. In the case where FPS > 24, spare encoded frames can be buffered. Buffered frames can be used to compensate the overall framerate if, at some points, FPS temporarily drops below the target.

Although the proposed rewards describe continuous functions, they need to be discretized in the same way the states were discretized in the previous section. Figure 6.1 illustrates the reward functions defined for FPS and PSNR in our problem formulation, and the effective values after they are discretized (blue lines). The dotted black lines represent the rewards before the discretization.

Actions definition

The definition of actions taken by the system (both in terms of number and distribution within a range), similarly to states, must be chosen based on expertise (problem knowledge) and requirements of learning time and accuracy. The different actions to consider are determined by those knobs to tune, and their possible values. Figures 5.3 and 6.2 provide Pareto curves that relate different output metrics and actions for a transcoding process on the target architecture: MAKALU (see Section A.1). Based on those results, the throughput, quality and bitrate requirements of our scenario, and the fact that *preset* knob is static and hence cannot be modified once set at the beginning of the execution, we have determined



(a) Number of threads and QP impact on Quality (PSNR) and Throughput (FPS)

(b) Frequency and QP impact on Power consumption and Throughput (FPS)

Figure 6.2: Metrics obtained with different knobs, setting 2.0GHz (left), and setting 4 threads (right), while encoding a 1080p-video using the default *ultrafast* Kvazaar configuration.

that the *ultrafast* preset fulfills all the requirements offering a good trade-off between quality and resource consumption. Therefore, in the following, all the experiments are configured to use the *ultrafast* preset; hence, it is not necessary to incorporate the preset knob into the decision process. For the remaining knobs, we base our decisions on the experimental results shown in Figures 5.3 and 6.2:

QP: As described in Section 5.2.1, QP is one of the most important encoding parameters, as it affects FPS, PSNR, and bitrate [95, 137]. Although QP can take a wide range of values (from 1 to 51), we use the same QP values proposed by JCT-VC: $\text{QP} \in \{22, 27, 32, 37\}$. Experimentally we have determined that these values produce valid encoding results for our requirements, as shown in the plots.

Number of Threads: While HEVC encoding can always benefit from multithreading to increase FPS, the results in Figure 6.2a reveal that Frames Per Second saturates above a certain number of threads. In addition, this limitation varies depending on the resolution of the encoded sequence. Based on this observation, we consider a limited number of threads: from 1 to 5 threads for High Resolution videos, and up to 3 threads for Low Resolution sequences.

DVFS: Our specific platform (see Section A.1) supports frequencies ranging from 1.00 GHz to 2.00 GHz and Turbo, selectable on demand and in a core-by-core basis in steps of 100 MHz (with Turbo enabled, this range extends up to 3.7 GHz, but is not under direct control of the user, as it depends on the current core occupation). Therefore, we consider the selection of frequencies within this range and granularity as valid values for the DVFS knob.

6.1.2. Mono-agent Q-Learning: formulation and drawbacks

As we are targeting the encoding of videos with two different resolutions in our scenario, each behaving differently for the same knobs modification, the original agent in our design is instantiated multiple times in our system, one in charge of a different resolution. These agent instances are defined to be independent, meaning that there is not any communication or information shared between them. In particular, each agent has its own *Q-table*, and the learning process is carried out independently by each one. On top of all the agents, a simple piece of logic is in charge of activating/deactivating the corresponding agent based on the resolution of the video being encoded. As we are considering scenarios where only one video is simultaneously encoded, in practice this means that only *one agent* is acting at each moment.

Intuitively, deploying only one agent to handle both resolutions will ultimately induce the system to learn poor-quality policies. In this regard, as the system can measure different output metrics when applying the same action to the same state, the outcome of the learning process will be sub-optimal, as the destination space will depend on the resolution of the video being encoded. Adding a new subspace to the space definition pointing to the current resolution of the video solves this problem. However, this solution is equivalent to having multiple agents, each in charge of a different resolution. Moreover, having different independent agents can improve the learning process in the sense that the total learning time can be split into smaller learning process, each for a different agent. The actual impact of the multi-agent approach is revealed in Section 6.3.

In the specific case of video transcoding, the modification of a knob does not always yield an instantaneous consequence on the encoding process, but the effects can appear a number of frames after. Experimentally, we have determined that the maximum delay is lower than 6 frames in our platform, meaning that an agent should act, at maximum, every 6 frames (i.e., supposing a constant framerate of 24 FPS, the agent will act 4 times per second).

Although the action space was divided into different sets in the previous section, having only one agent in charge of tuning the different knobs ultimately means that the agent has to consider all the possible combinations of all the different knobs every time it takes an action. In our formulation, that means 240 and 144 different knob configurations for High Resolution and Low Resolution videos respectively (4 QP values $\times$ 5/3 possible threads $\times$ 12 different frequency values = 240/144). A similar observation can be extracted for the space definition: although the space is divided into multiple sub-spaces, all of them need to be considered as a unique space. This ultimately means that the agent needs to consider 126 different states (considering that S_{occ} can take only three different values when only one video is encoded).

Altogether, this makes the exploration space for one agent to be composed by $126 \times 240 = 30\,240$ different *state/action* pairs in our scenario. Under the assumption that the agent is able to maintain an average framerate of 24 FPS, and it acts every 6 frames, the amount of time needed to explore each pair at least one time is 2.1 h in perfect conditions $\left(30\,240\,\text{actions} \cdot \frac{6\,\text{frames}}{1\,\text{action}} \cdot \frac{1\,\text{s}}{24\,\text{frames}} \cdot \frac{1\,\text{h}}{3600\,\text{s}} = 2.1\,\text{h}\right)$. However, if the framerate decreases under 24 FPS, or the states are not uniformly explored, the learning time can increase even more. Also, for a proper learning process, each pair needs to be explored

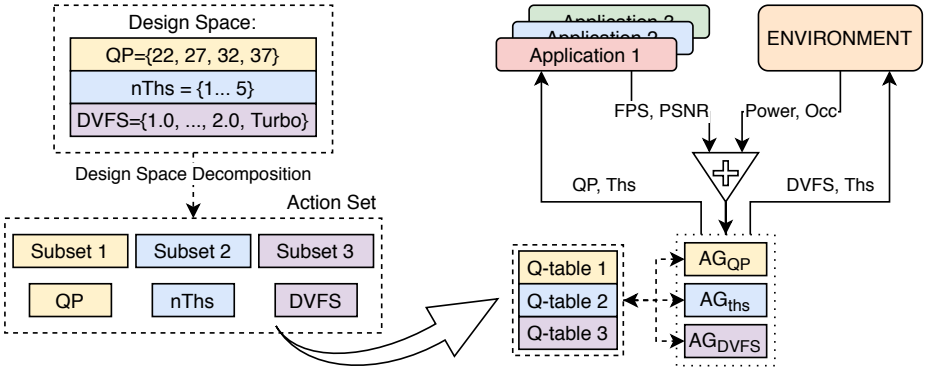


Figure 6.3: Proposed multi-agent Reinforcement Learning approach.

multiple times, making the learning time far from being negligible.

Multi-Agent Learning (MAL) addresses those problems by decomposing the problem domain into smaller sub-problems, deploying multiple agents acting simultaneously, each one in charge of a different knob. In MAL, multiple agents need to interact and behave cooperatively or competitively with some degree of autonomy. As a result of such *co-operative* and *concurrent* learning, it is feasible to deal with considerably larger search spaces. Therefore, if complexities arising from interactions between agents are correctly managed, co-operative multi-agent learning is a promising alternative to explore larger design spaces with less computational complexity, leading to faster learning phases compared to mono-agent learning.

The main challenge of co-operative concurrent learning is that each learner (agent) needs adjust its behaviour according to the others. In the next Section, we show how our problem formulation can be extended to a Multi-Agent Q-Learning scenario.

6.2. Integrating Multi-Agent Learning

Similar to conventional mono-agent learning, the QL algorithm in multi-agent learning is composed of a finite action set $\mathcal{A} = \mathcal{A}_1 \times \ldots \times \mathcal{A}_n$ split in multiple independent, a finite state space $\mathcal{S}$ and a a reward function valuing the optimality of each action at each state. Each agent AG_i is in charge of taking action $a_t^i \in \mathcal{A}_i$ at each moment t, and moves the whole system from its current state s_t to the next one s_{t+1}. The state definition is shared between all the agents, having each a different Q-table. Then, the corresponding Q-table [171] is updated after each reward, showing the value of applying a_t^i at s_t, is received.

In our MAL formulation, we consider three different agents, each in charge of a different action subset (i.e., in charge of tuning a different knob). Each agent modifies its internal *Q-table*, but at the same time it is able to consult other agents' *Q-tables* during the decision process. This makes the agents to *cooperate* on the decision process instead of competing, as described later. Figure 6.3 shows an overview of the proposed approach where three agents cooperate with each other.

This new formulation of the system is able to reduce the learning time (the action space is split and therefore reduced) at the same time it can improve the quality of the learning

Agent	Action frequency	Main target	Action Origin
AG_{qp}	$k \cdot 24 + 0$	Application	Application
AG_{thread}	$k \cdot 12 + 1$	Application and Platform	Application and Platform
AG_{dvfs}	$k \cdot 6 + 2$	Application and Platform	Platform

Table 6.1: Agent configuration overview.

policies (due to the cooperation between agents). However, modelling the system as a multi-agent approach entails a new set of difficulties to tackle:

1. Determining the best activation sequence and frequency of each agent so they do not compete, but the actions chosen by one agent can be useful by the agents acting on the next frames.
2. The previous learning rate function (Equation 5.4) was designed to consider only one agent, not being valid for scenarios where multiple agents learn simultaneously at different speeds.
3. The decision-making process has to be reformulated to incorporate the cooperative process at the same time it is adapted for a stochastic environment.
4. Measuring metrics in a real system can led the system to read noisy measurements and affect negatively the learning process; those *state/action* pairs that are affected by wrong measurements need to be cleaned.

6.2.1. Agent design and activation sequence

In MAL, we propose the use of three different agents acting cooperatively, based on the different knobs to tune. Therefore, we define agents for tuning QP (AG_{QP}), deciding the number of threads used to encode a frame (AG_{thread}) through Wavefront Parallel Processing (WPP), and per-core DVFS (AG_{dvfs}). Similarly to the mono-agent approach, an independent set of agents is configured for each resolution. If needed, adding the resolution into the state definition allow to have an unique set of agents for all the resolutions at the expenses of increasing the learning time.

We consider a different action frequency for each agent based on the environment domain that each agent can directly influence, and the origin of their action (from the application, or from the platform). Action granularity is also directly related with the agent activation sequence, and with the relative effects on output metrics of a single step variation in each action. Hence, one step in terms of QP implies large modifications in both quality and throughput (see Figure 6.2). For the latter, wrong actions taken by the QP agent can be solved or alleviated, with more detail, by subsequent application of actions by the threads or DVFS agents, each one with progressively higher granularity. This is actually the activation sequence of agents in our design, as detailed next.

We experimentally determined how frequently each agent should act, based on overhead, impact on our target objectives, and the number of parameter values to be explored as it is desirable that all agents finish the exploration phase at the same time. For our setup,

AG_{QP} acts every 24 frames. With one frame as the offset, AG_{thread} takes action every 12 frames. AG_{dvfs} takes action every 6 frames with an offset of 2 frames. Since AG_{dvfs} and AG_{thread} act after AG_{QP}, they can modify the output throughput if it is degraded (or above the required constraints) because of AG_{QP} taking an action to increase (decrease) the video quality. In addition, as AG_{dvfs} takes actions more frequently, it can take charge of small content variations and tune the throughput to the desired FPS. Also, not having different agents acting in the same frame makes easier for the system to determine and isolate the real effects each action has on the system. Figure 6.4 shows the proposed sequence for the agents, and Table 6.1 summarizes this information.

6.2.2. New learning rate function

Since each agent acts at a different frequency, the speed of the learning process of each agent varies. In addition, although all the agents have the same states to explore, the number of actions differs, and so does the size of the exploration space of each agent, and therefore, the number of pairs it has to explore before obtaining the final policy. Thus, each agent must have its own learning rate for each state/action pair. The proposed learning rate function is a decreasing function of the number of state-action observations, differently from those proposed by the literature [97, 96, 105]. The reason is that if a learning rate function similar to the literature is considered, it is likely that an agent claims the end of the exploration phase even if other agents have not taken enough different actions. This issue ultimately makes one or more agents behave sub-optimally. Alternatively, we use the following learning rate function for each agent, AG_i, which allows each agent to monitor the number and variety of actions taken by other agents:

$$\alpha^{(i)}(s_t, a_t^i) = \frac{\beta_i}{Num(s_t, a_t^i)} + \frac{\beta_i'}{1 + \sum_{j \neq i} \left(\min_{a \in \mathcal{A}_j} (Num(a)) \right)} \tag{6.3}$$

Here, the first term is taken from the literature [105] and it is the one used in the mono-agent formulation. This term represent the learning progress of the agent on its own, while the second term represents the learning progress of the other agents.
Indeed, in the second term, $Num(a)$ is the number of times agent AG_j has taken action $a \in \mathcal{A}_j$. Then, $\min_{a \in \mathcal{A}_j}(Num(a))$ gives the minimum number of times that all actions available to AG_j have been selected. Subsequently, constants β_i and $\beta_i^{'}$ need to be set such that the exploration phase for (s_t, a_t^i) cannot finish until the following two conditions are satisfied: (I) (s_t, a_t^i) is observed so many times that $\frac{\beta_i}{Num(s_t, a_t^i)}$ can drop below a threshold, and (II) other agents have tried all their actions (at least once).

Now, let us assume two extreme cases. The first one occurs if AG_i observes (s_t, a_t) a large number of times such that $\frac{\beta_i}{Num(s_t, a_t^i)} \longrightarrow 0$, and not all other agents tried all their actions at least once. Then, $\alpha^{(i)}(s_t, a_t^i) = \beta_i^{'}$ implies that $\alpha_{th} < \beta_i^{'}$ must hold. In the second extreme case, (s_t, a_t) is observed once while all other agents have tried all their actions so many times that $\sum_{j \neq i}(\min_{a \in \mathcal{A}_j}(Num(a))) \longrightarrow \infty$. Consequently, $\alpha^{(i)}(s_t, a_t^i) = \beta_i$ and this implies that $\alpha_{th} < \beta_i$ must hold.

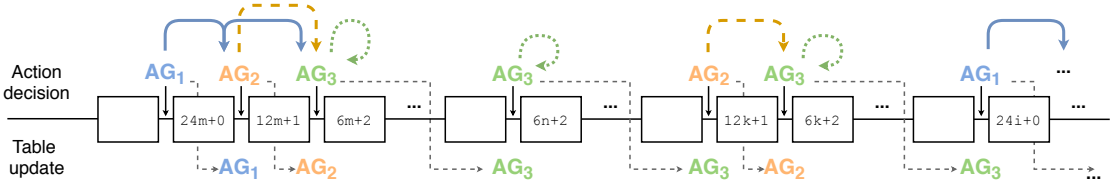


Figure 6.4: Agent sequence. Colored arrows show the cooperation between agents to take an action.

Due to the different frequencies at which each agent takes an action, in addition to the different sizes of the sub-spaces each agent needs to explore, the learning rate parameters can vary from one agent to the other. However, in this work we experimentally set $\beta_i = 0.3$ and $\beta_i' = 0.2$, $\alpha_{th1} = 0.1$ and $\alpha_{th2} = 0.05$, and $\gamma = 0.6$, which is the discount factor and controls the significance of the history of Q-values vs. recently obtained rewards.

6.2.3. Cooperation process: dealing with a stochastic environment

Since each agent has its own action set, we let the agents explore only its own actions. As we need to deal with a stochastic environment, applying action a_t^i by AG_i at state s_t may not always result in a particular s_{t+1}. The reason lies in the fact that: (I) contents of a video can change from one frame to another, (II) other agents taking actions for a specific video may apply an action that alters the next expected state to a different one, and (III) other videos existing in a multi-user platform with their corresponding contents and agents can change the state unexpectedly. Thus, once a_t^i is taken at state s_t, all state transitions to new states need to be recorded during the exploration phase. Assume that $Num(s_t \xrightarrow{a_t^i} s_{t+1})$ shows the number of times that applying a_t^i at s_t resulted in s_{t+1}, and $Num(s_t, a_t^i)$ represents total number of times that a_t^i was taken at state s_t. Then, the probability by which, after taking a_t^i at s_t, the agent observes s_{t+1} is $P(s_t \xrightarrow{a_t^i} s_{t+1}) = Num(s_t \xrightarrow{a_t^i} s_{t+1})/Num(s_t, a_t^i)$. This probability is updated throughout the learning process, and used to choose the best action cooperatively as described next. This process is equivalent to building the set of probabilities $\mathcal{P}$ referenced in the MDP formulation.

Similarly to the mono-agent approach described before, the learning process of the different agents follows a three-phase approach; however, while in *exploration* phase the agents still explore the different actions randomly, when moving to the following phases they start to cooperate.

Although each agent explores the design space separately and has its Q-table, it needs to act in the exploitation phase cooperatively. Consequently, the goal of each agent is not just to maximize the Q-value attainable from its Q-table, but rather, maximizing the *expected Q-value* after a sequence of actions taken by all agents. Consider, for example, the sequence of agents shown as in Figure 6.4 representing our particular scenario. The first agent, AG_1, is followed by two different agents, AG_2 and AG_3 in consecutive frames. Thus,

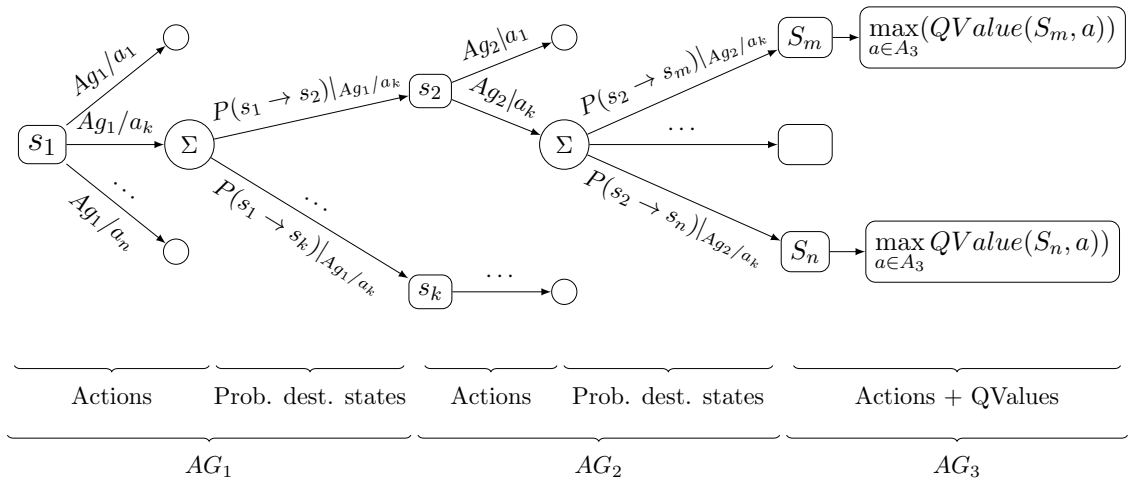


Figure 6.5: Cooperative decision process. Each path in the tree represents a possible transition of the system.

the action taken by AG_1 should consider the probable transitions from one state to the other throughout the entire chain, composed of these three agents, in order to maximize the Q-value. Indeed, AG_1 should select an action which ultimately moves the entire system to a state in which an action taken by AG_3 is capable of providing the highest Q-value. This is equivalent to considering the *expected Q-value* given that a particular action is selected by AG_1. Hence, the conditional expected Q-values should be computed for all available actions in the current state s_t, in the chain of $AG_1 \rightarrow AG_2 \rightarrow AG_3$ in Figure 6.5.

Algorithm 6.1: Decision-making process: exploitation phase.

Input : Q^i, $P(s_t \xrightarrow{a_t^i} s_{t+1})$, $\mathbf{A}$; `// ` $i \in \{1, \ldots, N\}$

Output: a_t^{i*} ; `// current action taken by the` i^{th} `agent`

1 $a_t^{i*} \leftarrow \arg\max_{a \in A_i^*} \left(\sum P(s_t \xrightarrow{a} s'_{t+1}) \times \boldsymbol{E}[Q_{Value}(AG_i.next(), s'_{t+1})] \right)$

2 **Function** $\boldsymbol{E}[Q_{Value}(AG, s)]$ **is:** `// list of agents, state`

3 **if** *(AG.next() == NULL)* **then**

4 **return** $\max_{a \in A_{AG}^*} \left(Q^{AG}(s, a) \right)$

5 **else**

6 **return** $\left(\max_{a \in A_{AG}^*} \left(\sum_P P^{AG}(s \xrightarrow{a} s') \times \boldsymbol{E}[Q_{Value}(AG.next(), s')] \right) \right)$

7 **end**

8 **end**

To calculate this expected Q-value, a recursive strategy is proposed in Algorithm 6.1. If there is not any other agent acting in the next frame (line 3), the expected *Q-value* for that agent at state s is determined by its own Q-values recorded. If another agent is acting

next, the expected Q-values are not the ones from the agent, but the ones stored in the *Q-table* of the next agent. In addition, because we are modelling a stochastic environment, each *Q-value* is adjusted by the recorded probability of each transition (line 6). The action chosen by the agents is the one which maximizes the expected *Q-value* (line 1).

6.2.4. Dealing with sensing noise

The previous definition of learning rate, as well as the three-phases learning process, guarantee a uniform exploration of all the actions of a specific state. However, it does not guarantee a uniform exploration of all the states, being possible to produce sub-optimal policies, or even non-terminating learning processes.

This problem arises especially when dealing with a non-deterministic problem, where it is common that not all the states are visited uniformly. This fact can be caused by two different reasons: (I) because of the definition of the states, or (II) because of wrong/noisy measurements from the system. A bad definition of the states or actions can affect the learning process not only in the learning time, but also in the quality of the policy. For instance, if a state definition is too strict and the interval is too narrow, it is less probable to visit this state than to visit other larger states. The second problem (noisy measurements) is a common problem and affects different metrics. For example, in a real system, power consumption measurements can be affected by other running processes, or precision problems can arise when measuring the frame processing time.

The first problem can be addressed by redefining the state definitions; the second problem cannot always be fixed, leading to the need to adapt the learning process to deal with this issue. In the mono-agent approach described in Section 6.1, this behaviour has no impact on the learning process as each pair *state/action* can evolve independently of the other pairs. However, in the multi-agent approach, the learning rate definition depends not only on the number of times one pair state/action has been visited, but also on the minimum number of times another agent has visited each pair state/action (second term of Equation 6.3). This definition affects not only the learned policy, but also the evolution of learning phases, making the worst case stall the system on the exploration phase, not allowing to progress to the next phases. To tackle this problem, we propose the use of a slightly modified version of the classical Z-score algorithm [167, 126] to detect outliers [81]. Our proposal, shown in Equation 6.4, identifies those *state/action* pairs that have not been visited enough times compared with the other pairs, but still to consider the states that have been visited more times than the others (this states usually will be detected as outliers too by classical algorithms). To identify those pairs, the classical Z-score formulation has been adapted (z-score$'^{(i)}(s_t)$) to consider the standard deviation (σ') of only the pairs that are below the average ($\mathbb{S}^i$), and not of the all pairs.
When the learning rate function has to be calculated, the states detected as *bottom-outliers* are discarded to compute on the function. However, these states are not removed from the Q-tables since it is possible that they will be visited in the future enough number of times to count again in the learning function. Experimentally, we have determined that discarding states with z-score$' \leq -3.5$ achieves the expected results.

$$\text{z-score}'^{(i)}(s_t) = \frac{\sum_{a_t^i} Num(s_t, a_t^i) - \mu^{(i)}}{\sigma'^{(i)}} \tag{6.4}$$

where:

$$\mu^{(i)} = \frac{\sum_{s_t, a_t^i} Num(s_t, a_t^{(i)})}{|\mathcal{S}|}, \qquad \mathbb{S}^i = \left\{ s_t \in \mathcal{S} \;\middle|\; \sum_{a_t^i} Num(s_t, a_t^{(i)}) \leq \mu^{(i)} \right\}$$

$$\mu'^{(i)} = \frac{\sum_{\hat{s}_t \in \mathbb{S}, a_t^i} Num(\hat{s}_t, a_t^{(i)})}{|\mathbb{S}^{(i)}|}, \qquad \sigma' = \sqrt{\frac{\sum\limits_{\hat{s}_t \in \mathbb{S}} \left(\sum_{a_t^i} Num(\hat{s}_t, a_t^i) - \mu'^{(i)} \right)^2}{|\mathbb{S}^{(i)}|}}$$

6.3. Proposed single-application scenarios and experimental setup

In this section, we experimentally evaluate a real implementation of the previously described MAL system on a modern multi-core server, MAKALU (see Section A.1). In the following, we report output metrics (application- and system-wide) both in terms of efficient use of the resources available in the platform, and in terms of the QoS obtained in the different encoded sequences.

In order to validate our approach in a sufficiently wide and representative spectrum of scenarios, we carry out all the experiments with a variety of videos covering realistic situations, as well as with different resolutions, ranging from *Low Resolution* videos –usually played in portable devices like mobile phones or tablets– to *High Resolution* videos –in this case, usually meant to be visualized in computers with a stable and high-bandwidth Internet connection–.

Additionally, we compare our proposal to two alternative strategies: a STATIC approach, and a state-of-the-art heuristic-based strategy (ARGO [66, 67]). Our results reveal better QoS and a comparable exploitation of the available resources comparing with both strategies, at the expense of a more flexible and automated policy extraction process.

6.3.1. System overview and implementation details

Figure 6.6 depicts a general diagram of our formulation, and shows how actions are chosen and applied. The system is continuously sensing the environment and receiving different (application- and system-wide) metric values frame to-frame basis, storing and processing them. Upon each agent activation, the state is built from the current and stored metrics, discretizating the continuous values according to the states defined in Section 6.1 (step 1 in the diagram). This state is used to update the *Q-table* of the previous agent (step 2, equation 1), and to determine the action of the next agent (step 3). As the learning rate is defined for each *state/action* pair, different pairs can belong to different learning phases –exploration, exploration-exploitation or exploitation– for the same state. In step 4, the system determines the phase of the next action and then which action belonging to that phase applies (step 5), explained in Algorithm 5.1. Finally, the system applies the chosen action in step 6.

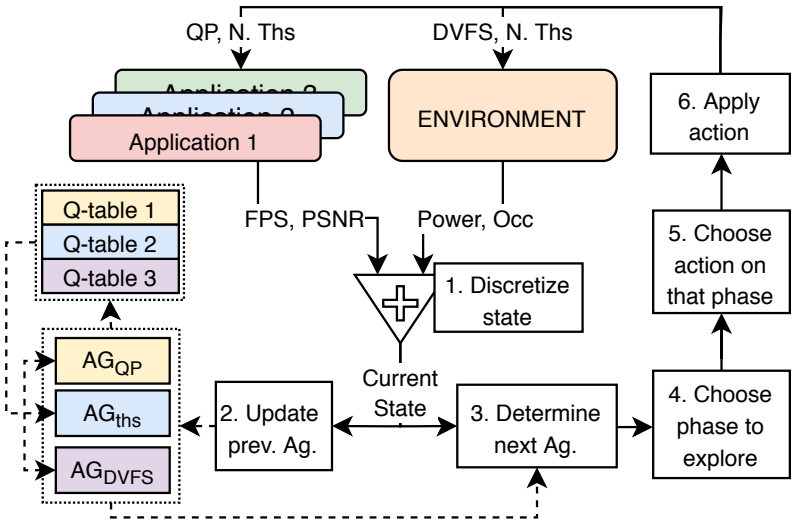


Figure 6.6: General system overview.

Similar to the heuristic-based resource management techniques described previously in Chapters 3 and 4, all the tested approaches explored here have been implemented and integrated into our centralized Resource Manager (see Appendix B). The main work of the resource manager is to integrate the multi-agent Q-Learning logic as well as the other tested approaches, and to interact with the encoding application. Specifically, the resource manager is in charge of:

1. *Receiving* the application-specific metrics on a regular basis (e.g., PSNR measurements or instantaneous FPS among others) from the encoding process. Also, the Resource Manager stores this information and processes it accordingly to its nature. For example, FPS measurements are processed through a moving window due to the great variability on the data.

2. Periodically *sensing*, storing and processing the underlying platform metrics (e.g., power consumption, thread affinity, or core occupation).

3. *Managing* all the underlying shared resources performing an efficient allocation to the application. In the case of an unique application running on the system, it is responsible of setting the proper core affinity to avoid a high number of thread migrations between cores, and to set the proper frequency only to those cores used by the application at each moment.

4. *Deciding* the most appropriate knob(s) to be applied to the encoding application based on the application metrics received and the running approach (i.e., MAL, STATIC or ARGO approach). In the case of the MAL approach, the discretization of the metrics according to the state definitions detailed in Section 6.1 is also a task for the centralized Resource Manager.

A detailed description of the actual implementation of the system and communication mechanisms used can be found in Appendix B. The Kvazaar encoder was slightly modified

	Video Id	Video Name	μ FPS	σ FPS	min FPS	max FPS
Training	HR1	*FourPeople*	30.6	6.5	11.2	47.6
	HR2	*KristenAndSara*	30.2	4.1	12.6	40.3
	HR3	*OldTownCross*	15.4	1.5	8.6	18.2
Test	HR4	*QuarterBackSneak1*	26.1	2.9	14.5	38.5
	HR5	*BT709Parakeets*	29.1	5.1	11.3	49.7
	HR6	*Johnny*	31.7	5.7	13.0	44.2
	HR7	*ThreePeople*	29.7	6.3	9.5	44.7

Table 6.2: Video characterization using static encoding resources: 3 threads, 1.5 GHz, QP=22. The corresponding *Id* prefixed by LR denotes low resolution versions of each video.

to support dynamic knob tuning and integration into the resource manager. In total, 37 new lines were added to the code base, where 5 of them are responsible of communicating with the centralized resource manager (to inform about the start and end of the execution, to send the metrics measured at each frame, and to ask for the QP value and the number of threads to use at each frame), while the others are responsible of adapting the encoding process to the dynamic knob values received from the centralized resource manager.

6.3.2. Dataset definition

We consider videos with two very different resolutions: Low Resolution (LR) videos, which is the default resolution provided by YouTube (832 × 480 pixels), and High Resolution (HR) videos which is the resolution considered as "High Definition" (720p/HD videos with resolution 1280 × 720). HR videos have been extracted mainly from the test sequences proposed by the JCT-VC [28]; LR videos used are the same as their HR counterparts, re-scaled to the proper resolution.

To characterize and classify the different videos, each sequence is encoded using the same resources during the complete execution (3 threads at 1.5 GHz with QP=22, i.e. maximum quality), measuring instantaneous FPS at the end of each frame. Table 6.2 shows the average FPS measured in each video (μ FPS), as well as the standard deviation of the instantaneous FPS measurements (σ FPS), and the maximum and minimum FPS recorded during the whole encoding process (max FPS and min FPS respectively). As shown, the chosen sequences cover a wide range of scenarios. On the one hand, videos like HR6 or HR1 produce high variability on the obtained FPS, but with low resource requirements (30.6 ± 6.5 FPS with fixed resources). On the other hand, HR3 or HR4 are sequences with low variability but higher resource demands (26.1 ± 2.9 FPS or 15.4 ± 1.5 FPS when encoded with the same knob values).
In our experiments, we divide the sequences between training and test, incorporating both types of videos in both groups. The training set (used to train the MAL system and fill the Q-tables), is comprised of a sequence with high variability but low resources demands (HR1), a low demanding sequence but with lower variability (HR2), and the highest demanding

	Action freq.	Freq. (GHz)		N. Ths		QP	
		HR	LR	HR	LR	HR	LR
MAL							
AG_{dvfs}	6	$\{1.0, \cdots, 2.0\}$		—		—	
AG_{thread}	12	—		1 – 5	1 – 3	—	
AG_{QP}	24	—		—		{22, 27, 32, 37}	
STATIC	—	1.5	1.4	3	3	32	22
ARGO	6	*ondemand*		1 – 5		{22, 27, 32, 37}	

Table 6.3: Activation frequency and actions considered by each tested approach. Notice how the number of threads varies depending on the resolution of the video as the amount of parallelism is limited by the resolution. ARGO approach has been tested only for HR videos.

sequence with low variability (HR3). Similarly, the test group is also formed by videos covering all the spectrum.

Training the system with a mix of videos (instead of a different training process per type) allows us to encode any video in the future without needing to determine its type before starting the encoding (and therefore which policy among those learned should be applied). The main problem of this approach is that different videos can introduce a considerable amount of noise into the learning process: if the system applies an action in a specific state, the destination state can differ depending on the video being encoded. However, a proper state and reward definitions together with the previously presented ideas can mitigate this problem.

6.3.3. Alternative approaches and reported metrics

For the sake of fairness, we propose a comparison between our proposal and two alternative strategies:

MAL approach: This is our proposal. The selection of each knob at runtime is carried out by the different agents which form our system (AG_{dvfs}, AG_{thread} and AG_{QP}). Each agent has a different frequency in which they take an action (based on the impact on the output metrics). Depending on the resolution, the amount of threads available to the encoding process varies due to the exposed parallelism in Kvazaar, which is limited by the video resolution. Table 6.3 shows the configuration options set in the experiments.

To train our system, different randomly selected videos of the training set are encoded one by one through the MAL system at the same time it fills its internal tables. This process is repeated until all the state/action pairs are in exploitation phase. As explained before, our training set comprises 2 videos with low resource requirements, and 1 video with low variability but a higher resource requirements. Targeting a correct learning process, the random selection of the video to be encoded at each time is performed with probabilities 1/4, 1/4 and 2/4, respectively. In this way, both types

of videos are explored the same number of times, and therefore, the obtained policy is valid for a wider variety of videos.

STATIC **approach:** In this approach, knob values are fixed to a constant value during the whole execution. For the sake of fairness, knob values in STATIC correspond to the average value learned by the MAL system for the training videos. For High Resolution videos, the sequences are encoded using 3 threads at 1.5 GHz with a QP value of 32. For Low Resolution videos, 3 threads at 1.4 GHz are used and a QP value of 22.

ARGO **approach:** This is the heuristic approach described in [66, 67] adapted to our scenario. ARGO bases its decisions on an internal database storing all feasible knob configurations called *Operating Points (OPs)*, and an estimation of the output metrics obtained for each OP. ARGO maintains a set of constraints that the system should never violate. At runtime it chooses, among all the promising OPs that do not violate the constraints, the configuration that maximizes a user-defined rank function. To provide self-adaptation, ARGO computes and modifies on the fly different coefficients to relate the obtained measurements to those expected and stored.

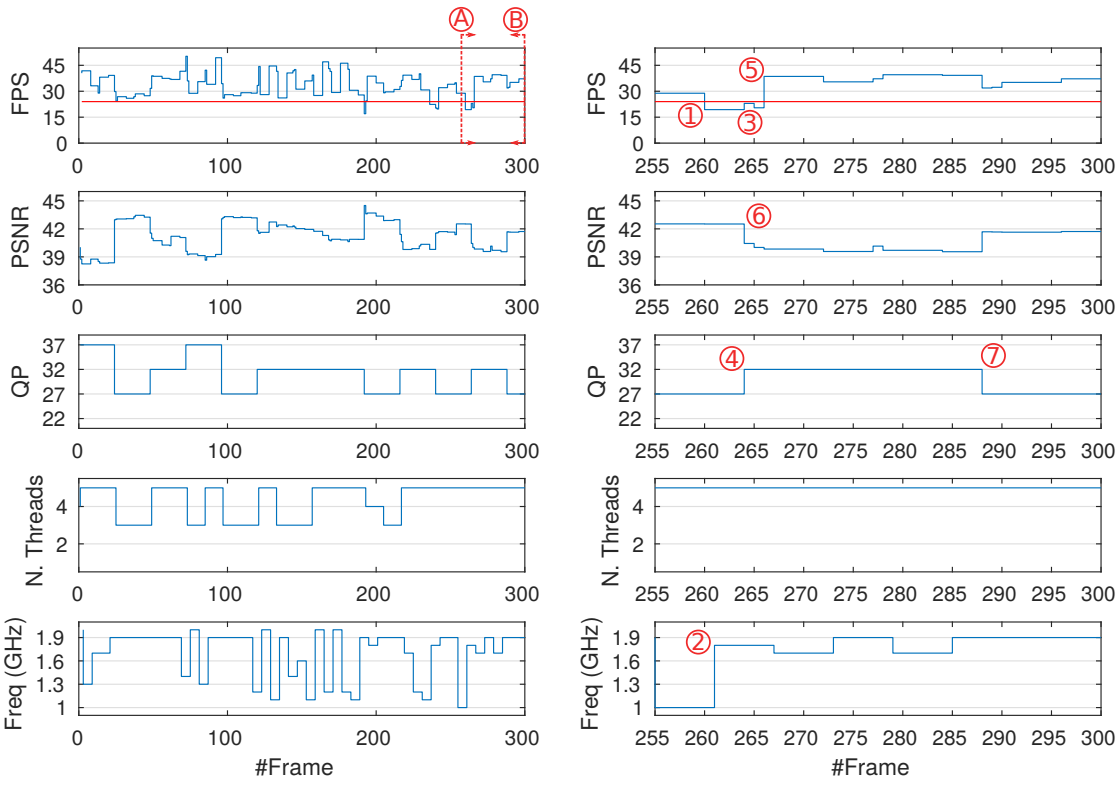
To compare our proposal with ARGO, we generate its internal database by profiling the same training videos used in our system; we set QP and number of threads as the dynamic knobs managed by the heuristic, delegating the frequency adjustment to the *ondemand* governor of the OS. To make the comparison as fair as possible, the heuristic acts every 6 frames (matching the minimum frequency of the agents in our approach), and the rank function used to evaluate OPs in ARGO matches the reward function used in the MAL approach. Without loss of generality, ARGO is compared only when encoding the most demanding resolution (HR).

Table 6.3 summarizes the aforementioned approaches and the configuration decisions made on each one. In the following, all the reported data correspond to the average value of five different executions. Results are reported in terms of QoS metrics measured from the application and resource usage metrics measured from the centralized resource manager. QoS metrics are reported in terms of: (I) QoS violations (i.e., percentage of the time the video is encoded below the predefined real-time threshold, represented as -Δ-), and (II) attained quality measured in PSNR (dB). Note that dB are measured in a logarithmic scale, so small variations in the numeric values entails big changes in the quality. Resource usage metrics are reported in terms of: (I) average number of threads used, (II) average QP value set, and (III) average frequency used during the whole encoding process.

6.4. Experimental results

6.4.1. A detailed analysis of agents' behavior for HR video sequences

Consider the execution trace represented in Figure 6.7. The plots on the left represent the complete encoding process of a HR4 sequence (*QuarterBackSneak1*). The first two plots on the top report output metrics of the applications (i.e., instantaneous FPS and PSNR measured frame by frame), while the remaining three report the decisions taken by the system in terms of knob modifications and the resources used at each frame.



(a) Full HR4 encoding trace.

(b) Detailed view of the marked zone.

Figure 6.7: A trace representing an encoding process of HR4 by the MAL system. On the left, the full trace of the whole sequence. On the right, a zoom in view of the marked zone. In both figures, the first two plots on the top represent the QoS output metrics of the application (FPS and PSNR), while the last three represent the resource usage metrics of the system (QP set, number of threads used and frequency utilized). Points marked in the Figure are explained in the text.

Focusing on the measured FPS, we can observe how the MAL system is able to encode most of the video above the real-time constraint set in this experiment (24 FPS represented by the red line in the figure), and how, when the instantaneous FPS drops under the threshold, the system recovers the required FPS again with a delay of only two frames. In addition, the PSNR measurements reveal how MAL is able to maintain the quality between the required limits (PSNR ranging from 30 dB to 50 dB).

The remaining three plots show the actual modifications carried out in knobs to adapt to video contents, and give an overview of the general strategy autonomously learned by the agents: varying system frequency more often than the number of threads, and modifying number of threads more often than QP. This strategy does actually have an explanation, and an *intelligent* behavior can be envisioned in the agents' behaviour. As introduced previously in this chapter, one-step frequency modification has a relatively small impact on the encoding process, exposing a finer-grained control over throughput.

If the system needs to perform larger changes in the encoding process (e.g., due to large changes in video contents), it will first modify the number of threads or QP. Next, the system will change the frequency to tune the execution in a finer-grained way. Observe how the main changes in FPS and PSNR are caused by changes in the QP value. This high-level behaviour exhibiting the impact of each knob (in the order of frequency, number of threads and QP) has been extracted automatically during the learning process, and is one of the main benefits of the RL-based approach.

Plots on the right side show a zoomed-in version of the trace enclosed between marks Ⓐ and Ⓑ on the left. In these plots, different points have been marked (circled numbers in red) explaining with more detail the policy learned by our system to encode a HR sequence. Although the actions taken by the system varies between executions (even for the same sequence) due to the stochastic nature of the system, the overall behaviour has been observed to be similar in other executions of the same sequence, as well as for the other tested sequences. Later, we will discuss how the system learns a different policy for LR videos, enhancing the use of a RL-based approach instead of a more traditional heuristic approach.

Summarizing, the different decisions taken by the system in the different points can be explained as follows:

① At this point, MAL observes that the instantaneous FPS are under the real-time requirements. Because MAL is composed by three different agents acting at different frequencies, the system will not take any action until the next frame in which one of the agents acts. In the worst case, the next agent will not act until the next 6 frames are processed.

② In this example, the first agent to act is AG_{dvfs}, which increases the frequency seeking to achieve real-time encoding process again.

③ The increase in frequency has positive impact on the instantaneous FPS. However the obtained FPS rate is still below the threshold, so other agent will need to take a trade-off action in the following frames.

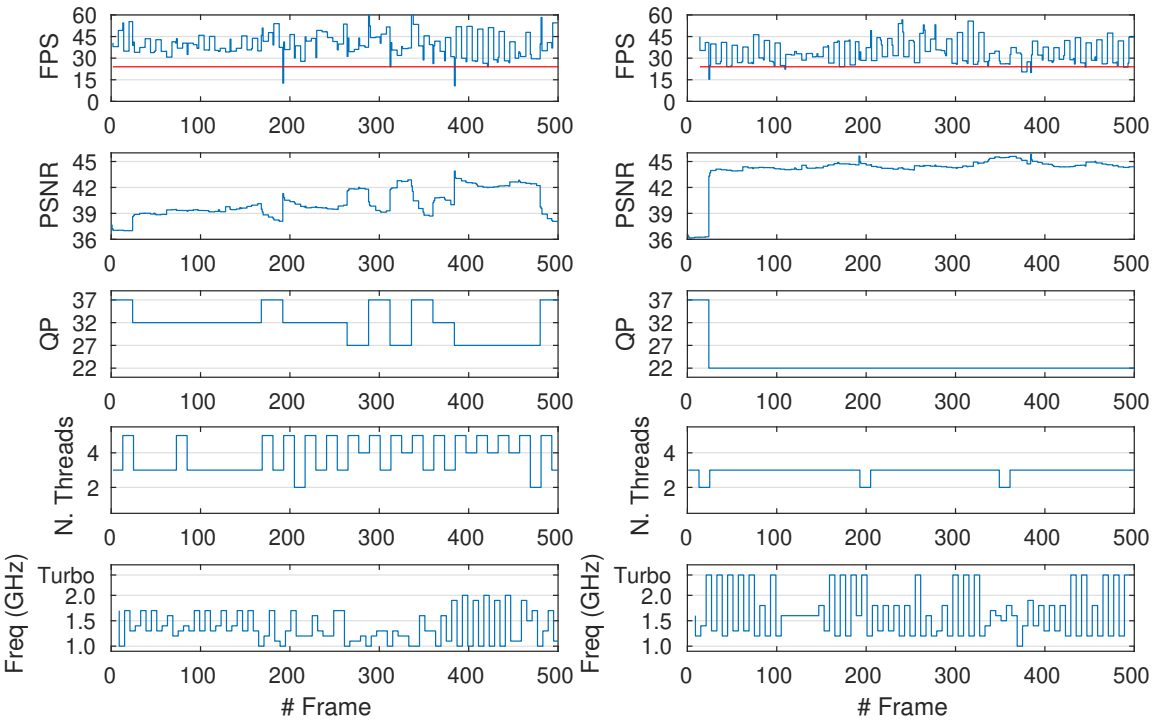
④ As the number of threads cannot be increased, the AG_{QP} decides to increase the QP one step. Note that this change is performed as the agent has learned not only that increasing the QP entails an increase in FPS, but also that the resulting PSNR will fall between the required limits.

⑤ As explained before, the increase in the QP value has a huge impact in the FPS, encoding again the sequence on real-time.

⑥ Observe how the change in QP also has a huge impact on the quality of the processed frame, but the PSNR is always maintained above the limit (30 dB).

⑦ When possible, the system will try to modify the QP value again to maximize the quality, always subjected to the restriction that the new knob configuration will encode the video at $\geq$ 24FPS.

6.4.2. General MAL learned policies: HR vs LR behaviour



(a) High Resolution version (HR5). **(b)** Low Resolution version (LR5).

Figure 6.8: MAL behaviour when encoding the same video (HR5/LR5 – *Parakeets*) in both resolutions. Observe how the learned policy is different in both sequences.

Figure 6.8 corresponds to the encoding of the same video (*Parakeets*) by the MAL system in two different resolutions: High Resolution on the left and Low Resolution on the right. The behaviour when encoding the highest resolution is consistent with the one previously described: sporadic changes in QP trying to increase the quality when possible, or trying not to encode the sequence below 24 FPS when there is no other knob configuration possible; and more frequent changes in the number of threads (hence, applying a finer control on the output metrics) and a constant adjustment of the frequency producing the finest tuning of the output metrics.

In the case of Low Resolution sequences, the behaviour of the system is slightly different to the previous one (but consistent among all the videos of the same resolution). Through the learning process, MAL has learned that LR videos need lower amount of resources to be encoded than HR videos, and therefore, it can increase the quality without increasing the amount of violations of the FPS constraints. Indeed, the system is able to encode the videos at the maximum quality (minimum QP) while adjusting the FPS metric by tuning the number of threads and frequency. This is possible thanks to the use of *Turbo* frequency. In the case of LR videos, the amount of threads used to codify a sequence is limited up to 3, meaning that $S_{occ} \in \{1, 2\}$, and therefore, the effective frequency will be high enough to encode the sequence (3.6 GHz and 3.4 GHz) at maximum quality. However, in Chapter 7 will

		Output metrics				*Avg. Knob values*		
		–Δ–		**PSNR** (dB)		**Freq**	**N. Ths**	**QP**
		MAL	STATIC	MAL	STATIC	MAL		
HR	1×HR4	7.8	14.6	41.1	40.8	1.6	4.0	31.1
	1×HR5	0.9	0.4	40.2	39.9	1.4	3.8	31.4
	1×HR6	1.0	0.4	40.7	40.1	1.2	3.6	31.3
	1×HR7	0.5	0.4	39.3	39.2	1.3	3.5	31.9
	1×HR *Avg.*	2.5	4.0	40.3	40.0	1.4	3.7	31.4
LR	1×LR4	27.9	71.7	44.0	44.7	1.7	2.5	23.7
	1×LR5	7.4	0.5	44.4	44.5	1.5	2.8	22.4
	1×LR6	0.9	0.4	44.8	44.8	1.3	3.0	22.0
	1×LR7	0.5	0.5	44.0	44.0	1.4	3.0	22.0
	1×LR *Avg.*	9.2	18.3	44.5	44.6	1.5	2.8	22.5

Table 6.4: Output metrics and resource usage for the MAL approach compared with the STATIC assignment for HR (top) and LR videos (bottom). STATIC approach encodes the different sequences with the average values learned by the MAL approach (i.e., 3 threads, 1.5 GHz and QP=32 for HR videos, and 3 threads, 1.4 GHz and QP=22 for LR videos).

show how, when the number of simultaneous videos increases (and therefore the effective turbo frequency decreases), the system is able to sacrifice (reduce) quality to mitigate the effects of the frequency decrease.

6.4.3. Comparison with a static approach

Table 6.4 illustrates the behaviour of our approach compared with STATIC when encoding the different *testing video* sequences in terms of the QoS metrics (percentage of the time the FPS constraint is violated -Δ-, and output quality measured through the PSNR). The resource usage of MAL (average number of threads, QP value and frequency set) is also shown in the table. These results illustrate how the MAL approach is able to restrict the attained throughput to the real time threshold. On the contrary, the percentage of time the STATIC strategy is not able to fulfill these requirements is larger even though the selected (and fixed a priori) values for each knob are those learned by our system. This is a clear sign of the benefits of the dynamic knob tuning process accompanied by an application-aware knowledge. The awareness of the status of the application at each moment allows not only to react when the constraints are violated, but to constantly tune the most suitable knobs in advance based on the predictions carried out with the current measurements and the recorded history.

From the results shown in the table, a number of insights can be extracted, namely:

- Based on the results and in the previous characterization, we can divide the sequences into two groups: HR4 sequence as the most demanding one, and HR5, HR6 and HR7 as non-demanding sequences.

HR	*Output metrics*				*Avg. Knob Values*					
	−Δ−		**PSNR** (dB)		**N. Ths**		**QP**		**Freq**	
	MAL	ARGO	MAL	ARGO	MAL	ARGO	MAL	ARGO	MAL	ARGO
1×HR4	7.8	9.6	41.1	43.6	4.0	2.7	31.1	25.0	1.6	*Turbo*
1×HR5	0.9	8.5	40.2	42.5	3.8	1.5	31.4	26.7	1.4	*Turbo*
1×HR6	1.0	9.9	40.7	42.9	3.6	1.1	31.3	25.2	1.2	*Turbo*
1×HR7	0.5	7.0	39.3	41.7	3.5	1.0	31.9	26.4	1.3	*Turbo*
1×HR *Avg.*	2.5	8.8	40.3	42.7	3.7	1.6	31.4	25.8	1.4	*Turbo*

Table 6.5: MAL compared to the ARGO approach. With comparison purposes, MAL results are reported again in this table.

- For the non-demanding sequences, both approaches behave correctly, obtaining similar results in terms of QoS metrics. In terms of resources, MAL requires more threads than STATIC (3 threads), but the frequency set in average is lower (1.5 GHz in STATIC).
- For the most demanding video, the MAL system has learned to increase the assigned resources (number of threads when dealing with HR sequences, and frequency when encoding LR videos). This is possible thanks to the dynamic knob adaptation and application-aware knowledge available in MAL, but not in the the STATIC strategy.
- MAL reduces the number of QoS violations when compared with the STATIC approach, yielding 1.6× and 2× better results for the HR and LR sequences respectively. In addition, the fast reaction of the MAL system when the throughput constraint is violated can favor the use of buffering techniques to serve the video to the final users in real time, avoiding delays in the stream.

To sum up, the dynamic knob tuning process, together with a constant knowledge of the internal status of the application allows a more precise tuning of the different knobs, and therefore, to obtain higher-quality executions in terms of any output metric.

6.4.4. Comparison with a state-of-the-art heuristic (ARGO)

Table 6.5 shows the results obtained when encoding the different HR videos under ARGO approach. For the sake of clarity, we report the MAL results again in this table.

At a glance, although ARGO obtains higher PSNR in all the tested scenarios, the amount of time the system violates the throughput constraint (-Δ-) is systematically larger than in our proposal. This behavior is explained in terms of the changes in the content of the video. While our proposal is able to perform a fine-tuning process by adjusting the processors' frequency, ARGO delegates this to the governor of the operating system. This delegation results on setting the frequency at turbo in all scenarios, thus, ARGO needs to increment the number of threads to compensate a violation in the QoS constraint. However, as explained before, although a change in the number of threads can have a huge impact on the instantaneous throughput, the consequences on the output metrics are rougher compared with

the expected consequences when changing the frequency. The later is the policy followed by the MAL approach.

In addition, in the case of videos with high requirements (like HR4), ARGO increments the number of threads to increase the instantaneous throughput (up to 2.7 threads when the average is 1.6). On the contrary, our approach has learned to decrease the quality in these scenarios, having more room to change the frequency and number of threads when needed, and therefore, to avoid violations in the throughput constraint.

6.5. Conclusions

In this chapter, we have shown how a resource management process for online HEVC video encoding can be formulated in terms of a Q-Learning algorithm. A generic state, action and reward decomposition was presented to easily map a generic *malleable* and *QoS-aware* application, allowing a fastest learning process than a mono-agent formulation and an easy tuning of the rewards ad states. However, although the mono-agent approach is perfectly valid for most scenarios, in complex scenarios where the number of states and actions increases, the time required for learning the system makes this formulation almost unfeasible.

A co-operative multi-agent approach solves this problem by splitting the actions between different agents, and therefore, reducing the exploration space. A new learning rate function and activation sequence for the agents were proposed to allow the agents to learn at different speeds and still obtain high-quality policies. In addition, the decision-making process was reformulated to incorporate the cooperative behaviour when the system runs on a stochastic scenario. Lastly, we provided the system formulation with an efficient mechanism to detect an ignore noisy measurements coming from the application or environment metrics measurements.

From the experimental perspective, our formulation was tested on a real platform encoding different sequences of two different resolutions, reporting the behaviour of our approach when comparing to the STATIC and ARGO approaches.

The results demonstrate how our multi-agent approach is able to adapt to changes in video contents and at the same time it is able to fulfill the throughput requirement and maximize the quality. The results also show how the system is able to learn different policies for the different resolution tested, and therefore, it proves that the system is able to make a decision based on the running applications.

When comparing our system with the STATIC approach, the results show how being aware of the application internals achieves remarkable improvements for HR and LR resolutions, respectively. Also, our proposal reduces the number of times the THROUGHPUT requirement is violated compared with ARGO, as it has learned to sacrifice quality to guarantee encoding is above 24 FPS, contrary to ARGO, which tries to increase the throughput increasing the number of threads, but still obtaining worst results than MAL.

To sum up, the main contribution of the chapter can be summarized as the demonstration of the feasibility of porting a specific application into a Q-Learning formulation (both mono- and multi-agent), and how a classical formulation can be modified to be applied into

a real system with a huge stochastic exploration space and noisy measurements. The presented formulation has been tested in a real scenario, and is generic enough to be mapped to other applications, both in the multimedia area and in other fields.

However, the contributions and experiments in the chapter are limited to one concurrent application scenarios. The next chapter extends the formulation to consider multi-application environments.

7

Extensions for Inter-Application Resource Management

The main contribution of Chapter 6 was the design and implementation of a ML approach (specifically a QL approach) that is able to efficiently manage all the *intra-application* metric interactions by dynamically tuning the proper knobs (both at application and system level). In particular, up to this point, we have shown how to model the design space in terms of states and substates, actions and reward functions to achieve high-quality real-time codification processes.

This chapter goes one step further, and proposes an extension of the previous scenario that incorporates the management of *inter-application* interactions. Hence, our target scenarios (both theoretical and practical) are shifted from single-application management to *multi-application* management.

The main objective of the chapter is to illustrate how a multi-agent QL resource management approach can be extended to integrate multi-application scenarios, and to discuss the necessary modifications in the baseline model to accommodate this new functionality. Being aware of all the running applications, a resource manager can determine the best knob configuration for each application, satisfying individual per-application constraints, and optimizing global metrics at the same time.

Following with our motivational application, our framework is developed and deployed considering a multi-core server in which a number of transcoding requests from users arrive at random points in time, each one of a different video type (resolution and contents), and hence, heterogeneous resource requirements. In addition, the previous application-specific constraints (performance/THROUGHPUT and quality/PSNR), are extended to consider a system-wide constraint: *power capping*.

However, to handle new intra-application dependencies, the basic formulation in Chapter 7 requires a number of modifications and extensions. In particular, this extended formulation needs to integrate a number of new features, among which we can highlight: (I) redefining the states and rewards to consider the new global metrics, (II) properly managing the shared resources to avoid overuse (in our scenario, manage the thread affinity to

avoid oversubscription scenarios), and (III) defining a proper method to train the system to avoid the interference between multiple agents of different applications into the learning, being able to associate system-wide effects to specific local actions.

The Chapter is structured in two main parts. Section 7.1 extends the ideas of the previous Chapter to incorporate the management of inter-application dependencies into our formulation. Specifically, it describes the new metrics, states and rewards added to the system, as well as the new problems associated to the management of simultaneous applications, and their solutions. In Section 7.2 we compare our formulation with the same approaches used in the previous Chapter (STATIC and ARGO approaches), a equivalent mono-agent implementation, and RAPL, the state-of-the-art hardware-based power capping mechanism. Lastly, Section 7.3 presents a set of conclusions and final remarks.

7.1. Integrating intra-application dependencies into the formulation

The extended multi-application design is composed by different agents each associated to a different knob, similar to that presented in Chapter 6; different *Q-tables* are considered for each resolution. However, if multiple videos of the same resolution are encoded simultaneously, a set of different agents is deployed for each process, all of them sharing the same *Q-table*, and therefore, the same policy. The existence of multiple agents using the same table has two major drawbacks, namely: (I) A synchronization mechanism is needed to avoid two simultaneous modifications of the same *state/action* pair in the table. This problem occurs only in the *exploration* and *exploration/exploitation* phases; the *exploitation* phase does not require this explicit synchronization, given the read-only nature of agents' access, and (II) If a resource is shared between agents (for example, physical cores in our case), it needs to be serialized to avoid oversubscription situations.

7.1.1. A modified learning process for system-wide metrics

As previously exposed, deploying multiple agents accessing the same table during the learning process can yield race conditions, and therefore, sub-optimal or incorrect policies. There is, however a second major problem in the extended formulation, mainly related with *system-wide metrics*; a clear example in our case is power consumption. This implies that changes in power may not be directly related to the changes applied by one specific application, but can be bound with other applications adapting their knobs at runtime. This phenomena can lead to incorrect learning from the agents, updating the Q-tables based on observations that are not consequences of their own actions.

To address this problem, we propose modifications in the learning process to deal with system-wide output metrics. During the learning process, only one encoding process learns and updates the Q-tables, and multiple background process encode random sequences and provide different scenarios to the first application. The number of background processes varies over time, allowing the learning process to explore all the different scenarios. In the case of our formulation for HR sequences, the system is first trained with only one sequence until the substate $S_{occ} = 1$ is fully explored, increasing the number of videos progressively to explore the other occupation states, up to 6 simultaneous videos that guarantees the

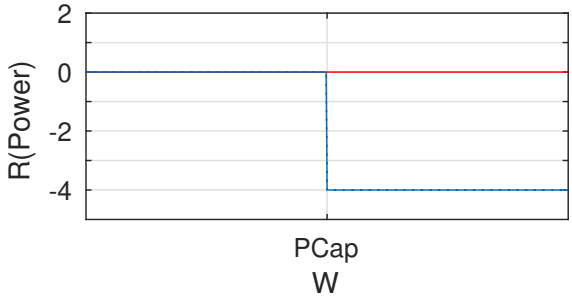


Figure 7.1: Power reward. This function does not maximize or minimize any metric, but penalizes power consumption above a specific cap by returning a negative reward of -4.0.

exploration of the $S_{occ} = 6$ substate. In the case of LR videos, the system is trained similarly, increasing the number of background sequences progressively until all states are fully explored.

Once the system has learned (and the Q-tables are not modified anymore), all processes can proceed with the encoding procedure using the same trained agents.

7.1.2. Power capping integration

To incorporate power capping abilities into our previous formulation, new state and reward functions are needed. However, thanks to our decomposition of the state definition (see Equation 6.1) and reward function (see Equation 6.2) into sub-states and sub-rewards, respectively, only the definition of a new substate and a new subreward function are required instead of the redefinition of the whole state or reward:

POWER **sub-state ($\mathcal{S}_{power}$):** As we are not considering power as a metric to maximize of minimize, but as a cap not to exceed, the POWER state ($\mathcal{S}_{power}$) is divided only into two different values: those below the cap, and those above the cap ($power < P_{cap}$ and $power \geq P_{cap}$). In case that Pcap is considered as a metric to minimize, the state space can be split into multiple sub-states, similar to S_{psnr}.

$$\mathcal{S}_{power} \equiv \quad \{\, power \in [0, P_{cap}), [P_{cap}, \infty) \,\}$$

POWER **sub-reward:** Power consumption is limited to a value defined by the server administrator (P_{cap}). The reward function will return a negative value to those states above the power cap, and no reward to the other state. Figure 7.1 illustrates the behaviour of this reward.

$$R(power) = \begin{cases} -4.0 & power \geq Pcap \\ 0.0 & \text{otherwise.} \end{cases}$$

Besides the addition of the new sub-state, encoding multiple videos simultaneously makes S_{occ} increasing from 3 different values up to 6. This implies that an agent should visit 504 different states and explore all the state/action combinations multiple times before

obtaining the final policy. In the case of a mono-agent approach, this space decomposition summarizes in 120 960 different pairs ($504\,states \times (4 \times 5 \times 12)\,actions$), which takes a considerable amount of time to be explored. Opposite to this formulation, a multi-agent approach is able to reduce the exploration time by performing a proper division of the action set (note that the states need to be equal for all agents). In our formulation, this decomposition reduces to 10 584 different pairs *state/action* ($504 \times 4 + 504 \times 5 + 504 \times 12$), yielding a reduction of $\approx 12\times$.

7.1.3. Management of shared resources

The formulation presented here is able to determine the proper value of each knob at a specific moment while satisfying the required constraints. However, if the underlying resources whose use is modified by a knob variation are shared across different applications, additional logic is needed to: (I) check if the resource is available or it has been taken by other agent, and (II) in the case the action cannot be taken (because there are not enough free resources), modify the learning process to avoid learning incorrect transitions.

In our formulation, both problems are present in the AG_{thread} agent: it is possible that the agent decides to assign to an application more threads than available free cores; and it is possible to assign the same physical cores to different applications simultaneously.

To solve the first problem (the agent requires more threads than available free cores), the agent checks after deciding which action to take, and before communicating it to the application if it is possible to apply that action. If not, the agent does not notify the application about the new knob value, and annotates this action to not update the *Q-table* in the next iteration. Additionally, to avoid oversubscription, the agent keeps internal information regarding core-to-application mapping. When this assignation needs to be modified (due to increases or decreases in the number of required threads), the agent will modify this allocation trying to reuse the same physical cores, and therefore, avoid thread migrations between cores.

7.2. Experimental results on multi-application scenarios

To test the extended abilities of our approach to manage inter-application dependencies, we have followed the same methodology used in Section 6.3. First, we show the behaviour of our MAL approach, and how the learned policy is different when dealing with different number of videos (i.e., $\mathcal{S}_{occ}$ varies). Second, we compare our proposal with the STATIC approach in two different scenarios: multiple videos of the same resolution being encoded simultaneously, and sequences of different resolution mixed. Later, we show the behaviour of our approach compared with the ARGO heuristic, and the benefits to use a multi-agent approach instead of a mono-agent approach. Finally, we show how our approach behaves in a power constrained scenario, and compare the results with RAPL, a power capping hardware-based alternative. And at the end of the chapter, we show the minimum overhead our approach introduces into a normal Kvazaar process.

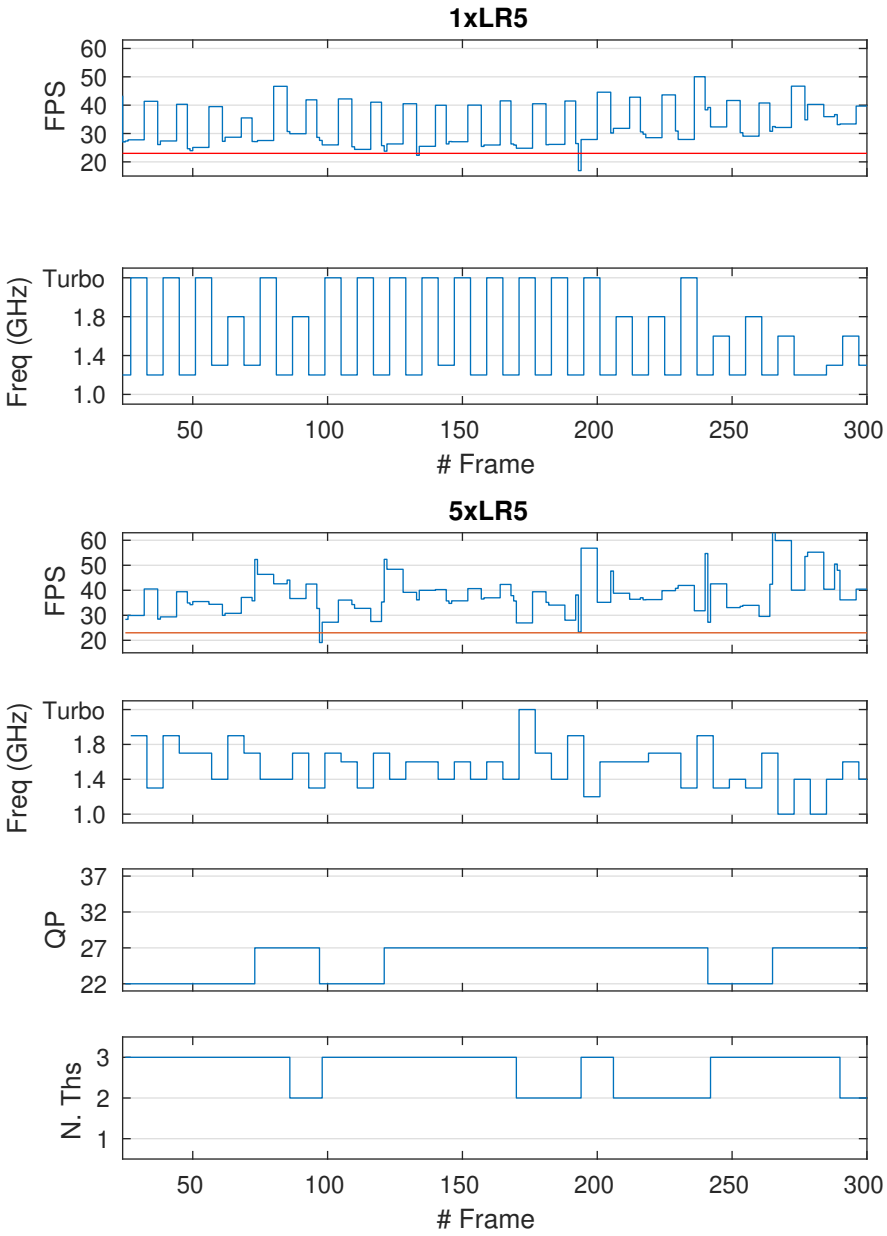


Figure 7.2: Encoding timeline for a single LR5 sequence (top) vs. 5 LR5 simultaneous sequences (bottom). For a single sequence, the system sets the number of threads to 3 and the QP value to 22 (not shown in the plots).

7.2.1. MAL and turbo behaviour

Consider the traces shown in Figure 7.2. The plots on the left represent an encoding process of a single high-demanding LR5 video, while the plots on the right show the behaviour of when 5 LR5 videos are encoded simultaneously (only traces of one of them are shown, but all sequences have similar behaviour). When a single video is encoded, the MAL system decides in this case to set the number of threads to 3 and QP value to 22, adapting itself to the video content changing only the frequency between 1.3 GHz and turbo frequency (3.4 GHz due to low occupation). This policy allows the system to codify the video with the maximum quality and still to not violate the throughput constraint set.
However, as described in Section 5.2, as the number of physical cores in use increases (and therefore $\mathcal{S}_{occ}$ increases too), the effective value of the turbo frequency decreases (down to 2.5 GHz when there are 15 cores occupied) not being enough to maintain the codification process above 24 FPS. Consequently, the system learns autonomously how to adapt to the content by properly modifying the other knobs. Similarly to the policy learned for 1 HR video, the system follows the same knob classification to tune the process: frequency is changed the most as it provides the finest control on the process, followed by less frequent changes in the number of threads, and lastly changes in QP, as it can make huge changes in the output, but provides no much control on the metrics. This is the general policy followed in all the videos and scenarios tested as described next. It is important to notice that there are not different tables nor learning process for different number of videos, but there is only one table for each agent, and thanks to the definition of $\mathcal{S}_{occ}$, the agent is able to predict the effective frequency of turbo, and therefore, to act consequently.

7.2.2. Comparison with a STATIC solution

To compare our approach with the STATIC approach, we have evaluated both in two different scenarios. In the first scenario we run as many videos of the same resolution as possible, avoiding oversubscription. This scenario tests the behaviour of the MAL system when encoding from 2 to 4 simultaneous high resolution random videos, and from 2 up to 6 low resolution sequences being encoded concurrently. The second scenario corresponds to a more realistic scenario, where multiple videos of different resolutions need to be encoded simultaneously satisfying a THROUGHPUT above 24 and maximizing quality at the same time.

Scenario 1. - Homogeneous-resolution videos: Table 7.1 shows the results of executing multiple videos of the same resolution concurrently, comparing MAL and STATIC. The sequences used in each experiment were randomly chosen, ensuring all the possible combinations in the number of high-demanding and non-demanding videos were explored. The main conclusion we can extract from the results is that when there is more than one video being encoded simultaneously, the results clearly show how our approach is consistently able to encode the different workloads with a low number of QoS violations, adapting to different occupation levels and video resolutions (mainly increasing QP, that is, decreasing quality to compensate the lower turbo frequency). While the STATIC approach works relatively well for a low number of videos, its QoS decreases when the computational demands increase, as shown in the table.

		Output metrics				*Avg. Knob values*		
		–Δ–		**PSNR** (dB)		**Freq**	**N. Ths**	**QP**
		MAL	STATIC	MAL	STATIC	MAL		
HR	2×HR *Avg.*	3.1	5.4	40.3	40.1	1.3	3.3	31.8
	3×HR *Avg.*	4.5	8.5	39.4	40.1	1.3	3.3	33.7
	4×HR *Avg.*	7.1	9.9	39.0	40.1	1.4	3.1	34.8
LR	2×LR Avg.	11.1	11.1	44.0	44.6	1.5	2.6	23.6
	3×LR Avg.	12.7	16.1	43.7	44.6	1.5	2.6	24.1
	4×LR Avg.	13.1	19.9	43.9	44.6	1.6	2.5	23.7
	5×LR Avg.	15.0	22.7	43.3	44.6	1.5	2.6	25.2
	6×LR Avg.	13.8	35.0	41.7	44.6	1.5	2.6	28.6

Table 7.1: Output metrics and resource usage for the MAL approach compared with the STATIC assignment for HR (top) and LR videos (bottom) when encoding multiple time simultaneously.

Diving into details of the behavior of each experiment, we can extract a number of specific insights, namely:

- As shown in Section 6.3, when there is only one video running, the STATIC approach behaves relatively well for most of the sequences. However, when the number of concurrent videos increases, so does the number of QoS violations.
- The flexibility of the MAL approach allows a dynamic adaptation of knobs during the execution to satisfy the different demands of the different videos on different server loads, obtaining better results than the STATIC approach. In the case of HR sequences, the degradation of the QoS ranges from 3.1 % to 7.1 % when the MAL approach encodes 2 to 4 simultaneous videos respectively, and from 5.4 % to 9.9 % when the videos are encoded using the STATIC approach. When LR sequences are encoded, it ranges from 11.1 % when 2 videos are encoded simultaneously (for both approaches), up to 13.8 % in the MAL approach and 35.0 % in the STATIC approach when 6 videos are considered.
- Although the MAL approach decreases the PSNR when the number of videos increases, the loss in quality is only of 2.9 dB in the worst case, thanks to the QP adaptation previously described.

To recap, the percentage of time in which the QoS requirements are violated increases with the number of simultaneous videos. However, the QoS degradation is greater for STATIC than for MAL. Hence, our proposed solution is able to adapt to different levels of occupation, which results in more desirable outcomes (1.7× and 2.5× for HR and LR videos respectively). This demonstrates how the MAL system is able to self-adapt to server's occupation, in addition to the adaptation to the video contents described before.

Scenario 2.- Behaviour under video combinations: Table 7.2 shows the results obtained for a more realistic scenario, where different number of videos of different resolutions are

	PSNR (dB)		–Δ–		**N. Threads**	
	MAL	STATIC	MAL	STATIC	MAL	STATIC
1HR+1LR	41.5	42.0	1.2	0.3	5.7	6
1HR+3LR	42.8	43.4	7.6	9.9	10.8	12
1HR+5LR	42.0	43.8	11.6	10.7	14.8	18
2HR+1LR	41.0	41.5	2.8	2.3	8.6	9
2HR+3LR	39.6	42.6	6.3	3.8	13.3	15
2HR+5LR	39.6	43.3	10.0	20.7	18.6	21
3HR+1LR	39.4	40.9	2.3	3.7	10.8	12
3HR+3LR	39.8	42.3	10.0	14.4	16.8	18
3HR+5LR	38.6	42.7	11.3	26.6	20.0	24
4HR+1LR	39.1	40.8	3.6	3.9	14.0	15
4HR+3LR	38.1	41.8	7.1	16.4	19.0	21
4HR+5LR	38.1	42.6	9.0	35.5	20.0	27

Table 7.2: Output metrics and number of threads (MAL vs STATIC) for different combinations of mixed videos.

simultaneously encoded. Each reported result corresponds to the average value of 10 different mixes of videos, each repeated 3 times. Qualitatively, the behavior is similar to the previous experiments: as the occupation of the server increases, the QoS violations increase too. In addition, in the extreme cases where the STATIC approach assigns more threads than available cores (20 in our platform) arising oversubscription (i.e. two active threads are executed on the same physical core), our approach is able to adapt the quality and number of threads between the videos in order not to exceed the available physical cores and to obtain maximum QoS. MAL obtains 2× improvements in QoS when compared against STATIC assigning 21 threads (experiment 2HR +5LR), and up to 4× when 27 threads are used by STATIC (experiment 4HR +5LR). The loss in quality is minimum, i.e., encoding always sequences with PSNR above 38 dB.

7.2.3. Comparison with a state-of-the-art heuristic (ARGO)

HR		***Output metrics***				***Avg. Knob Values***					
		–Δ–		**PSNR** (dB)		**N. Ths**		**QP**		**Freq**	
		MAL	ARGO	MAL	ARGO	MAL	ARGO	MAL	ARGO	MAL	ARGO
2×HR	*Avg.*	3.1	9.4	40.3	42.7	3.3	1.6	31.8	25.8	1.3	*Turbo*
3×HR	*Avg.*	4.5	6.3	39.4	42.7	3.3	1.7	33.7	25.8	1.3	*Turbo*
4×HR	*Avg.*	7.1	7.1	39.0	42.7	3.1	1.9	34.8	25.9	1.4	*Turbo*

Table 7.3: MAL compared to the ARGO approach when encoding multiple videos at the same time. MAL results are reported again with comparison purposes.

Table 7.3 shows the results obtained when execution the same combination of videos used in the previous section. For the sake of clarity, we report the MAL results again in this table.
Similar to the results obtained when only one video was encoded (Section 6.3), ARGO approach is able to encode the different sequences with a slightly better quality due to the use of a lower QP value. However, this increase in quality carries an increase in the complexity of the encoding process, that ARGO tries to compensate supported by the *turbo* frequency of the system. However, similar to the experiments for one video, the amount of time the real-time constraint is violated is superior than in MAL (up to 3×).
Table 7.3 also shows that the number of threads used by ARGO increases with the number of videos. As described when talking about turbo behaviour in Section 5.2, the turbo frequency decreases as the number of active core increases. Hence, an increase in the amount of threads is needed to compensate the loss in frequency. Opposite to the strategy followed by ARGO, observe how the policy learned by MAL prefers to decrease the quality slightly (increasing the QP value) instead of increasing the number of threads.

7.2.4. Improvements over a mono-agent implementation

To show the advantages and disadvantages of our multi-agent proposal against other mono-agent-based Q-Learning approaches, we have implemented the general mono-agent approach described in previous chapters. For the sake of fairness, the mono-agent takes an action every 6 frames (the minimum frequency in MAL), and considers all possible combinations of actions considered by MAL. All results hereafter correspond to a training process with 4 HR simultaneous transcoding processes, using the sequences for training and test described before. Only results with 4 simultaneous videos are reported due to the unfeasible time required to train the mono-agent system for different number of videos, as commented in the previous chapter and described with more details next. The reported results correspond to the average values obtained when encoding 4 test sequences selected randomly at the same time (each executed 5 times), using the same combination of sequences for both approaches.
Data measured confirm the weak points previously mentioned for the mono-agent approach and alleviated by the MAL implementation, namely:

1. The *learning time* for the mono-agent is dramatically larger than that of the MAL approach (6 times longer in our experiments).
2. The mono-agent approach has *less control* on the consequences of each action as the decision space considers all the knobs together, hiding the relations between them.
3. Changing all the knobs constantly (specially the output quality), can produce a negative impact on the Quality of Experience (QoE).

Learning time: The two plots on the top of Figure 7.3 report the learning time for the mono-agent and multi-agent approach, respectively, under equivalent experimental scenarios. Each line represents the amount of *state/action* pairs (in terms of percentage) at each phase of the learning process. In the case of the multi-agent approach, lines represent combined data for the three agents. Both x-axes are equally scaled for comparison purposes,

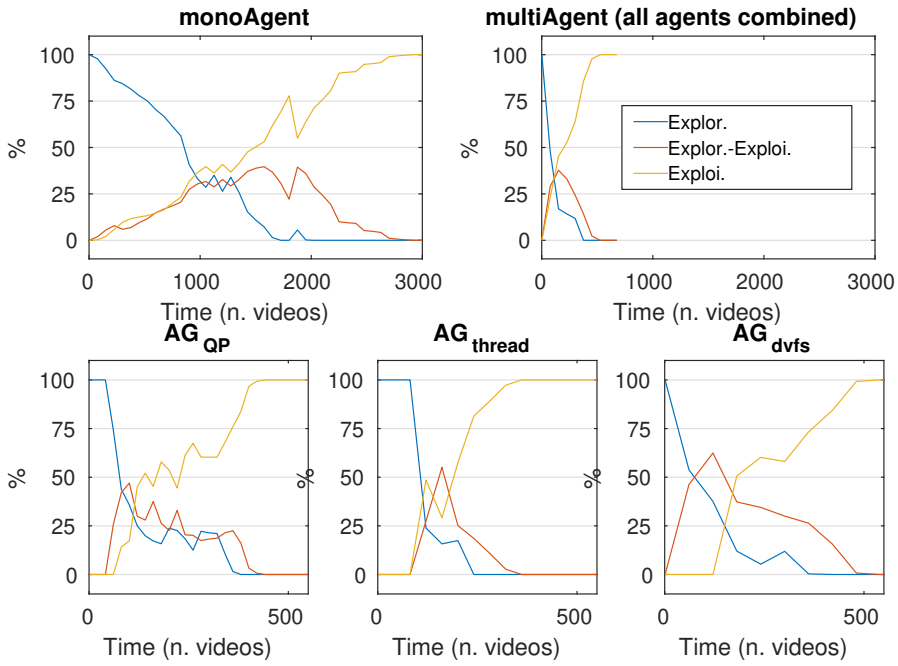
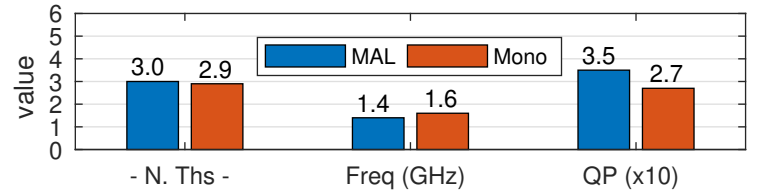


Figure 7.3: Learning evolution of the mono-agent approach vs multi-agent approach. Each line represents the percentage of state-action pairs that are in each phase. Top, the mono-agent approach vs the multi-agent approach (all agents combined). Bottom, a detailed view of the behavior of each agent.

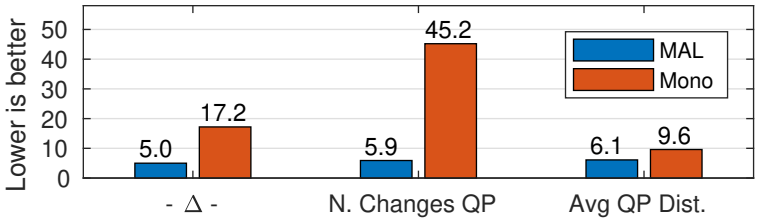
and represent the status of the system after training with a certain number of sequences ($\approx 500 frames/sequence$). The results clearly show how all *state/action* pairs start in the *exploration* phase, and how they move to *exploration-exploitation* and then to *exploitation* phase while the pairs are being visited over time. As described in Section 6.2.4, running the system on a real platform produces noisy measurements, which forces the adoption of filtering techniques to remove these noisy data. Our filtering algorithm does not remove these measurements, but ignores them until it is sure that these measurements are correct (Equation 6.4). This leads to the lines in the plot not to be monotonic, but still convergent. As the decision space for the mono-agent considers all combinations for all actions (4 QP values $\times$ 5 different numbers of threads $\times$ 12 frequencies = 240 different actions), the convergence for the mono-agent is slower than for the multi-agent, which splits the decision space and explores them concurrently, yielding 6$\times$ faster learning times for the tested setup.

The plots at the bottom illustrate the learning process of each agent in MAL. Even though all of them show the same behavior, the convergence slightly varies among them, due to the different number of actions each agent needs to explore, as well as by the different frequency at which each agent acts.

Learned policies: Figure 7.4a shows how the number of threads and selected frequency are similar to those learned by MAL (3.0 vs 2.9 threads for the MAL and mono-agent respectively, and 1.4 GHz vs 1.6 GHz); in the mono-agent case, QP values are considerably smaller (26.9 vs 34.6), yielding higher-quality videos while the computing resources increase. This implies



(a) Avg. Knob values.



(b) QoE metrics.

Figure 7.4: Top: resource usage by the MAL and mono-agent approach. Bottom: QoS and QoE metrics obtained. The data represents average values for different combinations.

an increase in the QoS violations shown in Figure 7.4, obtaining worse results than for MAL. The reason of this behavior is that, when considering simultaneous knob changes, the relation between them is hidden, obtaining at the end a more coarse-grained control than the MAL approach.
Also, MAL does not only rely on the *Q-values* learned on the decision process, but also on the stored probabilities between states when applying each action. Thus, it has more information than the mono-agent to make right decisions.

Actions variability (QoE): In addition to QoS, it is also crucial to consider Quality of Experience (QoE) [142]. This is directly related to the perception of the user when visualizing the encoded sequence. For example, even if the frame rate is always above 24 FPS, continuous or abrupt changes in quality can damage QoE.
To quantify these two metrics, Figure 7.4b reports the number of QP changes during the encoding process (i.e. quality changes), and average distance between the selected QP values (if the distance is large, the change in quality is significant). Besides the coarser-grained control of the mono-agent when knobs are considered jointly, it is impossible to set a different frequency (for agent activation) to modify each knob. This produces a higher number of changes in QP than MAL (in our case, every 6 frames vs every 24), and hence constant changes in quality. In addition, as the mono-agent learned to use a lower QP value with the same number of threads and frequency as MAL, when the system needs to adapt the QP value, it performs abrupt changes, damaging QoE. For the example shown in Figure 7.4, the benefits in terms of QoS and QoE can be summarized as a reduction of 12 % in FPS violations, 7× less QP changes, and 30 % reduction in average QP distance.

7.2.5. Power capping integration

Finally, we investigate on the ability of MAL to apply tight power capping limits while adapting resources and keeping thresholds on THROUGHPUT and PSNR. The experiments are based on the execution of the maximum number of similar videos before oversubscription arises (in our case, 6 LR simultaneous videos), combined with a power capping limit (75 W, which is 60 % of TDP). We study three different power capping mechanisms, adding to the previous MAL formulation power capping abilities via software and via hardware, and compare all the results with similar executions where no power capping were applied:

Software capping (MAL-SW): This approach corresponds to our MAL formulation in which MAL autonomously learns the optimal knob combinations to achieve power capping. This approach correspond to the one presented at the beginning of this chapter.

Hybrid software-hardware capping (MAL-SWHW): A version of MAL in which no power states are considered at learning time and there is no power reward, but instead hardware mechanisms are applied to maintain power under the cap. In our platform, this is done via the Intel-RAPL [138] mechanism.

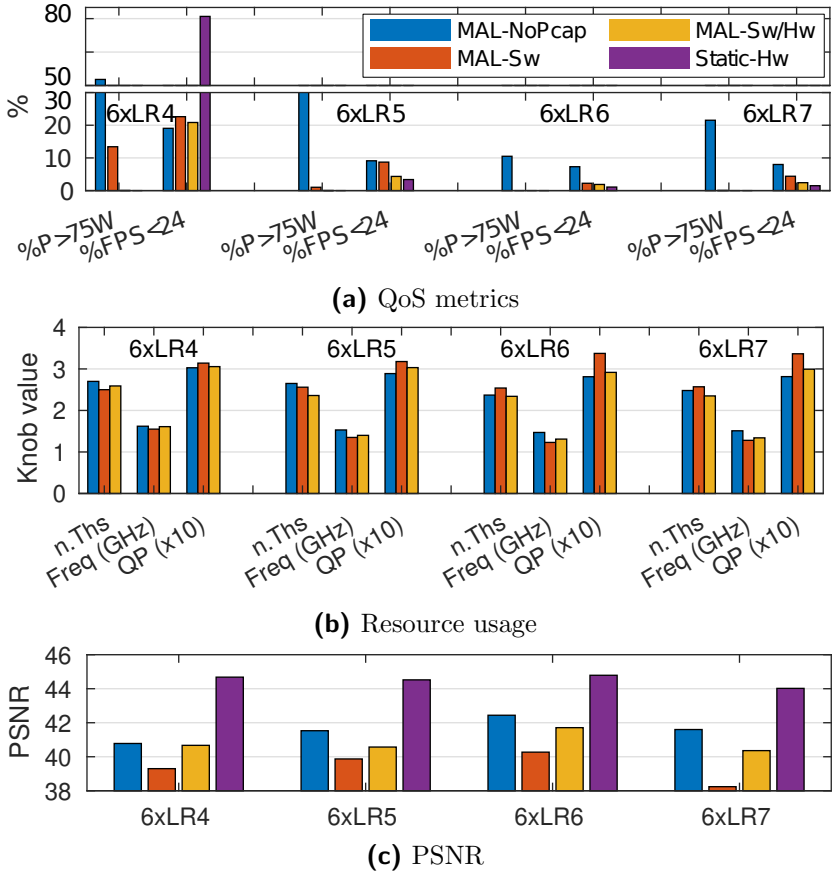
Static-hardware capping (STATIC-HW): An implementation with hardware capping where the values for the different knobs are statically selected a priori. The values chosen for the STATIC-Hw approach correspond to the average values learned by MAL.

No powercapping (MAL-NOPCAP): executions using MAL with no power capping abilities. Used in this section as the baseline to compare with the other three approaches.

We report results in terms of used resources (Figure 7.5b) and output metrics (Figure 7.5a and Figure 7.5c) for the three power capping mechanisms and for MAL-NOPCAP, which means MAL with TDP as power cap.

First, consider the capabilities of the three methods to maintain power consumption below the cap. For reference, if none of the three previous mechanisms is used, the amount of time in which the cap is exceeded in a normal execution of the MAL system ranges from 52 % (LR4) to 10 % (LR6). By using any of the three approaches, power capping violations (labelled as $\%P > 75W$) reduce this range in all cases. When hardware capping is applied, obviously, the percentage of capping violations is reduced to a value close to 0 %. In the case of pure software capping, it ranges from 13 % (LR4) to 0 % (LR6), which demonstrates the ability of MAL to dynamically adjust knobs to meet the power capping requirements.

Regarding QoS, both software-hardware and software power capping are able to achieve similar FPS violations (columns labelled as $\%FPS < 24$) in all cases compared with situations in which no power capping is applied. Recall that, in those cases, power capping is simultaneously met. This achievement is possible due to a correct adaptation (reduction) of quality (PSNR). At a glance, MAL manages to slightly reduce average core frequency, from a range between 1.62 GHz (LR4) to 1.47 (LR6) to a range between 1.55 GHz (LR4) to 1.23 (LR6). This reduction is accompanied by an increase in QP, and hence a reduction in quality (PSNR), see Figure 7.5c. Note that, however, quality is maintained under acceptable limits. In summary, the static approach combined with hardware power capping mechanism is able



(a) QoS metrics

(b) Resource usage

(c) PSNR

Figure 7.5: (a) QoS, (b) resource usage, and (c) quality for the same 6 videos simultaneously encoded under a power cap of 75 W, using the different policies described in the text. The bars represent: (blue) the MAL system without power cap; (red) the MAL system when power capping is learned and applied via software; (yellow) the MAL system when power capping is applied via hardware; and (purple) the STATIC approach combined with power capping via hardware.

to keep the power consumption under the power limit set, at the expense of a considerable number of QoS violations. On the contrary, MAL achieves software power capping by means of reducing the frequency and trading off quality. Besides, if hardware capping mechanisms are available, MAL can cooperate with them.

7.2.6. Overhead introduced by the MAL system

To minimize the overhead introduced by MAL in a typical real-time Kvazaar encoding process, the system has been designed trying to minimize as much as possible the time Kvazaar has to be blocked waiting for the synchronization with MAL. To do that, the system has been designed to be able to execute in parallel with Kvazaar without interrupting the normal operation of it. However, there are three moments of the execution where Kvazaar has to interrupt its execution to synchronize with MAL:

Comm. Type	Type	Time (μs/comm)	Freq (N/sec)	% Time
Transfer Metrics	Rx	5.68	24	0.014%
Adjust QP	Rx & Tx	20.16	1	0.002%
Adjust n. Ths	Rx & Tx	95.44	2	0.019%

Table 7.4: Overhead introduced by MAL into the normal Kvazaar operation. The measured time comprises the time taken by the transmission of the message from Kvazaar to MAL, the computations of the new metrics by MAL, and the transmission time of the message containing the new knobs.

1. To transfer metrics. Each time Kvazaar finishes to encode a frame, the output metrics of that frame are sent to MAL. This message has no answer from MAL, so Kvazaar can start to encode the next frame without needing to wait for the answer. This transfer is also used to notify MAL about the beginning of a new frame, and therefore, to determine if AG_{dvfs} needs to act. In that case, the frequency is modified at the same time Kvazaar starts to encode the next frame, avoiding to block Kvazaar.

2. To ask for the new QP value. Every 24 frames, Kvazaar asks MAL about the QP value to use in the following 24 frames. In this case, Kvazaar blocks until the MAL system decides the next value to use (i.e., AG_{QP} determines the next action), and receives the answer.

3. To determine the number of threads to encode the next 12 frames. Similar to the previous communication, Kvazaar blocks until MAL determines the number of threads to use in the following 12 frames. As described previously, to determine the number of threads to use, not only AG_{thread} has to act, but also the system has to determine the affinity of those threads to avoid oversubscription situations.

Table 7.4 shows the amount of time each of these communications take individually, and the percentage it represents of the whole process. As expected, transferring the output metrics is the fastest communication as Kvazaar has not to wait for an answer, while asking for the number of threads to use is the slowest because the system has to determine and set the affinity of each thread. In general, the amount of the time Kvazaar is blocked is below 0.03 %, being low enough to not affect our real-time encoding processes.

7.3. Conclusions

In this Chapter, we have extended the ideas presented in the previous Chapter to support inter-application dependencies by our Q-Learning formulation. In addition, we have extended the capabilities of our approach to support power capping restrictions. To do so, the learning process, as well as the management of the shared resources were modified to consider the possible interferences between agents of different processes.

The proposal was tested on different real scenarios, comparing our approach with other state-of-the-art approaches. In particular, the results have demonstrated that our multi-agent approach adapts to changes in video contents and server occupation, achieving an

improvement of 2×/4× when the occupation of the server is low/high, respectively. We have given evidences that reveal the appealing of a multi-agent approach in terms of learning time (6× reduction compared with a mono-agent approach) and quality of learned policies (3.4× improvements on QoS, and 7× in QoE). We have shown how power capping capabilities can be incorporated into the resource manager obtaining competitive results when compared against hardware power capping mechanisms. The management of dynamic application- and system-level knobs in a holistic fashion is general enough to be extended with further parameters or output metrics, and to other applications, both in the multimedia area and in other fields; also, the architectural-related techniques applied to deal with system knobs are of wide appeal to be applied (isolated or in conjunction) to other present and future architectures.

This Chapter, together with the previous one, have proven our proposal as a valid approach to handle multiple applications concurrently in a real environment. To do so, our formulation relies on a QL approach to handle the multiple application- and system-knobs based on the metrics received from both application and system. However, the current proposal considers only the generation and utilization of one policy governing all the applications simultaneously. In the next Chapter, we extend the current formulation to generate multiple policies with minimum effort, and show how a real system can benefit from having multiple policies, deciding which policy to apply to each application at each moment, based both in application- and system- metrics.

8

A Methodology for Multi-Policy Resource Management

Chapter 7 has demonstrated the feasibility of applying a Reinforcement Learning strategy to manage both resources and application knobs via a centralized resource manager, able to autonomously learn and apply policies both at application and system-wide level to optimize a common QoS objective. In all cases, the learned policies are considered to be homogeneous across applications. However, it is common to find scenarios in which a number of parallel applications coexist, each one featuring *different Quality of Service (QoS) requirements*. This heterogeneity in requirements entails new challenges in terms of resource management in order to improve resource usage while meeting acceptable rates in terms of QoS.

With this regard, in this chapter we pursue two different objectives:

1. We propose an efficient methodology to build multiple policies that depart from a common Reinforcement Learning formulation, and that can be served to different applications at runtime to fulfill heterogeneous QoS requirements. Specifically, we show how the Reinforcement Learning formulation described in the previous chapters can be reformulated to obtain multiple policies with minimum effort and learning time, and we propose a simple methodology to accomplish this goal in an easy and controlled manner.

2. We show how multiple of these ML policies can be combined together at a higher level. To demonstrate that, we build a two-component resource manager that can handle multiple applications concurrently with different QoS requirements. This two-component resource manager is comprised by a MAL system similar to that described in previous chapters, and a heuristic layer determining which policy will be applied to each application at each moment. Here, the MAL system is in charge of tuning the different application- and system-knobs based on the measured metrics and the learned policy, while the heuristic component is in charge of communicating with the

MAL system about which policy apply to each application, based on system metrics and its internal status.

Specifically, Section 8.1 motivates the existence of real scenarios where different QoS policies are applied simultaneously, and describes how different policies can be formulated for our driving example, HEVC online video transcoding. Section 8.2 describes a framework to obtain multiple Q-Learning policies with minimum effort while minimizing the learning time, showing the results obtained when the system encodes different sequences with different policies. In Section 8.3 we describe how the resource manager can be extended to incorporate multiple policies, describing a simple heuristic aware of the system status to determine the best policy to apply to each instance. Lastly, Section 8.4 closes the chapter with some conclusions.

8.1. Motivation for multi-policy resource management

The integration of multiple polices for resource management and application tuning in shared computing systems, specially in the cloud, is becoming a field of paramount interest to efficiently exploit the potential of the underlying architectures without human intervention. In situations where limitations in terms of Quality of Service (QoS), tight per-application Service Level Agreement (SLA) or energy consumption are imposed, the development and application of such policies become a hurdle difficult to be automatically addressed [189].

Usually, the development of those different policies is commonly carried out in a loop fashion where different metrics and actions are tested and applied to applications and system, sensoring the effects on the output metrics. This cyclic steps orbit around the existence of a shared Knowledge Base (KB) [90] storing rules that, properly orchestrated, can fulfill the requirements and restrictions imposed without further human intervention.
The development of the KB, however, can become a daunting task when the amount of architectural and application-level knobs increase and their interplay is nontrivial. In stochastic environments such as shared nodes in cloud deployments, in which the application of a given rule does not always yield the same result in terms of performance and/or energy consumption, the creation, maintenance and effective application of the knowledge of the KB is even a more complex task. The challenge is harder in scenarios in which the request arrival rate and its distribution are unknown, or when the throughput or quality attained are content-dependent, and hence *unpredictable*.

In the following, we leverage RL to build, maintain and enrich the Knowledge Base of our centralized resource manager in shared servers in order to attain automatic and efficient resource management and allocation for multiple concurrent applications exhibiting *heterogeneous* QoS demands.

8.1.1. Heterogeneous QoS for HEVC encoding processes

In the previous chapters we have shown how a multi-agent Q-Learning approach can be applied to a real-world HEVC encoding application to obtain simultaneous real-time encoding processes (that is, with a tight lower bound in throughput of 24 frames per

second –FPS–), meeting at the same time constraints in terms of quality and power consumption. However, in that scenario, a *unique* common policy was applied to all the videos concurrently processed by the system.

For the sake of clarity, let us consider now a simpler problem formulation in which turbo frequencies are not considered, and power is considered as an independent value for each application, based on a previous profiling of the system and online estimation, similar to Chapter 4. Also, consider a slightly different decomposition of the states as shown in Equation 8.1. Similar to the previous chapter, we design a reward function composed by three different sub-reward functions, each one associated to a specific sub-state, and its coefficients as described before. In this scenario, and assuming the state definition and action set stay constant, this simpler formulation offers 9 different dimensions (3 reward definitions $\times$ 3 coefficient values) the designer of the experiment can tune to obtain different policies.

$$\begin{aligned} \mathcal{S}_{psnr} &\equiv \{\, \text{PSNR} \in (0,36], (36,38], (38,40], (40,42], (42,44], (44,\infty)\ dB \,\} \\ \mathcal{S}_{FPS} &\equiv \{\, FPS \in (0,24), [24,30), [30,40), [40,50), [50,\infty) \,\} \\ \mathcal{S}_{power} &\equiv \{\, power \in [0,17), [17,21), [21,24), [24,\infty) \,\} \end{aligned} \tag{8.1}$$

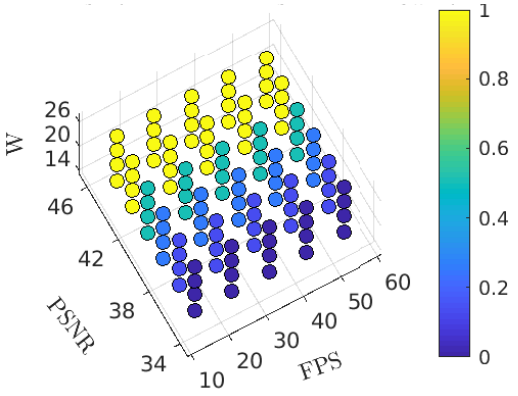
Figure 8.1 shows different combinations of sub-rewards and coefficient values in this scenario. Each dot in the plot represents different state in our formulation; and its color represents the reward given to that state (the higher the better). On the left, three different examples of sub-reward functions are shown, namely:

1. R_{PSNR-H} which gives maximum reward to the states with maximum quality.
2. R_{PSNR-L} which minimizes quality, but ensures a minimum quality (giving a reward of 0 to the states below the threshold).
3. R_{FPS} which aims at obtaining real-time encoding processes giving no reward to the states with throughput below 24 FPS, maximum reward to the states between 30 and 40 FPS, and a decreasing reward to the states above.

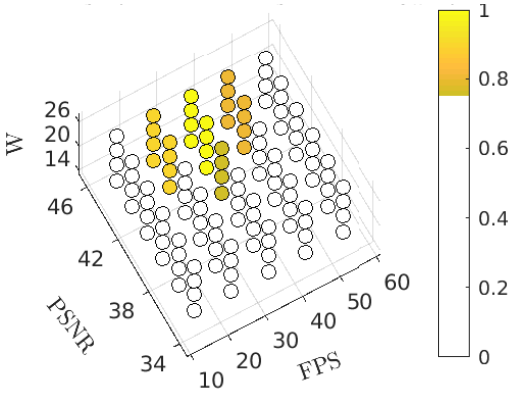
Observe how modifications in the sub-reward functions alter which are the best solutions, or goal states (yellow points in the figure). This fact can be clearly observed in the first two plots: while the former one maximizes quality, changing the definition, the latter turns the states with minimum quality, but above the constraint (36 dB in the plot), into the goal states.

On the right, different combinations of the same sub-reward functions with different coefficients are shown. For the sake of clarity, only the states with reward ≥ 0.75 are colored. In this case, R_{POWER} minimizes power consumption. Observe how small changes in the way functions are combined dramatically alter the goal states of the system. For example, results vary from maximizing quality and ensuring real-time throughput on the top, to restricting the goal space to those with minimum power, meeting real time

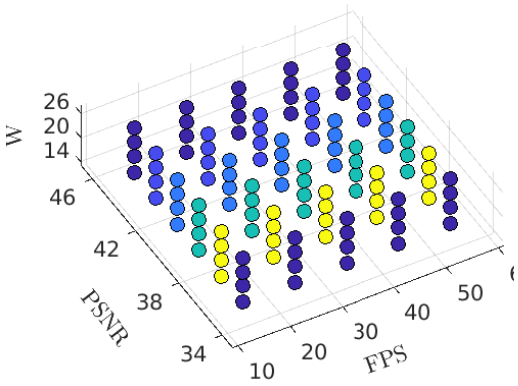
$1.0 \cdot R_{PSNR-H} + 0.0 \cdot R_{FPS} + 0.0 \cdot R_{POWER}$



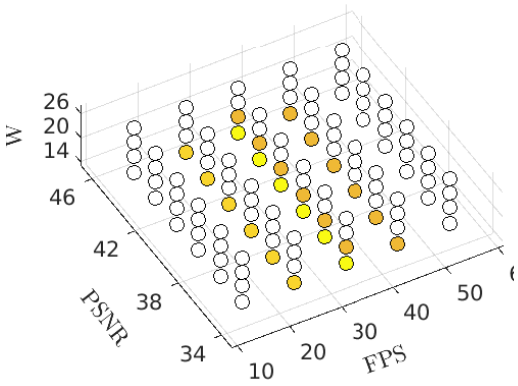
$0.5 \cdot R_{PSNR-H} + 0.5 \cdot R_{FPS} + 0.0 \cdot R_{POWER}$



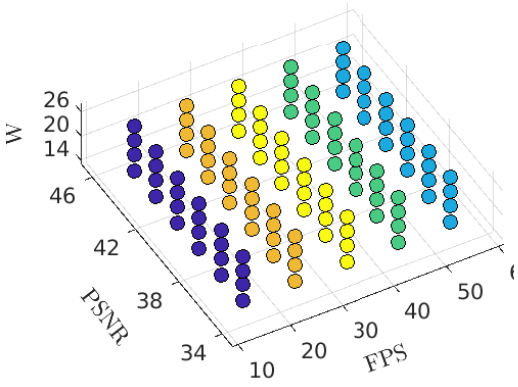
$1.0 \cdot R_{PSNR-L} + 0.0 \cdot R_{FPS} + 0.0 \cdot R_{POWER}$



$0.0 \cdot R_{PSNR-H} + 0.5 \cdot R_{FPS} + 0.5 \cdot R_{POWER}$



$0.0 \cdot R_{PSNR-H} + 1.0 \cdot R_{FPS} + 0.0 \cdot R_{POWER}$



$0.3 \cdot R_{PSNR-H} + 0.3 \cdot R_{FPS} + 0.3 \cdot R_{POWER}$

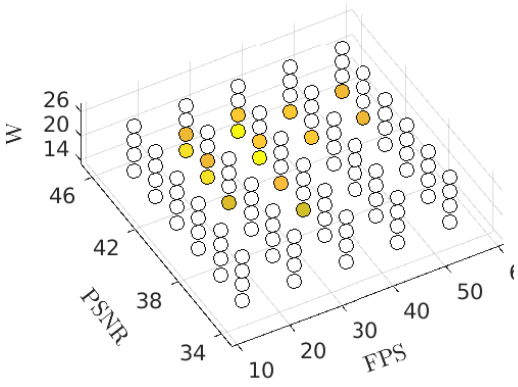


Figure 8.1: Rewards obtained in the different states for different combinations of sub-reward definitions (left) and coefficients (right)

requirements in the middle, and a combination of both behaviors on the bottom.

The figure ultimately demonstrates how, even for a simple space decomposition, there is a huge number of different policies that can be applied to the same problem, obtaining different results. However, the definition of those policies, and its subsequent extraction is far from being trivial, and it is definitely a time consuming task if it is not carried out following a proper methodology. In the next section, we proceed with a description of how the creation of different policies can be standardized, and we propose a methodology to follow when creating the following policies to alleviate this burden.

8.2. Designing a Reinforcement Learning multi-policy framework

Reinforcement Learning (RL) algorithms tackle the problem of finding the optimal policy of a MDP where the set of probabilities ($\mathcal{P}$) relating the different actions and transitions between states are unknown (see Section 5.3). In most real problems, determining the probabilities that define a specific MDP is not an easy task, being in most of the cases unknown, or estimated from noisy measurements.

Throughout the previous chapters, Q-Learning (QL) has been proved to be a valid RL algorithm to learn the optimal policy through an autonomous exploration of the design space (and inherently, discovering the probabilities that define the problem). However, due to the infinite nature of the formulation of the algorithm, the optimality of the obtained policy π' will be ultimately based on the time the algorithm has been exploring the system and, in essence, the distribution of the number of times each *state/action* pair has been explored. In real-life problems, where the definition of the state comes from real measurements of the environment, this exploration time depends on the frequency the system provides the different metrics, producing long time training sessions. Moreover, if the final goal is to train the system several times to obtain different behaviors (with a modification in the reward functions prior to each iteration), the total training time can grow to unacceptable times.

In the following, we address both questions, namely:

1. How to define the system to obtain different learned behaviors.
2. How to boost the learning time providing the transition probabilities to the Q-Learning system.

8.2.1. Learning different policies

Training a Q-Learning system is a trial-and-error process in which state and reward function definitions are modified until the obtained policies meets the desired behaviour. Even with expert knowledge, this process can be tedious or even unfeasible in some scenarios. To reduce this burden, in Chapter 6 we proposed the decomposition of states and rewards into multiple substates, allowing us to propose a new methodology to obtain new policies with minimum effort:

$$s = (s_1, \ldots, s_n), \quad \text{with } s \in \mathcal{S} = \mathcal{S}_1 \times \ldots \times \mathcal{S}_n,\ s_i \in \mathcal{S}_i$$

$$R(s) = R(s_1, \ldots, s_n) = \lambda_1 \cdot R_1(s_1) + \ldots + \lambda_n \cdot R_n(s_n), \quad \text{where } s_i \in \mathcal{S}_i,\ \lambda_i \in \mathbb{R}$$

Given this definition of a reward function, training the system to obtain different policies boils down into tuning each sub-reward to obtain the desired behaviour, by means of a tuning-and-test cycle as described in Section 8.2.3. This tuning process can be carried out in two different ways based on the desired behaviour:

- *Modifying the coefficients:* Assuming all sub-reward functions have the same range (i.e., all functions produce values in the same interval), each coefficient λ_i represents the importance of each sub-state in the problem. Modifying these coefficients allows us to give more or less importance to each sub-state, and consequently to the metrics used to build it.

- *Modifying the reward functions:* Each sub-reward function represents how the system will behave, respectively, on each sub-state (and, therefore, on each metric used to define each sub-state). The goal of modifying the definition of a sub-reward is not to modify the importance given to a set of metrics as before, but instead to modify the behaviour of the system with respect to this metric. For instance, changing one sub-reward function can imply modifying the behaviour from maximizing certain metric to minimizing it.

8.2.2. Reducing learning time

As described in Chapter 5, classical Q-Learning formulations are based on a table combining all the actions and states (*Q-table*), and representing the expected rewards obtained for each pair. This table, initially empty, is updated at the same time the system explores the different transitions. As the exploration level advances, the learned policy is more similar to the optimal policy, but the convergence ratio decreases. It is a responsibility of the designer of the experiment to set a trade-off between the exploration time and the optimality of the learned policy. Implicitly, at the same time the table is updated, the system is learning the unknown probabilities between states and actions ($\mathcal{P}$).

Although at a first glance the bottleneck of the algorithm seems to be the number of the iterations the system needs to perform in order to explore all the transitions enough number of times, in real-world problems it is limited by the frequency at which the actions can be applied and the metrics needed to build the states can be measured. For example, in a situation in which the system is applied to a video encoding process at 24 frames per second, and a transition occurs between frames, the speed of the exploration is limited to 24 transitions per second. In the case when multiple policies are required, and consequently, one training session per policy is needed, this limitation in the speed of the algorithm can make the problem unfeasible.

However, if the probability set between states ($\mathcal{P}$) is known a priori, the algorithm does not need to wait for actual readings of the required environmental metrics to determine movements between states; on the contrary, it can simulate the transitions based on the information provided by $\mathcal{P}$.

Following this idea, we propose an *offline learning process* which dramatically reduces the amount of time needed to obtain each policy:

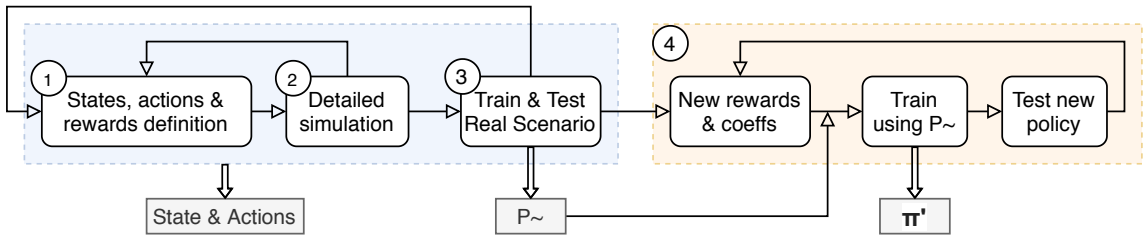


Figure 8.2: Proposed methodology to extract multiple policies from the same KB.

1. First, all combinations between states and actions are explored enough number of times to build a transition table ($\mathcal{P}'$) that stores the probabilities of moving from one state to the others when applying a specific action.

2. Once $\mathcal{P}'$ has been built, following training processes can proceed with the classical Q-Learning formulation. However the state is determined based on the information provided by $\mathcal{P}'$, not by observation.

The process of building $\mathcal{P}'$ can be carried out independently of the learning process, or can be extracted from one initial training process storing explicitly the probabilities at the same time the classical Q-Learning algorithm explores the different transitions of the system.

The obtained $\mathcal{P}'$ table will be valid for all the subsequent learning processes, unless the transitions between states change due to external factors (for example, processor operating frequency can be altered by changes in temperature), or due to changes in the definition of states or actions. Note that $\mathcal{P}'$ is used only in the learning process, but once the system has finished learning, the states and transitions are obtained directly from measurements of the system, and *not* from $\mathcal{P}'$.

8.2.3. A methodology to extract multiple policies

As explained before, the creation of these policies can be a time-consuming process due to the tune-and-test nature of the process. To reduce the amount of time required to complete the process, we propose a simple and concise methodology to generate the different policies based on the previous ideas, removing a reasonable amount of time and effort from the process:

(Step 1) In a first step, the definition of the different states and rewards is carried out. Each sub-state needs to be discretized based on expert knowledge of the problem, while the reward functions are defined based on the goal the policy has to achieve. In this step, coefficients are set together with each reward function.

(Step 2) In a second step, a detailed simulation of the reward functions for the different states is performed to check if the goal states are the right ones (similar to the space exploration shown in Figure 8.1). If the states with higher rewards are not the desired ones, steps 1 and 2 are repeated until the reward functions and coefficients are properly tuned.

(STEP 3) Once the reward functions and states are properly tuned, the system is trained with the previous rewards and coefficients. Note that this process has to be carried in the real scenario, and can take a considerable amount of time. If the obtained results are not the desired ones, all the previous steps are repeated until states, rewards and coefficients are correct. At the same time the system is trained, the table storing $\mathcal{P}'$ is recorded. As described in Section 8.2.2, this table can be used in subsequent learning processes to simulate the environment, instead of training the system in the real system, and therefore, to reduce the time spent on the process.

(STEP 4) Once the definition of states and actions is set, and the table $\mathcal{P}'$ is recorded, obtaining new policies is a trivial process where the reward functions or coefficients are modified, and the system is trained and tested offline.

The main steps of the methodology are depicted in Figure 8.2.

8.2.4. Experimental results for multi-policy resource management

To illustrate our proposal, and following with our driving example, let us consider now a more realistic scenario in which a video provider needs to attend *multiple* video encoding requests from different users *with different requirements*:

- *Regular users* which require a minimum of quality.
- *Premium users* which need encoded videos with maximum quality.

In both cases, real-time results are expected in the encoding process. In this scenario, two different policies are desirable: one which *maximizes quality* for premium users (π^P), and another which *guarantees a minimum quality* for regular users (π^R). Additionally, when the server load increases, it would be desirable to apply different policies to *minimize the resources used by each user*, satisfying a minimum quality to each type of user, and still maintaining real-time throughput (24 FPS) (policies π_P and π_R).

To achieve that, the sub-rewards shown in Figure 8.3 were used to generate the policies: three different reward functions which provide three different levels of quality (R_{PSNR-L}, R_{PSNR-M}, and R_{PSNR-H}), a reward function which guarantees real-time encoding (R_{FPS}), and a reward function which minimizes power (R_{POWER}).

To guarantee a minimum of quality, the functions shown in Figure 8.1 were slightly modified to give a negative reward to those states below the threshold (PSNR < 36). Similarly, the definition of R_{FPS} gives a negative reward to the states below real-time. In this case, a lower reward is given to ensure that, in the case both constrains are violated (quality and throughput), the system will maximize throughput instead of quality, and therefore, achieve real-time encoding processes. The maximum reward of R_{FPS} is not given to 24 FPS, but to the next state, due to the fact that, being close to 24 FPS will produce a higher amount of frames being encoded below the threshold due to the variability on the content between frames.

To estimate the power consumption of each application, the model $P = \text{nth} * (\alpha * \text{freq}^2 + \beta) + \gamma$ was used, experimentally setting the parameters for our

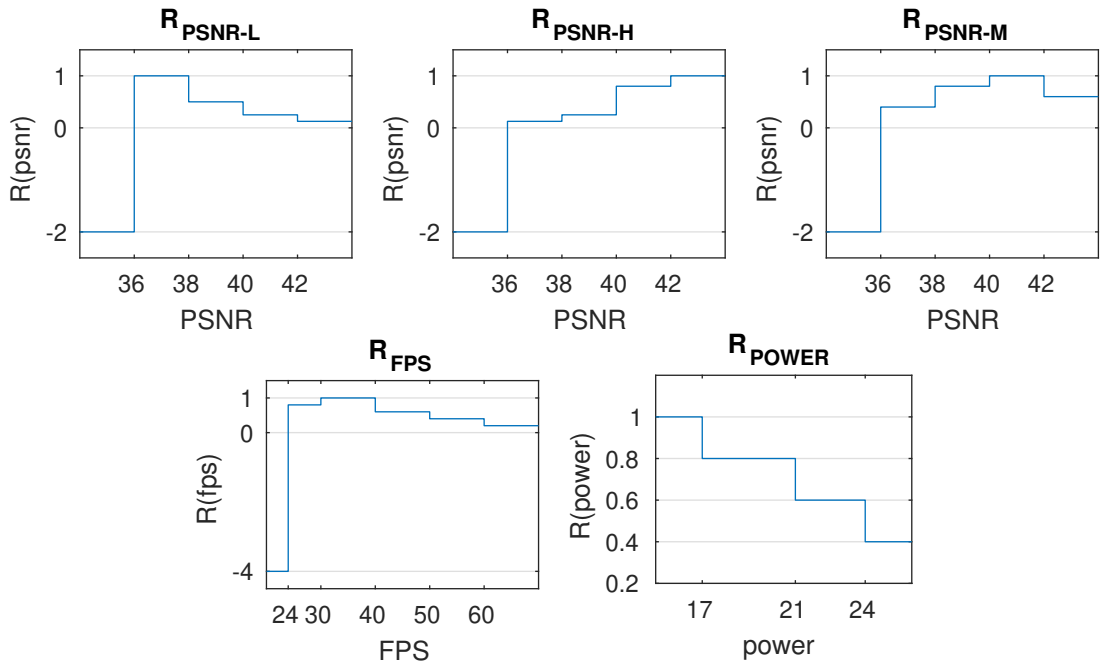


Figure 8.3: Sub-reward functions used for the different policies: three sub-rewards for different levels of quality (top), and the functions for real-time encoding and power (bottom).

$$\pi^R : 0.7 * R_{PSNR-L} + 0.1 * R_{POWER} + 0.5 * R_{FPS}$$
$$\pi^P : 0.7 * R_{PSNR-H} + 0.0 * R_{POWER} + 0.5 * R_{FPS}$$
$$\pi_R : 0.7 * R_{PSNR-L} + 0.5 * R_{POWER} + 0.5 * R_{FPS}$$
$$\pi_P : 0.7 * R_{PSNR-M} + 0.5 * R_{POWER} + 0.5 * R_{FPS}$$

Figure 8.4: Reward functions defined for each policy.

specific platform, obtaining a root mean squared error of 0.97 W and a maximum error of 2.4 W, which are negligible in our machine with a maximum energy consumption of 125 W.

We identify π^R as our base policy, used to tune and polish the actions and state definitions, at the same time the transition table $\mathcal{P}'$ is recorded. Once this policy is created, other policies can be easily derived in a reasonable time by means of the methodology in Section 8.2.3. Our policies, detailed in Figure 8.4, were defined based following a number of considerations:

- The definition of π^R is a combination of a function which minimizes PSNR but ensures a minimum quality (R_{PSNR-L}), and a function which ensures real-time encoding (R_{FPS}). Additionally, a reward to minimize power consumption was incorporated with a small coefficient (R_{POWER}).

		N_{th}		Freq		QP		Quality		Δ	
		π^R	π_R	π^R	π_R	π^R	π_R	π^R	π_R	π^R	π_R
Regular	Hr4	4.2	3.3	1.6	1.7	35.2	36.0	39.4	39.1	6.0	2.4
	Hr5	3.1	2.9	1.4	1.5	37.0	36.9	37.8	37.8	0.4	0.4
	Hr6	3.1	2.6	1.3	1.5	37.0	37.0	38.0	38.0	0.3	0.3
	Hr7	3.3	2.9	1.3	1.4	37.0	37.0	37.1	37.1	0.4	0.4
	avg.	3.4	2.9	1.4	1.5	36.6	36.7	38.1	38.0	1.8	0.9
		π^P	π_P	π^P	π_P	π^P	π_P	π^P	π_P	π^P	π_P
Premium	Hr4	4.6	3.6	1.8	1.6	24.9	33.4	43.7	40.2	2.0	6.3
	Hr5	4.2	3.3	1.7	1.4	24.9	32.2	43.2	40.2	2.3	4.2
	Hr6	4.2	3.1	1.6	1.3	23.7	33.0	43.4	39.9	0.6	0.7
	Hr7	4.7	3.1	1.7	1.3	22.1	33.2	43.2	39.0	0.7	0.5
	avg.	4.4	3.3	1.7	1.4	23.9	33.0	43.4	39.8	1.4	2.9

Table 8.1: Average knob values learned by the system for Regular (top) and Premium (bottom) users with and without resource minimization, and output metrics for the different videos used to validate the system.

- The major difference between π^R and π^P is the reward function used to evaluate quality. The former minimizes quality, and the latter maximizes it (R_{PSNR-H}). To achieve high quality videos without throughput violations, the reward which minimizes power is removed. As real-time encoding is mandatory in both cases, this term is still in place without modifications.

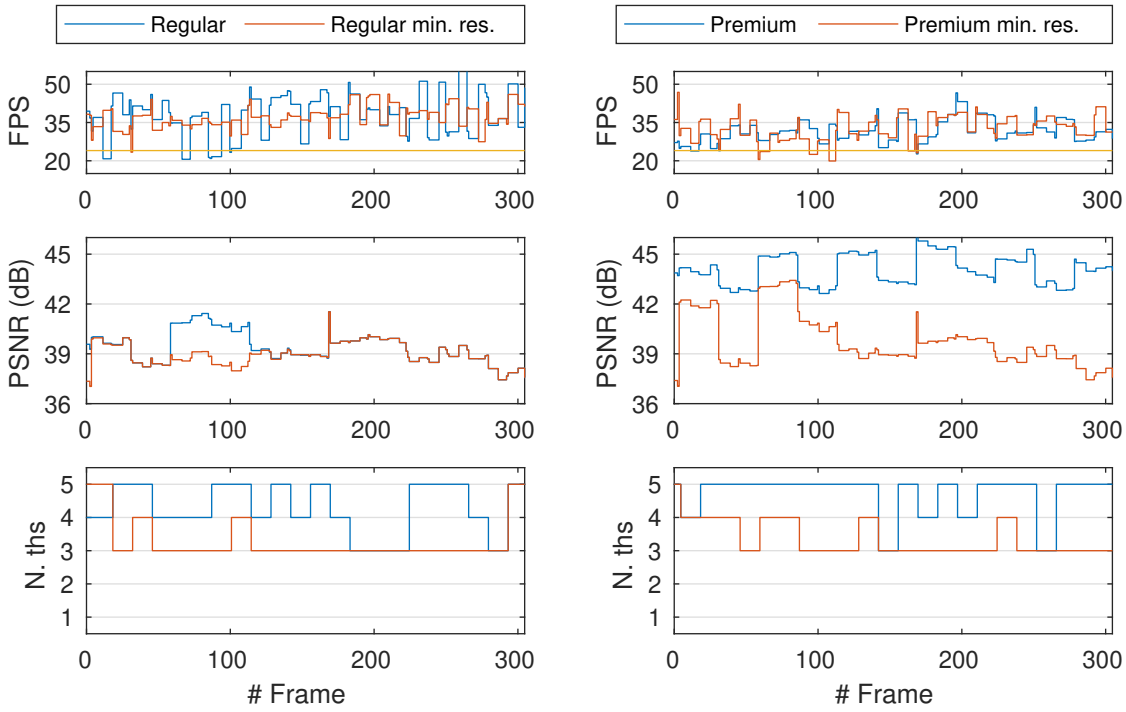
- If resource minimization is desired for a regular user, π_R achieves that goal by increasing the coefficient of the function which minimizes the power consumption respectively to its counterpart π^R (from 0.1 to 0.5).

- The design decisions to create π_P are similar to the ones used to create π_R but in this case, because obtaining high quality videos is a resource-hungry process, the sub-reward function associated with PSNR is modified to still obtain high quality videos, but lower quality than that in π^P (R_{PSNR-M}).

Results for individual policies

The aforementioned policies have been implemented as described in Chapter 7 and under the same conditions. Table 8.1 reports the behaviour of each described policy applied to each video, showing the average knob values set for number of threads (N_{th}), frequency (in GHz) and QP, and the gathered output metrics: quality (PSNR, measured in dB), and real-time throughput violations (measured as the percentage of time the video has been encoded below 24 FPS (-Δ-)).

First, observe how changes in one reward function can produce opposite behaviors. Consider, for example, policies π^R and π^P: by modifying exclusively the reward function in charge of quality, the obtained PSNR changes drastically (38.1 dB to 43.4 dB, respectively).



(a) Regular users **(b)** Premium users

Figure 8.5: System behaviour timelines and metrics obtained when encoding the HR4 sequence with all different policies for Regular users (left) and Premium users (right). The yellow line indicates real-time encoding (24 FPS).

On the contrary, observe how modifications in the coefficients without altering the reward functions can keep the global behaviour of the system intact, but modify the internal actions chosen by the system. Comparing policies π^R and π_R, both achieve comparable quality levels (38.1 dB vs 38.0 dB), but the number of threads used when the policy π_R is acting decreases down to one thread in average (in the case of HR4 sequence) with respect to π^R. Regarding policies π^P and π_P, observe how the impact in the number of threads is the same as in the other policies, but the change in the reward which affects quality produces slightly lower PSNR.

Figure 8.5 shows a timeline of the encoding process of the sequence HR4 under different policies. For the sake of clarity, only changes in number of threads are shown, but dynamic adaptation of frequency and QP values are also in place. In overall, observe how in the case of regular users, both policies π^R and π_R obtain similar (and lower) quality but the number of threads used in each one changes drastically. On the right, both policies obtain higher quality results, but there is a clear difference in quality and resource usage between them as desired in our formulation.

	N. frames	Learning Time
MONO-AGENT	$n \times 3000$	$n \times 17h$
MULTI-AGENT	$n \times 500$	$n \times 3h$
THIS METHODOLOGY	$500 + (n-1) \times 500$	$3h + (n-1) \times 1min$

Table 8.2: Learning time to obtain n different policies by different approaches. The Mono- and Multi-agent approaches are those described in Chapter 6. Learning time has been calculated assuming a learning rate $\approx 24fps$.

Learning time analysis

Following the ideas described in Section 8.2.2 to boost learning time, once the first policy (π^R) was defined, the training time to obtain the remaining policies was reduced from days to hours, as shown in Table 8.2 when compared against the traditional approach described in Chapter 6.

Note that, in this example, learning times were extracted using the same machine and setup; also, observe that our approach inherits the advantages of the multi-agent implementation (in terms of a reduction in the number of necessary frames to converge from 3000 to 500 compared with a mono-agent approach), and adds additional gains as the number of desired policies increases. In this case, adding a new policy is translated into roughly one extra minute of computing time. In the case of the traditional mono- or multi-agent approaches, each new policy would require a complete learning process, adding 17 h and 3 h per policy, respectively.

8.3. Combining multiple policies via heuristics

In the previous section, we have explored how different policies can be extracted from the same formulation of the system, and how, following the proposed methodology, the effort and time needed to produce them can be considerably reduced. In this section, we show a simple and effective way to combine those policies in a realistic scenario, where multiple requests from different types of users arrive distributed over time, varying the number of users to attend simultaneously at each moment. Under this scenario, if the number of simultaneous videos is large enough, it may be impossible to attend all the request simultaneously due to an insufficient amount of resources. In these situations, it is common to enqueue the requests, and attend them in order as the resources are freed by previous clients. Of course, the time a request is hold in the queue does not depend only in the videos being encoded, but also on the type of user that made the request.

To handle this scenario, we propose the use of a *3-tier heuristic* that, based on the policies obtained in the previous section, is able to choose the proper policy to apply to each client and thus reduce the overall waiting time. This heuristic demonstrates how all the ideas described in this thesis can be combined in an unique solution:

- At the application level, a MAL solution is used to tune the different knobs based on application and system metrics.

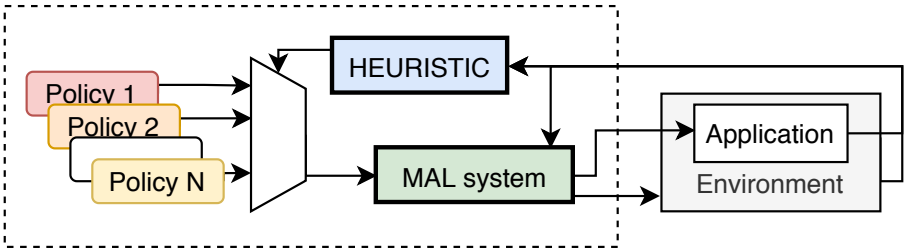


Figure 8.6: System design. The MAL system is in charge of modifying the knobs while the heuristic decides the best policy to apply to each application at each moment.

- At a higher level, the heuristic influences he decision taken by MAL choosing the proper policy to apply.

The policy decision is not randomly taken, but also from the data measured from the system (*occupation*) and the application (*user type*). Figure 8.6 shows how the heuristic and the MAL system are integrated into the system.

Note that this heuristic is just an illustrative example of how a simple approach can benefit from having different policies to apply, obtaining better results than other static approaches. Moreover, the methodology described in the previous sections is also valid for any other approach based on multiple simultaneous policies.

8.3.1. Heuristic design

The proposed 3-tier heuristic is based on a Finite State Machine (FSM) with three different states. Each state represents a different level of occupation, and therefore, a different set of policies to apply to each user type. Algorithm 8.1 shows a detailed pseudocode describing how the heuristic moves between the following states:

1. *S0*, the initial state. This state is active when there are enough resources for all the clients, so that resource usage reduction is not needed. In this state, policies π^R and π^P are applied to regular and premium users, respectively, to produce encoding with maximum quality for both kind of users

2. If there are not enough resources to encode all requests, state *S1* reduces resource consumption of regular users by applying policy π_R. In this state, premium users are still allowed to use as many resources as they require (policy π^P).

3. When there are not enough resources for the incoming request in state *S1*, and only if the incoming request arrives from a *premium* user, the heuristic will move to state *S2*, minimizing the resources for both kind of users, *regular* and *premium*. In this state, policies π_R and π_P are used.

To determine if a client can be attended in a specific state, the function `canRunClient` predicts if it is possible or not based on a estimation of the number of cores each running client is using in the specified state. In order to predict the number of cores in use, the function relies in an internal table storing the average number of cores used by each policy with the training videos, as an estimation value of the cores used by future videos.

Algorithm 8.1: 3TierPol: Determine if a new request can be processed.

```
   Data:
     queue<ClientRequest> queue: clients' requests waiting to be processed
     nReg, NPrem: number of regular and premium users running on the server
     currentState: state the system is
   Result: True if it was possible to attend the request. False in other case.
 1 Function startClient () is:
   |  /* Determine new state                                              */
 2 |  cl ← queue.first;
 3 |  newState ← currentState;
 4 |  if (canRunClient(currentState, cl, NReg, NPrem)) then
 5 |  |  newState ← currentState;
 6 |  else if (currentState = S0 AND canRunClient(S1, cl, NReg, NPrem)) then
 7 |  |  newState ← S1 ;                                  // Try to move to S1
 8 |  else if (cl.type = premium AND canRunClient(S2, cl, NReg, NPrem)) then
 9 |  |  newState ← S2 ;                                  // Try to move to S2
10 |  else
11 |  |  queue.insert_front(cl);                      // Not enough resources
12 |  |  return False;
13 |  end
   |  /* Update state                                                     */
14 |  currentState ← newState;
15 |  if (client.type = regular) then nReg++;
16 |  else nPrem++;
   |  /* Start encoding request                                          */
17 |  return runClient(cl);
18 end
```

Armed with the instantaneous knowledge of the policies that are in use (i.e., the current state), the number of regular and premium users being attended, and the incoming user type, a prediction of the total number of cores in use is calculated.

If the amount of predicted cores is lower or equal to the number of physical cores of the machine, the request is attended and the video encoding can commence. Else, the heuristic tries to move to the next state. If there is not enough room for the user in any state, the request is pushed in front of the queue again. Inserting the client in the front (instead of enqueueing it again in the back) allows to attend the users at arrival order. When a video finishes to be encoded and enough resources are freed, the heuristic checks if it can move to a previous state that does not minimize the resource usage and provides bigger quality.

8.3.2. Experimental results for multi-policy combination heuristic

In order to improve realism, we assume that several different videos from different users arriving over time need to be served simultaneously, minimizing the waiting time of each client and meeting requirements in quality (based on the type of user) and throughput

($\geq$24 FPS). Each experiment is determined by the arrival rate (5s, 10s and 15s), and the percentage of premium users (0%, 25%, 50%, 75% and 100%). Each experiment comprises 10 sequences to be encoded, randomly selected, with a duration of 2500 frames each ($\approx$ 100 seconds at 24 FPS).

Each video is concatenated to itself multiple times producing 2500 frames sequences. To obtain reliable data, each configuration of frequency and premium/total users relation was explored through 5 different combinations of 10 videos, and each combination was run 3 times, reporting average values. Although three different arrival frequencies where explored in the experiments (5s, 10s and 15s), only the results of 10s are shown, although similar and comparable measurements were observed for other frequencies.

For the sake of comparison, we have compared 3TierPol with two modifications of the MAL approach presented in previous chapters. Both alternatives implement a static decision making process, choosing the policy to serve each video based on the kind of user, and not on the environment. The policy utilized does not change during the whole encoding process:

1Pol**:** This strategy corresponds to the MAL approach presented in previous chapters, where only one policy (π^R) is used to encode all the sequences as an extreme case, trading off quality for throughput. Note that in this case, no premium users are considered.

2Pol**:** This approach corresponds with a slightly modified MAL approach. In this case, the system determines between two different tables, which policy apply on the user type to attend (π^R and π^P). This policy is the opposite to the previous one: it offers maximum quality to each type of user, without considering decreasing quality to serve more users simultaneously.

3TierPol**:** The heuristic proposed in this chapter. It uses a heuristic to determine the policy to apply to each client. A MAL instance is in charge to apply the policies.

In the former two cases, the algorithm used to determine whether a video can be encoded or needs to wait on the queue is the same as that used in 3TierPol.

The plot on the top of Figure 8.7 shows the amount of users attended per minute (on average) based on the number of premium user requests for the three explored approaches. Depending on the number of premium users, two different behaviors are observed: when the amount of premium users is below 50 %, and when the amount is greater or equal. On the first group, 3TierPol outperforms the other strategies, and it is able to process more users per minute reducing slightly the quality obtained (0.13 dB for regular and 1.9 dB for premium users in the worst case), see the bottom plot in Figure 8.7.

Diving into details of the behaviour of the second group (premium $\geq$ 50%), we observe how the amount of premium users impacts in the performance of the system: as many premium users are attended, more resources are used to encode these videos, and less resources are available to encode new incoming requests. This behaviour is shown in 2Pol and 3TierPol, but not in 1Pol. In the case of 1Pol, the observed behaviour is to encode all the videos with the same policy (π^R), used by the others approaches to encode only regular users. On the one hand, this approach uses less resources to process each premium

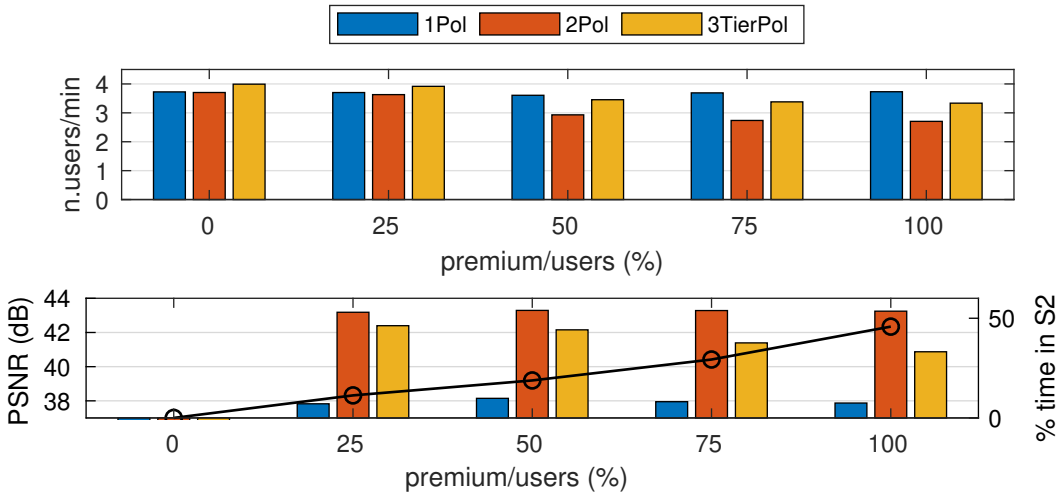


Figure 8.7: Users per minute attended by each approach with users requests arriving every 10 seconds (top), and quality obtained by each approach when encoding videos from premium users (bars), and percentage of time 3TierPol is in the S2 state (line).

user (because they are considered as regular), but on the other hand, the quality obtained (lower plot on the Figure) is quite lower than the other approaches, obtaining not admissible levels for premium users (up to 5.4 less dB in PSNR).

Second, when comparing 2Pol and 3TierPol, we can observe how the latter is able to serve more videos at the same time (up to 1.24×), reducing slightly the obtained quality (a loss of 2.4 dB in the worst case). Finally, observing the PSNR obtained in each experiment (lower plot) we can see the internal behaviour of 3TierPol: as the number of premium users increases, 3TierPol needs to move to states that reduce the resource usage at the expense of reducing the quality. This can be seen in the plot at the bottom of Figure 8.7, showing the percentage of the time 3TierPol is in the *S2* state depending on the number of premium users. Observe how the time 3TierPol is at state *S2* increases with the number of premium users.

8.4. Conclusions

In previous chapters we have shown how Reinforcement Learning can be used to formulate and implement a complex resource manager to apply a global policy to multiple applications simultaneously. However, some scenarios require the application of different policies to each application, based on specific characteristics of the running applications and/or the status of the system at each moment of the execution.

The previous formulation allows to obtain multiple policies for the same scenario, at expenses of a considerable amount of training time, as well as a manual tuning process for the different states and rewards. To reduce this burden, in this chapter we have proposed a new methodology to obtain these policies with minimum effort and with minimum learning time. To do so, our methodology decomposes the design space into different sub-spaces, helping the users to tune the reward functions with minimum effort. In addition, storing the

probability functions relating the different states and actions allows the creation of those policies without needing to re-train the system.

In addition, in this chapter we have shown how a FSM can be built on top of MAL, to dynamically apply different policies to different applications based on the type of the applications and the status of the system. Our solution determines the best policy to apply at each application at each moment, while the MAL system is in charge to apply the selected policy.

9

Conclusions

9.1. Conclusions and main contributions

As described in the motivation of this dissertation (Section 1.1), the evolution of the newest architectures yields complex scenarios where performance is not the sole optimization target, but a conjunction of conjunction of multiple system- and application-wide metrics needs to be considered to manage them. To achieve these goals, modern solutions need to consider different application- and system-wide metrics as a *holistic* solution, determining the best knob configuration at each moment, taking into account the effects of each knob in the output metrics, as well as the possible dependencies among knobs and across applications. However, the management of multiple knobs targeting multi-objective optimization goals in a multi-application scenario is far from being trivial, and hardly manageable by traditional approaches.

In this scenario, the main goal of this thesis was the *"design, implementation and validation of different resource management strategies for power-constrained multi-application scenarios, able to deal with a multi-objective optimization goal combining application- and system-metrics, dynamically tuning application- and system-knobs"*.

Targeting this objective, three different approaches were described targeting different scenarios, all of them with the previous goal as a common objective. The main contributions of this work can be summarized as:

- We have shown how resource managers can leverage internal information of the running applications and runtimes to properly orchestrate the distribution of shared resources across applications, as well as the correct dynamic configuration of the different knobs.

- We have shown how intra- and inter-application dependency management can be integrated into the resource manager formulation. In addition, we have shown how resource managers can tune different knobs simultaneously, each affecting the same output metrics.

- We have demonstrated how software-based power-capping techniques are as effective as other equivalent hardware-based approaches. However, unlike hardware-based techniques, our approaches are able to optimize other metrics at the same time as a power-cap is applied due to the online sensing of multiple application- and system-metrics.
- We have tested all the approaches in real platforms using applications widely used in the industry simulating real-world scenarios. Running all the experiments on real platforms, instead of simulated scenarios, guarantees that our approaches are perfectly valid for real scenarios.

In addition to this general contributions, each proposal explores a different scenario, and therefore, each of them has some specific contributions:

Targeting asymmetric platforms

A full of set of policies for task-based parallel application have been developed targeting energy-efficiency on asymmetric architectures. The policies explore different frequency scaling and scheduling techniques for both clusters in the platform. All the proposals rely on the internal classification of the tasks among in critical and non-critical, and they are completely agnostic to the running applications.

A complete implementation and evaluation of the different proposed policies has been developed on top of the Nanos++ runtime, and performed in different asymmetric platforms. The QR and Cholesky factorization were used to carry out the experiments, as they are representative examples of dense linear algebra operations, widely used in multiple scientific fields. Experimental results reveal how, taking advantage of the classifications of the different tasks and their relation, our approach increases the energy efficiency consistently in all the tested applications and platforms.

Targeting power-limited scenarios on modern platforms

Another contribution of this dissertation was the development of bar and baco to increase the performance on power-limited scenarios. Given a limited amount of power budget to one application, bar performs a dynamic redistribution of the budget between threads in terms of frequency adaptation of the running cores. bar was built on top of the Nanos++ approach, and bases its decisions in the status of the running threads. Similarly to the previous approach, the developed approach relies only in the internal information extracted from the runtimes and not from the applications, being totally agnostics to them. baco extends the previous approach dealing with scenarios with more than one application running simultaneously. Based on the status of each runtime at each moment, baco performs a dynamic redistribution of the power budget assigned to each application, managed by the bar instance.

Several contributions are derived from the implementation of bar and baco. As for the bar implementation, optimal results were achieved in traditional scenarios, outperforming the default configurations used in Nanos++. Our approach was proved to work on scenarios with a non-uniform distribution of the power budget, contrary to other hardware-based approaches that are not able to distinguish the running application, and therefore,

to perform an application-specific policy. BACO was tested on multiple random scenarios, achieving optimal results with negligible or no overhead.

These proposals show how software-based power-capping can be used to manage power-limited scenarios obtaining optimal results thanks to a constant monitoring of the runtimes. The use of runtime metrics, as well as domain-specific power models makes the contribution agnostic to the running applications, and therefore, valid for any application.

Targeting QoS-aware scenarios

On scenarios with application-specific metrics, a complex approach was developed to combine application- and system-metrics in a unique multi-objective goal, dealing with application- and system-knobs in a jointly manner. Our approach was based on the use of Reinforcement Learning to automatically tune the available knobs. A specific model for the applications was developed targeting multi-user real-time encoding scenarios with tight limits in throughput, quality and power consumption. The use of Q-Learning to formulate resource managers has been proved as a promising alternative to manage complex scenarios with complex dependencies between multiple knobs and metrics.

Our solution has demonstrated how application- and system-knobs can dynamically be tuned to achieve high-quality encoding processes never violating the real-time requirements imposed, by an autonomous classification of the different knobs based on their impact on the output metrics.

In addition, a methodology was proposed to create multiple policies with minimum effort and no overhead on the learning process. A heuristic solution was built to deliver these different policies to multiple clients based on the type of application, as well as the status of the system as each moment of the execution. The proposed solution demonstrated how different policies can be applied simultaneously to different applications, while pursuing a common goal.

9.2. Related publications

The contributions of this thesis are supported by the publication of multiple articles in different peer-reviewed international conferences and journals. Publications are classified in directly and indirectly-related to the content of the dissertation.

9.2.1. Directly related publications

Costero, L., Iranfar, A., Zapater, M., Igual, F. D., Olcoz, K., and Atienza, D. Resource Management for Power-Constrained HEVC Transcoding Using Reinforcement Learning. *IEEE Transactions on Parallel and Distributed Systems 31*, 12 (2020), 2834–2850 — JOURNAL

Costero, L., Igual, F. D., Olcoz, K., and Tirado, F. Leveraging knowledge-as-a-service (KaaS) for QoS-aware resource management in multi-user video transcoding. *The Journal of Supercomputing* (2020) — JOURNAL

Costero, L., Igual, F. D., Olcoz, K., and Tirado, F. Providing On-Demand Quality & Resources for Malleable Applications. In *International Conference Computational and Mathematical Methods in Science and Engineering - CMMSE* (2019) — CONFERENCE PROCEEDINGS

CONFERENCE PROCEEDINGS

COSTERO, L., IRANFAR, A., ZAPATER, M., IGUAL, F. D., OLCOZ, K., AND ATIENZA, D. MAMUT: Multi-Agent Reinforcement Learning for Efficient Real-Time Multi-User Video Transcoding. In *Design, Automation and Test in Europe Conference and Exhibition - DATE* (2019)

CONFERENCE PROCEEDINGS

COSTERO, L., IGUAL, F. D., OLCOZ, K., AND TIRADO, F. Energy Efficiency Optimization of Task-Parallel Codes on Asymmetric Architectures. In *International Conference on High Performance Computing Simulation - HPCS* (2017)

CONFERENCE PROCEEDINGS

COSTERO, L., IGUAL, F. D., OLCOZ, K., AND TIRADO, F. Modifying OmpSs/Nanox to exploit energy efficiency in asymmetric architectures for the Cholesky factorization. In *HiPEAC Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems - ACACES* (2017)

9.2.2. Indirectly related publications

JOURNAL

CORPAS, A., COSTERO, L., BOTELLA, G., IGUAL, F. D., GARCÍA, C., AND RODRÍGUEZ, M. Acceleration and energy consumption optimization in cascading classifiers for face detection on low-cost ARM big. LITTLE asymmetric architectures. *International Journal of Circuit Theory and Applications 46*, 9 (2018), 1756–1776

JOURNAL

COSTERO, L., IGUAL, F. D., OLCOZ, K., CATALÁN, S., RODRÍGUEZ-SÁNCHEZ, R., AND QUINTANA-ORTÍ, E. S. Revisiting conventional task schedulers to exploit asymmetry in multi-core architectures for dense linear algebra operations. *Parallel Computing 68* (2017), 59 – 76

9.3. Open research lines

The research developed in this Ph.D. thesis has addressed the design and development of resource managers combining multiple application- and system-metrics while tuning both application- and system-knobs, targeting energy-efficiency. The techniques proposed have tackled different scenarios, from asymmetric platforms with only one application running, to power-limited modern servers with multiple applications running simultaneously. However, some interesting points of future research have emerged during the evolution of this work.

Some of the open research lines are detailed next:

- The use of heterogeneous platforms is widely extended in the HPC field nowadays, being fully supported by most of the current task-based runtimes. As this kind of platforms offers a plethora of new knobs to tune and metrics to optimize, our proposals can be extended to support this kind of platforms.

- The learning process can be improved by a profiling of the different tasks of the applications, performing an automatic classification of them to improve the scheduling algorithms.

- Furthermore, a proper characterization of the different task can be used on heterogeneous platforms to perform an optimal assignation of the tasks to the different compute devices. Autonomous learning techniques can be applied to this scenario.
- Our MAL approach is based on an extension of the Q-Learning algorithm to support a cooperative multi-agent behaviour. Other reinforcement learning algorithms can be used to train the system and compare the quality of the learned policies. Specifically, neural networks are highly used nowadays to train this kind of systems due to the emergence of hardware accelerators speeding up the learning and inference process.
- The extension of our proposals to support multi-application scenarios in cloud systems, managing the resources at the cloud level instead of node level. A more complex approach can be developed comprising two different layers, one working at the cloud level, and the other managing the resources inside each node, both cooperating to achieve optimal results.
- Other applications a part of dense linear algebra kernels and video encoding can be used to validate our proposal, as well as other platforms implementing other micro-architectures or system organizations.

A

Platform description

A.1. Hardware description

All the experiments presented on this thesis have been carried out in three different platforms: multi-core 32- and 64-bits ARM big.LITTLE boards (ODROID and JUNO), and a modern Intel Xeon server (MAKALU). Those systems are a representative example of the different architectures present nowadays, so most of the results extracted on these platforms can be extrapolated to other platforms without loss of generality.

ODROID: ODROID refers to the ODROID-XU3 board manufactured by Hardkernel Co. Ltd.[1]. This board comprises a Samsung Exynos 5422 SoC featuring a 32-bit ARM big.LITTLE processor and a 2 GB DDR3 RAM. The chip features an ARM Cortex-A15 quad-core processing cluster (big cluster) and a Cortex-A7 quad-core processing cluster (LITTLE cluster). Both types of cores implement an ARMv7a architecture, with an in-order pipeline in Cortex-A7, and an out-of-order pipeline in Cortex-A15.

Each ARM core (either Cortex-A15 or Cortex-A7) has a 32+32KByte L1 (instructions + data) cache. The four A15 cores share a 2-MByte L2 cache, while the four A7 cores share a smaller 512-KByte L2 cache. All cores of the same cluster share the same frequency of operation, clocking from 800 MHz to 1300 MHz in steps of 100 MHz in both cases. This board exposes independent power measurements for each cluster.

JUNO: JUNO ARM DEVELOPMENT PLATFORM, developed by ARM, is the first ARMv8-a 64-bit platform featuring a Cortex-A57 dual-core processing cluster (big cluster) and a Cortes-A53 quad-core processing cluster (LITTLE cluster).

Big cluster clocks up to 1.1 GHz, while frequency of the LITTLE cluster is limited up to 800 MHz. Cortex-A57 implements a 15-step out-of-order pipeline, and Cortex-A53 implements a 8-step in-order pipeline. Big and LITTLE clusters have an L2 cache of 2 MByte and 1 MByte respectively. Each Cortex-A53 has a 32 KByte L1 cache (instructions and data share same cache space), while each Cortex-A57 is accompanied

[1] https://www.hardkernel.com/shop/odroid-xu3/

Software package	ODROID	JUNO	MAKALU
OS/kernel	3.10.51+	3.10.63	4.9.0-11-amd64 (Debian 9)
GCC/G++ Compiler	4.8	4.9.1	6.3.0
NANOS++	0.10a	0.10a	0.15a
Mercurium	2.0.0	2.0.0	2.3.0
Extrae	3.2.1	3.2.1	3.6.1
BLIS	0.1.8	0.1.8	-
Intel MKL	-	-	2018.1.163
Kvazaar	-	-	2018-07-18

Table A.1: Software version configured in each platform.

by a 48+32KByte L1 cache (instructions + data). This board offers physical *shunt* resistors exposing independent power measurements for each cluster.

MAKALU: MAKALU refers to a modern Intel server comprising two 20-core Intel Xeon Gold 6138 CPU and 128 GB of DDR4 RAM. Per-core DVFS ranges from 1.00 GHz to 2.00 GHz and turbo frequency. Turbo frequency ranges from 2.3 GHz to 3.7 GHz depending on the number of cores and the instruction set used (normal, AVX2 or AVX512). Maximum power consumption is limited by a TDP of 125 W.

Each core has a 32+32KBytes L1 cache (instructions + data) and an independent 1 MByte L2 cache. A 28 MByte L3 cache is shared between cores. Hyperthreading was disabled in all the experiments.

A.2. Software description

The software used in our work can be divided in two different groups, each related with a different part of this document. All the runtime-based experiments carried out in Part I were built on top of the OmpSs programming model [56], comprising the NANOS++ runtime [127], and its associated Mercurium compiler [120]. Linear algebra kernels were executed using BLIS [183, 182] and Intel MKL [92] libraries in ARM and Intel platforms respectively. Offline analysis was carried out using Extrae [60] and Paraver [140] software.

For the experiments shown in Part II, Kvazaar [186] video encoder was used. Energy measurements have been done through RAPL mechanism, using the PAPI [177] library when on-line measurements were needed, and Pmlib [13] for external application measurements. Table A.1 summarizes all the software and system versions configured in each platform.

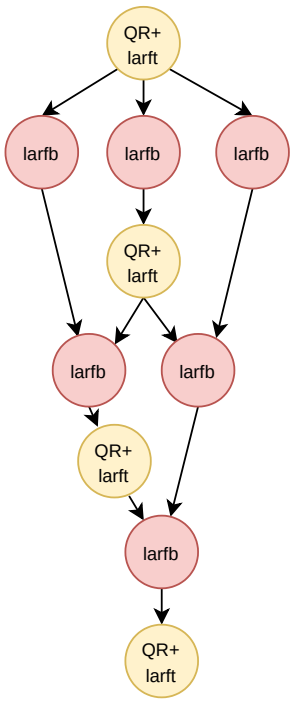
A.2.1. Dataset definition

Dense linear algebra operations

The Cholesky and QR factorizations were chosen to carry out all the experiment related with runtime-based proposals (Part I). These operations in particular are a representative example of many other factorizations used to solve linear equation systems, so many of

```
1  void qr_blocked(int m, int n, int t) {
2   //m=n. rows; n=n.columns; t=tileSize
3
4   int nt = (n + t - 1) / t;
5   int nr = nt * t;
6   int mr = m < nr?nr:m;
7   int tt = t * t;
8   int mt = mr * t;
9
10  // [...]
11  for ( int j = 0 ; j < nt ; j++ ) {
12   int skip = j*t;
13
14   double *Ab = &A[mt*j];
15   double *Sb = &S[j*tt];
16
17   //QR fact. & triang. fact. of block reflect.
18   ompss__dgeqr2_dlarft( skip, mr, t, Ab, Sb);
19
20   int jj;
21   for (jj = j+1 ; jj < nt ; jj ++) {
22    double *Aupd = &A[mt * jj];
23    //Block reflector
24    ompss__dlarfb( skip, mr, t, Ab, Sb, Aupd);
25   }
26  }
27 }
```

(a) Simplified C version of a blocked QR factorization



(b) Diagram dependencies

Figure A.1: Blocked QR factorization.

the conclusions done for these factorizations can be easily extrapolate to other similar applications widely used in science and engineering.

Cholesky factorization: The Cholesky factorization [74] decomposes a symmetric positive-definite matrix A in the product $A = U^T U$, where U is an upper triangular matrix. This routine block-oriented decompose the main operation in a set of kernels: Cholesky factorization (`potrf`), linear triangular system resolution (`trsm`), matrix multiplication (`gemm`), and symmetric rank-b update (`syrk`), operating on the different blocks, and implemented on the LAPACK [112] and BLAS [24] standards. Section 3.1 shows a simplified C-implementation of this kernel.

QR factorization: Similar to the Cholesky factorization, the QR factorization [74] (QR decomposition or QU factorization) is widely used in the science and engineering fields, and it is a representative example of many other DLA operations. Given a matrix A, the QR factorization decompose the matrix in the form $A = QR$ where Q is an orthogonal matrix and R an upper triangular matrix.

The code in Figure A.1a shows a simplified C-like implementation for the factorization of matrix A of size $m \times n$, divided in blocks of size t. The order in which the different kernels are invoked during the execution, but also the sub-matrices each kernel reads and writes, generate a DAG showing the dependencies between tasks (kernel instances), and therefore, the potential parallelism between tasks. For instance, Figure A.1b corresponds

```
#pragma omp task inout( A[0;m*n] ) output( Q[0;n*n] )
void ompss__dgeqr2_dlarft( int skip, int m, int n, double *A, double *Q)
{
  //[...]
  A = A + skip;
  int Adim = m - skip;

  dgeqrf_( &Adim, &n, A, &m, tau, w, &n, &info ); //QR factorization
  dlarft_("Forward", "Columnwise",
          &Adim, &n, A, &m,
          tau, Q, &n);       //Triangular factor of a block refector
  //[...]
}


#pragma omp task input( A[0;m*n], T[0;n*n] ) inout( Aupd[0;m*n] )
void ompss__dlarfb(int skip, int m, int n, double *A, double *T, double *Aupd)
{
  //[...]
  Aupd += skip;
  A += skip;
  int Adim = m - skip;

  dlarfb_("Left","Transpose","Forward","Columnwise",
          &Adim, &n, &n, A, &m, T, &n, Aupd, &m,
          LDWORK, &n);         //Block reflector to a general matrix
  //[...]
}
```

Listing A.1: Parallel implementation of a QR Factorization in OmpSs.

to the DAG of the previous code when executing a factorization on a matrix divided into 4×4 sub-matrices, showing the tasks (nodes) and dependencies (edges).

The code on Listing A.1 shows the annotations needed to exploit the task parallelism of a QR block-based factorization in OmpSs (`#pragma` lines). Note how clauses `input`, `output` and `inout` denote the direction of the data dependencies, and help the runtime to keep track of the dependencies between tasks during runtime. In this implementation, the basic kernels are implemented as calls to external libraries offering *dgeqrf* (general QR factorization), *dlarft* (triangular factor of a block reflector) and *dlarfb* (block reflector to a general matrix) application programming interfaces (APIs).

Video sequences

All the experiments covering video encoding processes were carried out using the video sequences proposed by the JCT-VC committee [28], as they offer highly variability in the contents of the videos. The same sequences were used in two different resolutions: High resolution (720p/HD, 1280×720 pixels) and Low resolution (832×480 pixels), obtaining the low resolution sequences from rescalating the high resolution ones. Specifically, the following sequences were chosen: *FourPeople*, *KristenAndSara*, *OldTownCross*, *QuarteBackSneak1*, *BT709Parakeets*, *Johnny* and *ThreePeople*. A detailed description of these sequences can be found in Section 6.3.2.

B

Centralized Resource Manager

All the ideas proposed in this thesis and the experimental results shown orbit around a centralized resource manager implemented from scratch. This centralized resource manager is in charge of collecting all the application- and system-metrics continuously, and select the most appropriate application and system-knobs. Application metrics are received from the clients through a provided library, while system metrics are measured directly from the system using different mechanisms. Similarly, application knob configurations are transmitted to applications through the same library, while system knob configurations are applied directly by the resource manager. Indirectly, this design implies that applications have an active role in the management process, being responsible to send the metrics and apply the knob configuration received. To mitigate the possible complexities of this process, all the interactions are encapsulated into a library, providing developers a non-intrusive and easy method to integrate their applications into the system.

The general design of the system is a client/server infrastructure, where the multiple running applications (clients) register into the resource manager (server) and communicate with it sending the different metrics and receiving the different knob configurations. The design and implementation of the system tries to be the most generic and flexible as possible, at the same time performance considerations are taken into account to minimize the introduced overhead, and therefore, to support real-time process management (as online video encoding).

B.1. Client design

As explained before, and trying to support as many application as possible with a minimum effort by the application developers, the integration of the applications into the resource manager is done through an external C library encapsulating all the logic and communications with the resource manager. The design of the library, similar to the rest of the system, follows a twofold objective: be as most generic as possible, and minimize the overhead introduced into the applications.

Trying to be the most *generic* as possible, the library provides functions to synchronize with the server (start and end point), send metric values to the resource manager and receive knob configurations from it. However, the measurement of the metrics, as well as the application of the knob configurations received from the server, is not specified by our design, being each application responsible to implement these actions. In addition, if the applications are run on top of a runtime (like OmpSs or OpenMP), the integration of this library into the runtime allow the execution of the applications with our resource manager without any modification. Indeed, this was the strategy followed in Part I.

Trying to maximize the *performance* of the applications, the communication protocol (described later) supports synchronous and asynchronous messages, and assumes that all the communications are started on the client side. On one side, this fact avoids clients to constantly check if there is a new message from the resource manager. On the other side, this design implies that the resource manager cannot change an application-knob at any moment, but only when a client asks if there is a new configuration to apply (for example, every frame in the Kvazaar implementation used in Chapter 6).

The code in Listing B.1 shows the API exposed by the library to the applications. This API offers all the functionality needed to synchronize the application with the resource manager (`SS_initLibrary`, `SS_startExecution` and `SS_finishLibrary`), send metric values (`SS_updateMetric`) and receive knob configurations (`SS_receiveKnobConfiguration`).

```
1  /* Synchronization operations */
2  int SS_initLibrary( int nOpts, int* optionIds, float* values );
3  int SS_startExecution( void ); //Blocks until permission is granted
4  int SS_finishLibrary( void );  //Async.
5
6  /* Dynamic resource management */
7  int SS_updateMetric( int metricId, float value ); //Async.
8  int SS_updateMetrics( int nValues, int* metricIds, float* values ); //Async.
9
10 int SS_receiveKnobConfiguration( int knobId, float* value );
11 int SS_receiveKnobConfigurations( int nKnobs, int* knobIds, float* values );
```

Listing B.1: Functions exposed to the applications by the library.

Specifically:

`SS_initLibrary`**:** This function call initializes the internal structures of the library and starts a communication channel with the server. Additionally, the server is notified about the application, creating the control structures needed on its side. Internally, this function call sends to the server the PID of the applications, the timestamp, and other additional information. All this information is gathered and sent automatically by the library, without further intervention of the application developer. This function call also allows the programmer to send additional information in the form `[key, value]` pairs. In our experiments, this was used to sent the resolution of the video being encoded, among others.

`SS_startExecution`**:** This function has to be called before the beginning of the execution. The application is blocked into this call until the resource manager grants the application to start its execution. This function allows the resource manager to control

how many applications can run simultaneously, being able to delay the execution of some of them if there are not enough available resources.

`SS_finishLibrary`: Notifies the server about the end of the execution, and frees all the resources reserved by the library. The communication is asynchronously, meaning that application does not wait to the server to receive the message, but exists immediately after sending it.

`SS_updateMetric`: Sends the value associated to a specific metric to the servers. This is an asynchronous operation (i.e., the application does not wait for confirmation from the server). A similar function is provided to submit more than one metric in the same function call.

`SS_receiveKnobConfiguration`: Asks the resource manager for the value of a specific application knob. The application is blocked in this call until an answer from the server is received. If there is not any modification on the knob configuration, the server answers with the previous value. Similar to the previous function, a similar version to ask for multiple knob configurations is provided.

B.2. Communication protocol

Interprocess communication mechanisms

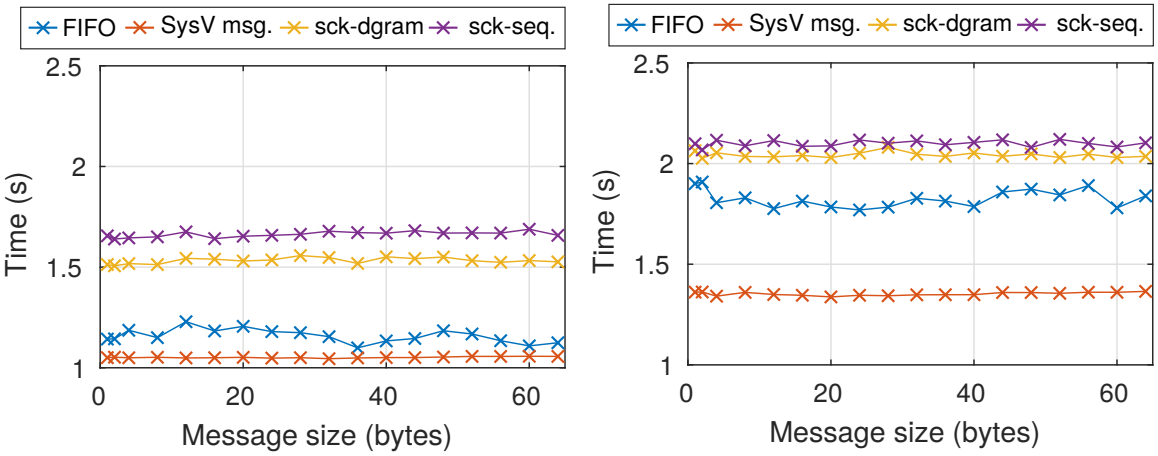
Modern operating systems offer a plethora of different solutions for communicating different processes in the same machine (inter-process communication mechanisms), each one with some advantages and disadvantages over the others. For our purpose, the communication between the applications and the resource manager, the method chosen has to fulfill two main characteristics: (I) it has to be generic and extensible, allowing to add new kinds of messages easily in the future if needed, and (II) it has to be fast enough in order to not produce an overhead in the execution time. In the following, we offer a brief comparison between some of the solutions which satisfy this requirements. An exhaustive comparison of the different methods can be found in [91, 106, 200].

Figure B.1 collects the different experiments carried out for evaluation the different chosen mechanisms, namely, named pipes (FIFOs), Unix Sockets using datagram and sequential packets, and System V message queues. Although all the tested mechanisms are limited to Linux systems (our target platform), the chosen mechanism is hidden to the application by the provided library, being possible to modify it without changes in the applications.

The plots show the average time taken to submit a batch of 1 000 000 messages between two processes for different sizes. Each value represents the average value for 10 executions on MAKALU (see Section A.1). On the left it is shown the results when both processes (sender and receiver) are running in the same processor; on the right if they are executed in different sockets.

From the results shown in the Figure, a set of general remarks can be extracted:

1. Independently of the allocation of the processes (same or different processor), the relative performance between all of the mechanisms keeps constant, being the System V



(a) Producer and consumer on the same processor

(b) Producer and consumer on different processors

Figure B.1: Average time taken by different inter-process mechanism when sending 1 000 000 messages of different sizes, where producer and consumer are in the same (left) or different (right) processors, on MAKALU.

message queues the fastest mechanism in our platform followed by named pipes. The slowest mechanisms are socket communications, having similar results.

2. The size of the message does not affect to the transmission time when the size is small. No greater sizes were tested as the messages sent in our design are not bigger than those.

3. Allocating the processes into the same processor or on different processors increases the transmission time about $1.3\times - 1.6\times$.

In view of these results, the method chosen for the communication between both sides of the framework is the System V message queues mechanism. This mechanism does not only allow a fast communication, but also, thankfully to the API provided by the system, provides natively a generic and easy way for managing different kind of messages (each one identified by a type and an user defined content) at the same time allow synchronous and asynchronous communication, simplifying enormously the communication process.

Message description

All the communications between the applications and the resource manager are done though multiple *channels* (called queues in the System V terminology). This channels are used to share information in a sequential way (i.e., messages are delivered in order), storing the messages until they are read by one extreme of the channel, and therefore, allowing asynchronous communication between both parts. In our design, each channel is composed by two different queues, one for each direction of the communication.

By default, a public channel is created to register the applications into the resource manager. When an application notifies the system about its existence (through the

Function call	Sync./Async.	Msg. size
`initLibrary`	Sync.	104 Bytes
answer:	Sync.	12 Bytes
`startExecution`	Sync.	16 Bytes
answer:	Sync.	8 Bytes
`updateMetric`	Async.	24 Bytes
`receiveKnobConf`	Sync.	12 Bytes
answer:	Sync.	16 Bytes
`finishLibrary`	Async.	16 Bytes

(a) Messages exchanged between applications and the resource manager.

```
1  struct msg_registration_t
2  {
3   long type;
4   unsigned long long
       timestamp;
5
6   int clientId;
7
8   int nOps;
9   int opsIds[MAX_OPS];
10  float values[MAX_OPS];
11 }
```

(b) Actual content sent by `SS_initLibrary` call

Figure B.2: Messages sent between applications and resource manager. On the left, a detailed list of the function calls and the associated messages showing the type of communication (synchronous or asynchronous) and the number of bytes exchanged. On the right, an example of the actual implementation of one of those messages (the associated with the `SS_initLibrary` call).

`SS_initLibrary` call), a new private channel is opened for future private communication between the application and server. As soon as all the initialization has been done in the server side, a confirmation message is sent to the client with the information of the new private communication channel to be used for future communications. This information is automatically processed by the library, which opens and sets the new communication channel without any further intervention by the application. Having a private channel for each client allows the server to attend all the clients in a concurrent way, avoiding delays on one client due to a high amount of messages of another client which can potentially saturate the channel.

Table on the left of Figure B.2 shows a detailed list of all the messages exchanged between the applications and the resource manager, grouped by the function call which triggers these messages. To allow fastest communications, the size of each message was reduced at minimum as detailed in the table. At implementation level, each message is coded like a C struct with a mandatory header indicating the type of communication (`long type` field), and an optional field containing the timestamp of the message. The body of the message will depend on the type of the message exchanged, being even possible to be empty. On the right of Figure B.2 is shown the actual implementation of the message sent when an application registers into the system (triggered by the `SS_initLibrary` call).

B.3. Centralized Resource Manager (server)

The centralized resource manager is a modular multi-threaded C++ application in charge of managing the communications with the running applications (clients), sensoring and tuning the underlying platform, and making the decisions based on the metrics received from the clients and the values measured from the platform. This framework com-

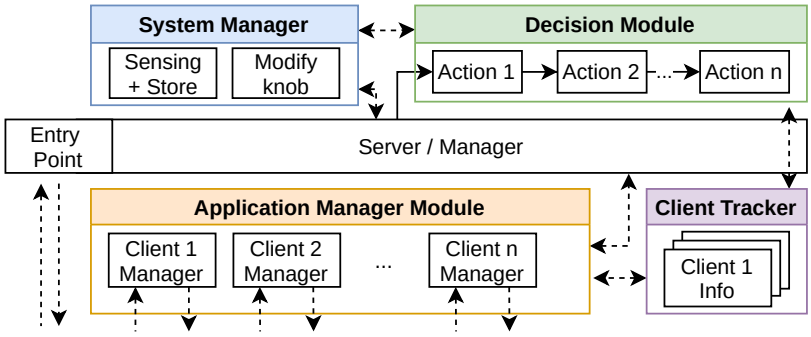


Figure B.3: System overview.

prises four different modules: a highly tunable and extensible module in charge of choosing the best values of each knob (decision module), and other three modules in charge of helping the first one in the decision process (system manager, applications manager and client tracker).

Figure B.3 shows an overview of the different modules and the way they interact:

System Manager: Performs a periodic and configurable sensing of the underlying system metrics, storing them for future query by the decision module. In addition, it provides functionality for tuning all the hardware-knobs. The access to the system-metrics, as well as the modification of the hardware-knobs, is provided in a thread-safe manner. In the experiments shown in this thesis, this module was configured to sense power consumption via the RAPL component in PAPI, set the frequency via the `libcpufreq` library or set the application-to-core affinity by means of POSIX Thread Affinity API calls, among others.

Applications management: Multi-threaded module in charge of communicating with each running client and addressing each message accordingly to its content. For each registered client a new thread is deployed and attached to the private channel communication created on the initialization. This configuration allows to attend multiple clients at the same time avoiding contention problems on the server side.

Client Tracker: This module keeps track of the metrics received from each client, as well the current knob values and other information needed by the decision module. Similar to the System Manager module, offers a thread-safe way to access the information, as well as different functionalities to access the stored data (as getting average values, apply a moving window function to the stored values or access directly to the raw values).

Decision Module: Selects the most appropriate knob configuration to be applied to each registered application based on both application metrics and platform status gathered by the previous modules, applying predefined techniques (e.g., multi-agent Q-Learning, mono-agent Q-Learning, ad-hoc heuristics, etc.). This module comprises a set of plugins called actions, that are called in a specific order. Each action is invoked every time an event in the system occurs (i.e., a message from one client is received).

All the actions follow the same API, providing an efficient and easy mechanism to extend the logic of the resource manager by the users.

In the case of this thesis, the different approaches proposed were implemented changing the actions used in this module, not needed to modify any other module in the system.

References

[1] Agullo, E., Beaumont, O., Eyraud-Dubois, L., Herrmann, J., Kumar, S., Marchal, L., and Thibault, S. Bridging the Gap between Performance and Bounds of Cholesky Factorization on Heterogeneous Platforms. In *Heterogeneity in Computing Workshop - HCW* (2015).

[2] Agullo, E., Beaumont, O., Eyraud-Dubois, L., and Kumar, S. Are Static Schedules so Bad? A Case Study on Cholesky Factorization. In *IEEE International Parallel & Distributed Processing Symposium - IPDPS* (2016).

[3] Ahn, Y. J., Hwang, T. J., Sim, D. G., and Han, W. J. Complexity model based load-balancing algorithm for parallel tools of HEVC. In *IEEE International Conference on Visual Communications and Image Processing - VCIP* (2013).

[4] Ahvar, E., Ahvar, S., Mann, Z. A., Crespi, N., Garcia-Alfaro, J., and Glitho, R. CACEV: A Cost and Carbon Emission-Efficient Virtual Machine Placement Method for Green Distributed Clouds. In *IEEE International Conference on Services Computing - SCC* (2016).

[5] Altman, E. *Constrained Markov Decision Processes*. CRC Press, 1999.

[6] AMD. AMD PowerNow! Technology. Available at: `http://www.amd.com/us/products/technologies/amd-powernow-technology/Pages/amd-powernow-technology.aspx`.

[7] AMD. Cool'n'Quiet Technology Installation, 2004. Available at: `https://web.archive.org/web/20100415060544/http://www.amd.com/us-en/assets/content_type/DownloadableAssets/Cool_N_Quiet_Installation_Guide3.pdf`.

[8] Angelini, C. Next-Gen Video Encoding: x265 Tackles HEVC/H.265. *Tom's Hardware* (2013). Available at: `https://www.tomshardware.com/reviews/x265-hevc-encoder,3565.html`.

[9] Arroba, P., Moya, J. M., Ayala, J. L., and Buyya, R. Dynamic Voltage and Frequency Scaling-aware dynamic consolidation of virtual machines for energy efficient cloud data centers. *Concurrency and Computation: Practice and Experience 29*, 10 (2017), e4067.

[10] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience 23*, 2 (2011), 187–198.

[11] Ayguade, E., Badia, R., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J., and Quintana-Ortí, E. S. A proposal to extend the OpenMP tasking model for heterogeneous architectures. *Lecture Notes in Computer Science 5568* (2009), 154–167.

[12] Badia, R. M., Herrero, J. R., Labarta, J., Pérez, J. M., Quintana-Ortí, E. S., and Quintana-Ortí, G. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience 21*, 18 (2009), 2438–2456.

[13] Barreda, M., Barrachina Mir, S., Catalán, S., Dolz, M. F., Fabregat, G., Mayo, R., and Orti, E. An integrated framework for power-performance analysis of parallel scientific applications. In *International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies - ENERGY* (2013).

[14] Barrett, E., Howley, E., and Duggan, J. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience 25*, 12 (2013), 1656–1674.

[15] Barroso, L. A. The price of performance: An economic case for chip multiprocessing. *Queue 3*, 7 (2005), 48–53.

[16] Bartolini, A., Cacciari, M., Tilli, A., and Benini, L. Thermal and Energy Management of High-Performance Multicores: Distributed and Self-Calibrating Model-Predictive Controller. *IEEE Transactions on Parallel and Distributed Systems 24*, 1 (2013), 170–183.

[17] Bartolini, A., Cacciari, M., Tilli, A., Benini, L., and Gries, M. A Virtual Platform Environment for Exploring Power, Thermal and Reliability Management Control Strategies in High-Performance Multicores. In *ACM Great Lakes Symposium on VLSI - GLSVLSI* (2010).

[18] Baskiyar, S., and Dickinson, C. Scheduling directed a-cyclic task graphs on a bounded set of heterogeneous processors using task duplication. *Journal of Parallel and Distributed Computing 65*, 8 (2005), 911–921.

[19] Bellasi, P., Massari, G., and Fornaciari, W. A RTRM proposal for multi/many-core platforms and reconfigurable applications. In *International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip - ReCoSoC* (2012).

[20] Bellman, R. A Markovian Decision Process. *Journal of Mathematics and Mechanics 6* (1957), 679–684.

[21] Belson, D. The State of the Internet. *Akamai* (2013), 40. Available at `http://www.akamai.com/dl/documents/akamai_soti_q213.pdf?WT.mc_id=soti_Q213`.

[22] Bhadauria, M., and McKee, S. A. An approach to resource-aware co-scheduling for CMPs. In *ACM International Conference on Supercomputing - ICS* (2010).

[23] Biatek, T., Raulet, M., Travers, J. F., and Deforges, O. Efficient quantization parameter estimation in HEVC based on ρ-domain. In *European Signal Processing Conference - EUSIPCO* (2014).

[24] OpenBLAS. `http://xianyi.github.com/OpenBLAS/`.

[25] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPOPP* (1995).

[26] Bohr, M. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *IEEE Solid-State Circuits Society Newsletter 12*, 1 (2007), 11–13.

[27] Bordes, P., Andrivon, P., Hiron, F., Salmon, P., and Boitard, R. Joint Collaborative Team on Video Coding (JCT-VC) of ITU-T SG 16 WP 3 and ISO/IEC JTC 1/SC 29/WG 11, 2016. Available at `https://hevc.hhi.fraunhofer.de`.

[28] Bossen, F. Test Conditions and Software Reference Configurations, JCTVC-L1100. *JCT-VC Doc* (2013). Available at `https://hevc.hhi.fraunhofer.de/`.

[29] Bossen, F., Bross, B., Suhring, K., and Flynn, D. HEVC complexity and implementation analysis. *IEEE Transactions on Circuits and Systems for Video Technology 22*, 12 (2012), 1685–1696.

[30] Brock, D. C., and Moore, G. E. *Understanding Moore's Law: Four Decades of Innovation*. Chemical Heritage Foundation, 2006.

[31] Buisson, J., Sonmez, O., Mohamed, H., Lammers, W., and Epema, D. Scheduling malleable applications in multicluster systems. In *IEEE International Conference on Cluster Computing - ICCC* (2007).

[32] Calore, E., Gabbana, A., Schifano, S. F., and Tripiccione, R. Evaluation of DVFS techniques on modern HPC processors and accelerators for energy-aware applications. *Concurrency and Computation: Practice and Experience 29*, 12 (2017), e4143.

[33] Caron, E., and De Assuncao, M. D. Multi-criteria malleable task management for hybrid-cloud platforms. In *International Conference on Cloud Computing Technologies and Applications - CloudTech* (2017).

[34] Cesarini, D., Bartolini, A., and Benini, L. Benefits in relaxing the power capping constraint. In *Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems* (2017).

[35] Charvillat, V., and Grigoraş, R. Reinforcement learning for dynamic multimedia adaptation. *Journal of Network and Computer Applications 30*, 3 (2007), 1034–1058.

[36] Chen, Q., and Guo, M. Adaptive Workload-Aware Task Scheduling for Single-ISA Asymmetric Multicore Architectures. *ACM Transactions on Architecture and Code Optimization 11*, 1 (2014).

[37] Chronaki, K., Rico, A., Badia, R. M., Ayguadé, E., Labarta, J., and Valero, M. Criticality-aware dynamic task scheduling for heterogeneous architectures. In *ACM International Conference on Supercomputing - ICS* (2015).

[38] Chronaki, K., Rico, A., Casas, M., Moretó, M., Badia, R. M., Ayguadé, E., Labarta, J., and Valero, M. Task scheduling techniques for asymmetric multi-core systems. *IEEE Transactions on Parallel and Distributed Systems PP*, 99 (2016), 1–1.

[39] Cilk project home page. `http://supertech.csail.mit.edu/cilk/`.

[40] Corpas, A., Costero, L., Botella, G., Igual, F. D., García, C., and Rodríguez, M. Acceleration and energy consumption optimization in cascading classifiers for face detection on low-cost ARM big. LITTLE asymmetric architectures. *International Journal of Circuit Theory and Applications 46*, 9 (2018), 1756–1776.

[41] Costero, L., Igual, F. D., Olcoz, K., Catalán, S., Rodríguez-Sánchez, R., and Quintana-Ortí, E. S. Revisiting conventional task schedulers to exploit asymmetry in multi-core architectures for dense linear algebra operations. *Parallel Computing 68* (2017), 59 – 76.

[42] Costero, L., Igual, F. D., Olcoz, K., and Tirado, F. Energy Efficiency Optimization of Task-Parallel Codes on Asymmetric Architectures. In *International Conference on High Performance Computing Simulation - HPCS* (2017).

[43] Costero, L., Igual, F. D., Olcoz, K., and Tirado, F. Modifying OmpSs/Nanox to exploit energy efficiency in asymmetric architectures for the Cholesky factorization. In *HiPEAC Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems - ACACES* (2017).

[44] Costero, L., Igual, F. D., Olcoz, K., and Tirado, F. Providing On-Demand Quality & Resources for Malleable Applications. In *International Conference Computational and Mathematical Methods in Science and Engineering - CMMSE* (2019).

[45] Costero, L., Igual, F. D., Olcoz, K., and Tirado, F. Leveraging knowledge-as-a-service (KaaS) for QoS-aware resource management in multi-user video transcoding. *The Journal of Supercomputing* (2020).

[46] Costero, L., Iranfar, A., Zapater, M., Igual, F. D., Olcoz, K., and Atienza, D. MAMUT: Multi-Agent Reinforcement Learning for Efficient Real-Time Multi-User Video Transcoding. In *Design, Automation and Test in Europe Conference and Exhibition - DATE* (2019).

[47] Costero, L., Iranfar, A., Zapater, M., Igual, F. D., Olcoz, K., and Atienza, D. Resource Management for Power-Constrained HEVC Transcoding Using Reinforcement Learning. *IEEE Transactions on Parallel and Distributed Systems 31*, 12 (2020), 2834–2850.

[48] Czúni, L., Császár, G., and Licsár, A. Estimating the optimal quantization parameter in H.264. In *International Conference on Pattern Recognition - ICPR* (2006).

[49] Das, A., Al-Hashimi, B. M., Shafik, R. A., Kumar, A., Merrett, G. V., and Veeravalli, B. Reinforcement learning-based inter-and intra-application thermal optimization for lifetime improvement of multicore systems. In *Design Automation Conference - DAC* (2014).

[50] David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., and Le, C. RAPL: Memory power estimation and capping. In *ACM/IEEE International Symposium on Low Power Electronics and Design - ISLPED* (2010).

[51] Deng, Q., Meisner, D., Bhattacharjee, A., Wenisch, T. F., and Bianchini, R. CoScale: Coordinating CPU and memory system DVFS in server systems. In *IEEE/ACM 45th International Symposium on Microarchitecture - MICRO* (2012).

[52] Dennard, R. H., Gaensslen, F., Yu, H.-N., Rideout, L., Bassous, E., and LeBlanc, A. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits 5* (1974).

[53] Desell, T., Maghraoui, K. E., and Varela, C. A. Malleable applications for scalable high performance computing. *Cluster Computing 10*, 3 (2007), 323–337.

[54] Donyanavard, B., Muck, T., Rahmani, A. M., Dutt, N., Sadighi, A., Maurer, F., and Herkersdorf, A. SOSA: Self-optimizing learning with self-adaptive control for hierarchical system-on-chip management. In *IEEE/ACM International Symposium on Microarchitecture - MICRO* (2019).

[55] Donyanavard, B., Muck, T., Sarma, S., and Dutt, N. SPARTA: Runtime task allocation for energy efficient heterogeneous manycores. In *International Conference on Hardware/Software Codesign and System Synthesis - CODES+ISSS* (2016).

[56] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters 21*, 2 (2011), 173–193.

[57] Duran, A., Corbalán, J., and Ayguadé, E. Evaluation of openmp task scheduling strategies. In *International Workshop on OpenMP - IWOMP* (2008).

[58] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. *IEEE Micro 32*, 3 (2012), 122–134.

[59] Even-Dar, E., and Mansour, Y. Learning rates for Q-learning. *Lecture Notes in Computer Science 2111*, Dec (2001), 589–604.

[60] Extrae. `https://tools.bsc.es/`.

[61] Farhad, S. M., Bappi, M. S. I., and Ghosh, A. Dynamic Resource Provisioning for Video Transcoding in IaaS Cloud. In *IEEE International Conference on High Performance Computing and Communications & IEEE International Conference on Smart City & IEEE International Conference on Data Science and Systems - HPC-C/SmartCity/DSS* (2017).

[62] Felter, W., Rajamani, K., Keller, T., and Rusu, C. A performance-conserving approach for reducing peak power consumption in server systems. In *ACM International Conference on Supercomputing - ICS* (2005).

[63] Fettes, Q., Clark, M., Bunescu, R., Karanth, A., and Louri, A. Dynamic Voltage and Frequency Scaling in NoCs with Supervised and Reinforcement Learning Techniques. *IEEE Transactions on Computers 68*, 3 (2019), 375–389.

[64] Frigo, M., Leiserson, C. E., and Randall, K. H. The implementation of the cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI* (1998).

[65] Fu, X., Kabir, K., and Wang, X. Cache-aware utilization control for energy efficiency in multi-core real-time systems. In *Euromicro Conference on Real-Time Systems - ECRTS* (2011).

[66] Gadioli, D., Palermo, G., and Silvano, C. Application autotuning to support runtime adaptivity in multicore architectures. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation - SAMOS* (2015).

[67] Gadioli, D., Vitali, E., Palermo, G., and Silvano, C. MARGOt: A dynamic autotuning framework for self-aware approximate computing. *IEEE Transactions on Computers 68*, 5 (2019), 713–728.

[68] Gao, G., Wen, Y., and Westphal, C. Dynamic resource provisioning with QoS guarantee for video transcoding in online video sharing service. In *ACM Multimedia Conference - MM* (2016).

[69] Gao, G., Wen, Y., and Westphal, C. Dynamic Priority-Based Resource Provisioning for Video Transcoding With Heterogeneous QoS. *IEEE Transactions on Circuits and Systems for Video Technology 29*, 5 (2019), 1515–1529.

[70] Garcia, M., Corbalan, J., and Labarta, J. LeWI: A Runtime Balancing Algorithm for Nested Parallelism. In *International Conference on Parallel Processing - ICPP* (2009).

[71] Garcia-Garcia, A., Saez, J. C., Risco-Martin, J. L., and Prieto-Matias, M. PBBCache: An open-source parallel simulator for rapid prototyping and evaluation of cache-partitioning and cache-clustering policies. *Journal of Computational Science 42* (2020), 101102.

[72] Gautier, T., Besseron, X., and Pigeon, L. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *International Workshop on Parallel Symbolic Computation - PASCO* (2007).

[73] Gautier, T., Lima, J. V. F., Maillard, N., and Raffin, B. XKaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *IEEE International Parallel and Distributed Processing Symposium - IPDPS* (2013).

[74] Golub, G. H., and Van Loan, C. F. *Matrix Computations*, fourth ed. The Johns Hopkins University Press, Baltimore, 2013.

[75] Gupta, U., Ayoub, R., Kishinevsky, M., Kadjo, D., Soundararajan, N., Tursun, U., and Ogras, U. Y. Dynamic power budgeting for mobile systems running graphics workloads. *IEEE Transactions on Multi-Scale Computing Systems 4*, 1 (2018), 30–40.

[76] Hähnel, M., Döbel, B., Völp, M., and Härtig, H. Measuring energy consumption for short code paths using RAPL. *Performance Evaluation Review 40*, 3 (2012), 13–17.

[77] Haidar, A., Cao, C., Yarkhan, A., Luszczek, P., Tomov, S., Kabir, K., and Dongarra, J. Unified development for mixed multi-gpu and multi-coprocessor environments using a lightweight runtime environment. In *IEEE International Parallel and Distributed Processing Symposium - IPDPS* (2014).

[78] Hamzaoui, R., and Saupe, D. *Document and Image Compression*. CRC Press, 2006.

[79] Hanumaiah, V., Desai, D., Gaudette, B., Wu, C. J., and Vrudhula, S. STEAM: A smart temperature and energy aware multicore controller. *ACM Transactions on Embedded Computing Systems 13* (2014).

[80] Hardavellas, N., Ferdman, M., Falsafi, B., and Ailamaki, A. Toward dark silicon in servers. *IEEE Micro 31*, 4 (Jul 2011), 6–15.

[81] Hayward, C., and Madill, A. A Survey of Outlier Detection Methodologies. *Artificial Intelligence Review 22*, 2 (2004), 85–126.

[82] He, J., Yang, E. H., Yang, F., and Yang, K. Adaptive Quantization Parameter Selection for H.265/HEVC by Employing Inter-Frame Dependency. *IEEE Transactions on Circuits and Systems for Video Technology 28*, 12 (2018), 3424–3436.

[83] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach*, sixth ed. Morgan Kaufmann, 2017.

[84] Ho, H. N., and Lee, E. Model-based reinforcement learning approach for planning in self-adaptive software system. In *ACM International Conference on Ubiquitous Information Management and Communication - IMCOM* (2015).

[85] Hoffmann, H., Itagaki, T., Wood, D., and Bock, A. Studies on the Bit Rate Requirements for a HDTV Format With 1920×1080 pixel Resolution, Progressive Scanning at 50 Hz Frame Rate Targeting Large Flat Panel Displays. *IEEE Transactions on Broadcasting 52*, 4 (2006), 420–434.

[86] Hoffmann, H., Maggio, M., Santambrogio, M. D., Leva, A., and Agarwal, A. A generalized software framework for accurate and efficient management of performance goals. In *ACM SIGBED International Conference on Embedded Software - EMSOFT* (2013).

[87] Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., and Rinard, M. Dynamic knobs for responsive power-aware computing. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS* (2011).

[88] Hogbin, E. J. ACPI: Advanced Configuration and Power Interface, 2015.

[89] Huang, Y., Chen, C., Fu, C., Hsu, C., Chang, Y., Chuang, T., and Lei, S. Method and Apparatus of Delta Quantization Parameter Processing for High Efficiency Video Coding, 2012. US Patent App. 13/018,431, Google Patents.

[90] IBM. An architectural blueprint for autonomic computing. Tech. rep., IBM, 2005. Available at `https://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf`.

[91] Immich, P. K., Bhagavatula, R. S., and Pendse, R. Performance analysis of five interprocess communication mechanisms across UNIX operating systems. *Journal of Systems and Software 68*, 1 (2003), 27–43.

[92] Intel Corporation. Intel math kernel library (MKL). `http://software.intel.com/en-us/intel-mkl`.

[93] Intel Corporation. Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor, March 2004. White Paper. Available at: `http://download.intel.com/design/network/papers/30117401.pdf`.

[94] Intel Corporation. Intel Xeon Processor Scalable Family: Specification Update, 2018. Available at `https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf`.

[95] Iranfar, A., Pahlevan, A., Zapater, M., Žagar, M., Kovač, M., and Atienza, D. Online efficient bio-medical video transcoding on MPSoCs through content-aware workload allocation. In *Design, Automation and Test in Europe Conference and Exhibition - DATE* (2018).

[96] Iranfar, A., Shahsavani, S. N., Kamal, M., and Afzali-Kusha, A. A heuristic machine learning-based algorithm for power and thermal management of heterogeneous MPSoCs. In *ACM/IEEE International Symposium on Low Power Electronics and Design - ISLPED* (2015).

[97] Iranfar, A., Zapater, M., and Atienza, D. Machine Learning-Based Quality-Aware Power and Thermal Management of Multistream HEVC Encoding on Multicore Servers. *IEEE Transactions on Parallel and Distributed Systems 29*, 10 (2018), 2268–2281.

[98] Jung, H., Rong, P., and Pedram, M. Stochastic modeling of a thermally-managed multi-core system. In *Design Automation Conference - DAC* (2008).

[99] Kaapi project home page. `https://gforge.inria.fr/projects/kaapi`.

[100] Kaelbling, L. P., Littman, M. L., and Moore, A. W. Reinforcement learning: A survey. *Journal of artificial intelligence research 4* (1996), 237–285.

[101] Kehrer, S., and Blochinger, W. Elastic Parallel Systems for High Performance Cloud Computing: State-of-the-Art and Future Directions. *Parallel Processing Letters 29*, 2 (2019), 1950006.

[102] Khan, B., Goodman, D., Khan, S., Toms, W., Faraboschi, P., Luján, M., and Watson, I. Architectural support for task scheduling: hardware scheduling for dataflow on NUMA systems. *The Journal of Supercomputing 6*, 71 (2015).

[103] Khan, K. N., Hirki, M., Niemi, T., Nurminen, J. K., and Ou, Z. RAPL in action: Experiences in using RAPL for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems 3*, 2 (2018).

[104] Khan, M. U. K., Shafique, M., and Henkel, J. Power-Efficient Workload Balancing for Video Applications. *IEEE Transactions on Very Large Scale Integration Systems 24*, 6 (2016), 2089–2102.

[105] Khan, U. A., and Rinner, B. Online learning of timeout policies for dynamic power management. *ACM Transactions on Embedded Computing Systems 13*, 4 (2014), 96.

[106] Khaneghah, E. M., Mirtaheri, S. L., and Sharifi, M. Evaluating the effect of inter process communication efficiency on high performance distributed scientific computing. In *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing - EUC* (2008).

[107] Khattar, N., Sidhu, J., and Singh, J. Toward energy-efficient cloud computing: a survey of dynamic power management and heuristics-based optimization techniques. *Journal of Supercomputing 75*, 8 (2019), 4750–4810.

[108] Kirk, D. E. *Optimal Control Theory: An Introduction.* Prentice-Hall, 1970.

[109] Koivula, A., Viitanen, M., Vanne, J., Hämäläinen, T. D., and Fasnacht, L. Parallelization of Kvazaar HEVC intra encoder for multi-core processors. In *IEEE International Workshop on Signal Processing Systems - SiPS* (2015).

[110] Kotselidis, C., Clarkson, J., Rodchenko, A., Nisbet, A., Mawer, J., and Luján, M. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments - VEE* (2017).

[111] Krommydas, K., Scogland, T. R. W., and Feng, W. On the Programmability and Performance of Heterogeneous Platforms. In *IEEE International Parallel and Distributed Processing Symposium - IPDPS* (2013).

[112] Lapack project home page. `http://www.netlib.org/lapack`.

[113] Lee, J., Shin, I., and Park, H. Adaptive intra-frame assignment and bit-rate estimation for variable GOP length in H.264. *IEEE Transactions on Circuits and Systems for Video Technology 16* (2006), 1271–1279.

[114] Lefurgy, C., Wang, X., and Ware, M. Power capping: A prelude to power shifting. *Cluster Computing 11*, 2 (2008), 183–195.

[115] Leiserson, C. E. The cilk++ concurrency platform. In *Design Automation Conference - DAC* (2009).

[116] Li, X., Salehi, M. A., Bayoumi, M., Tzeng, N. F., and Buyya, R. Cost-Efficient and Robust On-Demand Video Transcoding Using Heterogeneous Cloud Services. *IEEE Transactions on Parallel and Distributed Systems 29*, 3 (2018), 556–571.

[117] Marchal, L., Simon, B., Sinnen, O., and Vivien, F. Malleable Task-Graph Scheduling with a Practical Speed-Up Model. *IEEE Transactions on Parallel and Distributed Systems 29*, 6 (2018), 1357–1370.

[118] Maurer, F., Donyanavard, B., Rahmani, A. M., Dutt, N., and Herkersdorf, A. Emergent Control of MPSoC Operation by a Hierarchical Supervisor/Reinforcement Learning Approach. In *Design, Automation and Test in Europe Conference and Exhibition - DATE* (2020).

[119] Menkovski, V., and Liotta, A. Intelligent control for adaptive video streaming. In *IEEE International Conference on Consumer Electronics - ICCE* (2013).

[120] Mercurium project home page. `https://pm.bsc.es/mcxx`.

[121] Merkel, A., Stoess, J., and Bellosa, F. Resource-conscious scheduling for energy efficiency on multicore processors. In *European Conference on Computer Systems - EuroSys* (2010).

[122] Miceli, R., Civario, G., Sikora, A., César, E., Gerndt, M., Haitof, H., Navarrete, C., Benkner, S., Sandrieser, M., Morin, L., and Bodin, F. AutoTune: A plugin-driven approach to the automatic tuning of parallel applications. *Lecture Notes in Computer Science 7782* (2013), 328–342.

[123] Moazzemi, K., Maity, B., Yi, S., Rahmani, A. M., and Dutt, N. HESSLE-FREE : Heterogeneous Systems Leveraging Fuzzy Control for Runtime Resource Management. *ACM Transactions on Embedded Computing Systems 18*, 5 (2019).

[124] Moore, S. K. Another Step Toward the End of Moore's Law. *IEEE Spectrum* (2019).

[125] Moore, S. K. The node is nonsense. *IEEE Spectrum 57*, 8 (2020), 24–30.

[126] Murray R., M., and Terry, S. *Statistics for Engineering and the Sciences*, fith ed. Pearson / Prentice Hall, 2007.

[127] Nanos++ project home page. `https://pm.bsc.es/nanox`.

[128] Narra, K. G., Lin, Z., Kiamari, M., Avestimehr, S., and Annavaram, M. Slack squeeze coded computing for adaptive straggler mitigation. In *International Conference for High Performance Computing, Networking, Storage and Analysis - SC* (2019).

[129] National Television System Committee. *Report and Reports of Panel No. 11, 11-A, 12-19, with Some supplementary references cited in the Reports, and the Petition for adoption of transmission standards for color television before the Federal Communications Commission.* National Television System Committee, 1953.

[130] Nishtala, R., Carpenter, P., Petrucci, V., and Martorell, X. The Hipster Approach for Improving Cloud System Efficiency. *ACM Transactions on Computer Systems 35*, 3 (2017), 1–28.

[131] Nogues, E., Berrada, R., Pelcat, M., Menard, D., and Raffin, E. A DVFS based HEVC decoder for energy-efficient software implementation on embedded processors. In *IEEE International Conference on Multimedia and Expo - ICME* (2015).

[132] Nogues, E., Heulot, J., Herrou, G., Robin, L., Pelcat, M., Menard, D., Raffin, E., and Hamidouche, W. Efficient DVFS for low power HEVC software decoder. *Journal of Real-Time Image Processing 13*, 1 (2017), 39–54.

[133] OpemMP Architecture Review Board. OpenMP Application Programming Interface. Tech. Rep. Version 5.0, OpemMP Architecture Review Board, 2018. Available at `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`.

[134] Ortega, C., Alvarez, L., Casas, M., Bertran, R., Buyuktosunoglu, A., Eichenberger, A. E., Bose, P., and Moreto, M. Intelligent Adaptation Of Hardware Knobs For Improving Performance and Power Consumption. *IEEE Transactions on Computers 9340*, c (2020), 1–1.

[135] Pääkkönen, P., Heikkinen, A., and Aihkisalo, T. Online architecture for predicting live video transcoding resources. *Journal of Cloud Computing 8*, 1 (2019).

[136] Pahlevan, A., Qu, X., Zapater, M., and Atienza, D. Integrating Heuristic and Machine-Learning Methods for Efficient Virtual Machine Allocation in Data Centers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37*, 8 (2018), 1667–1680.

[137] Palomino, D., Shafique, M., Susin, A., and Henkel, J. Tone: Adaptive temperature optimization for the next generation video encoders. In *ACM/IEEE International Symposium on Low Power Electronics and Design - ISLPED* (2014).

[138] Pan, J. RAPL (Running Average Power Limit) driver. LWN: news from the source - online, Apr 2013. Available at: `https://lwn.net/Articles/545745/`.

[139] Paone, E., Gadioli, D., Palermo, G., Zaccaria, V., and Silvano, C. Evaluating orthogonality between application auto-tuning and run-time resource management for adaptive OpenCL applications. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors - ASAP* (2014).

[140] Paraver: the flexible analysis tool. https://tools.bsc.es/.

[141] Petrangeli, S., Claeys, M., Latré, S., Famaey, J., and De Turck, F. A multi-agent Q-learning-based framework for achieving fairness in HTTP adaptive streaming. In *IEEE/IFIP Network Operations and Management Symposium - NOMS* (2014).

[142] Petrangeli, S., Famaey, J., Claeys, M., Latré, S., and De Turck, F. QoE-driven rate adaptation heuristic for fair adaptive video streaming. *ACM Transactions on Multimedia Computing, Communications and Applications 12*, 2 (2015), 28:1——–28:24.

[143] Petrica, P., Izraelevitz, A. M., Albonesi, D. H., and Shoemaker, C. A. Flicker: A dynamically adaptive architecture for power limited multicore systems. In *International Symposium on Computer Architecture - ISCA* (2013).

[144] Pham Van, L., De Praeter, J., Van Wallendael, G., Van Leuven, S., De Cock, J., and Van De Walle, R. Efficient Bit Rate Transcoding for High Efficiency Video Coding. *IEEE Transactions on Multimedia 18*, 3 (2016), 364–378.

[145] Planas, J., Badia, R. M., Ayguadé, E., and Labarta, J. Self-adaptive ompss tasks in heterogeneous environments. In *IEEE International Parallel and Distributed Processing Symposium - IPDPS* (2013).

[146] Pothukuchi, R. P., Greathouse, J. L., Rao, K., Erb, C., Piga, L., Voulgaris, P. G., and Torrellas, J. Tangram: Integrated control of heterogeneous computers. In *IEEE/ACM International Symposium on Microarchitecture - MICRO* (2019).

[147] Pricopi, M., Muthukaruppan, T. S., Venkataramani, V., Mitra, T., and Vishin, S. Power-performance modeling on asymmetric multi-cores. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems - CASES* (2013).

[148] Pricopi, M., Muthukaruppan, T. S., Venkataramani, V., Mitra, T., and Vishin, S. Power-performance modeling on asymmetric multi-cores. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems - CASES* (2013).

[149] Qin, H., Zawad, S., Zhou, Y., Yang, L., Zhao, D., and Yan, F. Swift machine learning model serving scheduling: A region based reinforcement learning approach. In *International Conference for High Performance Computing, Networking, Storage and Analysis - SC* (2019).

[150] QUARK project home page. http://icl.cs.utk.edu/quark.

[151] Quintana-Ortí, G., Quintana-Ortí, E. S., van de Geijn, R. A., Van Zee, F. G., and Chan, E. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software 36*, 3 (2009), 14:1–14:26.

[152] Raghavendra, R., Ranganathan, P., Talwar, V., Wang, Z., and Zhu, X. No "power" struggles: Coordinated multi-level power management for the data center. *ACM Operating Systems Review 42*, 2 (2008), 48–59.

[153] Reda, S., Cochran, R., and Coskun, A. K. Adaptive power capping for servers with multithreaded workloads. *IEEE Micro 32*, 5 (2012), 64–75.

[154] Reinders, J. *Intel Threading Building Blocks : outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[155] Roca, A., Rodriguez, S., Segura, A., Marquet, K., and Beltran, V. A Linux Kernel Scheduler Extension for Multi-core Systems. In *IEEE International Conference on High Performance Computing - HiPC* (2019).

[156] Rodrigues, C. F., Riley, G., and Luján, M. Exploration of Task-Based Scheduling for Convolutional Neural Networks Accelerators under Memory Constraints. In *ACM International Conference on Computing Frontiers - CF* (2019).

[157] Ross, S. *Introduction to Stochastic Dynamic Programming*. Academic Press, 1983.

[158] Rotem, E., Naveh, A., Ananthakrishnan, A., Weissmann, E., and Rajwan, D. Power-management architecture of the intel microarchitecture code-named Sandy Bridge. *IEEE Micro 32*, 2 (2012), 20–27.

[159] Russell, S., and Norving, P. *Artificial Intelligence: A Modern Approach*, third ed. Pearson, 2016.

[160] Sandvine. Global Internet Phenomena. Tech. rep., Sandvine, 2016. Available at `https://www.sandvine.com/hubfs/downloads/archive/2016-global-internet-phenomena-report-latin-america-and-north-america.pdf`.

[161] Shoham, Y., Powers, R., and Grenager, T. Multi-Agent Reiforcement Learning: a critical survey. Tech. rep., Computer Science Department, Standford University, 2003. Available at `https://www.cc.gatech.edu/classes/AY2009/cs7641_spring/handouts/MALearning_ACriticalSurvey_2003_0516.pdf`.

[162] Silvano, C., Agosta, G., Cherubin, S., Gadioli, D., Palermo, G., Bartolini, A., Benini, L., Martinovič, J., Palkovič, M., Slaninová, K., Bispo, J. a., Cardoso, J. a. M. P., Abreu, R., Pinto, P., Cavazzoni, C., Sanna, N., Beccari, A. R., Cmar, R., and Rohou, E. The ANTAREX Approach to Autotuning and Adaptivity for Energy Efficient HPC Systems. In *ACM International Conference on Computing Frontiers - CF* (2016).

[163] Silveira, D., Porto, M., and Bampi, S. Performance and energy consumption analysis of the X265 video encoder. In *European Signal Processing Conference - EUSIPCO* (2017).

[164] Singh, A. K., Prakash, A., Basireddy, K. R., Merrett, G. V., and Al-Hashimi, B. M. Energy-efficient run-time mapping and thread partitioning of concurrent OpenCL applications on CPU-GPU MPSoCs. *ACM Transactions on Embedded Computing Systems 16*, 5s (2017).

[165] Singh, S., and Chana, I. QoS-aware autonomic resource management in cloud computing: A systematic review. *ACM Computing Surveys 48*, 3 (2015).

[166] Singla, G., Kaur, G., Unver, A. K., and Ogras, U. Y. Predictive dynamic thermal and power management for heterogeneous mobile platforms. In *Design, Automation and Test in Europe Conference and Exhibition - DATE* (2015).

[167] Spiegel, M. R., and Stephens, L. J. *Schaum's Outlines Statistics*, fourth ed. McGraw Hill, 2008.

[168] StarPU project home page. `http://starpu.gforge.inria.fr/`.

[169] Sullivan, G. J., Ohm, J. R., Han, W. J., and Wiegand, T. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on Circuits and Systems for Video Technology 22*, 12 (2012), 1649–1668.

[170] Superglue project home page. `http://www.it.uu.se/research/scientific_computing/software/superglue`.

[171] Sutton, R. S., and Barto, A. G. *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge, 1998.

[172] Tan, X., Bosch, J., Vidal, M., Álvarez, C., Jiménez-González, D., Ayguadé, E., and Valero, M. General Purpose Task-Dependence Management Hardware for Task-Based Dataflow Programming Models. In *IEEE International Parallel and Distributed Processing Symposium - IPDPS* (2017).

[173] Tang, X., Wang, H., Ma, X., El-Sayed, N., Zhai, J., Chen, W., and Aboulnaga, A. Spread-n-share: Improving application performance and cluster throughput with resource-aware job placement. In *International Conference for High Performance Computing, Networking, Storage and Analysis - SC* (2019).

[174] Tapus, C., I-Hsin Chung, and Hollingsworth, J. Active Harmony: Towards Automated Performance Tuning. In *ACM International Conference on Supercomputing - ICS* (2015).

[175] Tembey, P., Gavrilovska, A., and Schwan, K. A case for coordinated resource management in heterogeneous multicore platforms. *Lecture Notes in Computer Science 6161* (2012), 341–356.

[176] Teodorescu, R., and Torrellas, J. Variation-aware application scheduling and power management for chip multiprocessors. In *International Symposium on Computer Architecture - ISCA* (2008).

[177] Terpstra, D., Jagode, H., You, H., and Dongarra, J. Collecting performance data with PAPI-C. In *International Workshop on Parallel Tools for High Performance Computing* (2010).

[178] Tesauro, G., Jong, N. K., Das, R., and Bennani, M. N. A hybrid reinforcement learning approach to autonomic resource allocation. In *International Conference on Autonomic Computing - ICAC* (2006), pp. 65–73.

[179] Tillenius, M. Superglue: A shared memory framework using data versioning for dependency-aware task-based parallelization. *SIAM Journal on Scientific Computing 37*, 6 (2015), C617–C642.

[180] Tillenius, M., Larsson, E., Badia, R. M., and Martorell, X. Resource-aware task scheduling. *ACM Transactions on Embedded Computer Systems 14*, 1 (2015), 5:1–5:25.

[181] van Hasselt, H. *Reinforcement Learning in Continuous State and Action Spaces*. Springer Berlin Heidelberg, 2012.

[182] Van Zee, F. G., Smith, T., Igual, F. D., Smelyanskiy, M., Zhang, X., Kistler, M., Austel, V., Gunnels, J., Low, T. M., Marker, B., Killough, L., and van de Geijn, R. A. The BLIS framework: Experiments in portability. *ACM Transactions on Mathematical Software 42*, 2 (2016), 12:1–12:19.

[183] Van Zee, F. G., and van de Geijn, R. A. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software 41*, 3 (2015), 14:1–14:33.

[184] Vega, A., Buyuktosunoglu, A., Hanson, H., Bose, P., and Ramani, S. Crank it up or dial it down: Coordinated multiprocessor frequency and folding control. In *IEEE/ACM International Symposium on Microarchitecture - MICRO* (2013).

[185] Vetter, J. S., DeBenedictis, E. P., and Conte, T. M. Architectures for the Post-Moore Era. *IEEE Micro 37* (2017), 6–8.

[186] Viitanen, M., Koivula, A., Lemmetti, A., Ylä-Outinen, A., Vanne, J., and Hämäläinen, T. D. Kvazaar: Open-Source HEVC/H. 265 Encoder. In *ACM Multimedia Conference - MM* (2016).

[187] Voss, M., Asenjo, R., and Reinders, J. *Pro TBB: C++ Parallel programming with Threading Building Blocks*. Apress open, 2019.

[188] Voss, M. J., and Eigenmann, R. ADAPT: Automated De-coupled Adaptive Program Transformation. In *International Conference on Parallel Processing - ICPP* (2000).

[189] Wang, L., and Gelenbe, E. Adaptive Dispatching of Tasks in the Cloud. *IEEE Transactions on Cloud Computing 6*, 1 (2018), 33–45.

[190] Wang, X., and Martínez, J. F. ReBudget: Trading Off Efficiency vs. Fairness in Market-Based Multicore Resource Allocation via Runtime Budget Reassignment. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS* (2016).

[191] Wang, Z., Sun, L., Wu, C., Zhu, W., Zhuang, Q., and Yang, S. A joint online transcoding and delivery approach for dynamic adaptive streaming. *IEEE Transactions on Multimedia 17*, 6 (2015), 867–879.

[192] Watson, D. Challenges of Exascale Computing. In *ENES Workshop on High Performance Computing for Climate and Weather* (2011).

[193] Weiser, M., Welch, B., Demers, A., and Shenker, S. Scheduling for reduced cpu energy. *Low-Power CMOS Design* (1998), 177–187.

[194] Welstead, S. *Fractal and Wavelet Image Compression Techniques*. SPIE Optical Engineering Press, Bellingham, WA, 2009.

[195] Whaley, R., and Dongarra, J. Automatically Tuned Linear Algebra Software. In *ACM/IEEE Conference on Supercomputing* (2014).

[196] Winter, J. A., Albonesi, D. H., and Shoemaker, C. A. Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *Parallel Architectures and Compilation Techniques - PACT* (2010).

[197] Yan, K., Zhang, X., Tan, J., and Fu, X. Redefining QoS and customizing the power management policy to satisfy individual mobile users. In *IEEE/ACM International Symposium on Microarchitecture - MICRO* (2016).

[198] YarKhan, A., Kurzak, J., and Dongarra, J. QUARK Users' Guide: QUeueing And Runtime for Kernels. Tech. rep., Innovative Computing Laboratory, University of Tennessee, 2011. Available at `https://www.icl.utk.edu/publications/quark-users-guide-queueing-and-runtime-kernels`.

[199] Yesil, S., Heidarshenas, A., Morrison, A., and Torrellas, J. Understanding priority-based scheduling of graph algorithms on a shared-memory platform. In *International Conference for High Performance Computing, Networking, Storage and Analysis - SC* (2019).

[200] Yuan, C., Yue, Y., Li, X., and Feng, L. Performance analysis and optimization of inter process communication in android. In *International Conference on Electronics Information and Emergency Communication - ICEIEC* (2016).

[201] Zee, F. G. V. `libflame`. the complete reference, 2016. `http://www.cs.utexas.edu/users/flame`.

[202] Zhang, H., and Hoffmann, H. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS* (2016).

[203] Zhang, H., and Hoffmann, H. Performance & Energy Tradeoffs for Dependent Distributed Applications Under System-Wide Power Caps. In *International Conference on Parallel Processing - ICPP* (2018).

[204] Zhang, H., and Hoffmann, H. PoDD: Power-capping dependent distributed applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis - SC* (2019).

[205] Zhang, Y., Kwong, S., and Wang, S. Machine learning based video coding optimizations: A survey. *Information Sciences 506* (2020), 395–423.

[206] Zhang, Y., Yao, J., and Guan, H. Intelligent Cloud Resource Management with Deep Reinforcement Learning. *IEEE Cloud Computing 4*, 6 (2017), 60–69.

[207] Zhao, T., Wang, Z., and Chen, C. W. Adaptive Quantization Parameter Cascading in HEVC Hierarchical Coding. *IEEE Transactions on Image Processing 25*, 7 (2016), 2997–3009.

[208] Zhuravlev, S., Saez, J. C., Blagodurov, S., Fedorova, A., and Prieto, M. Survey of energy-cognizant scheduling techniques. *IEEE Transactions on Parallel and Distributed Systems 24*, 7 (2013), 1447–1464.

Acronyms

AI Artificial Intelligence . . . 83
AMP Asymmetric Multiprocessor . . . 7
API application programming interface . . . 174
DAG Direct Acyclic Graph . . . 28
DLA Dense Linear Algebra . . . 2
DVFS Dynamic Voltage-Frequency Scaling . . . 2
FIFO First-In-First-Out . . . 58
FPS Frames Per Second . . . 87
FSM Finite State Machine . . . 159
HEVC High Efficiency Video Coding . . . 83
HPC High Performance Computing . . . 1
HR High Resolution . . . 91
IC integrated circuit . . . 1
I/O Input/Output . . . 81
ISA Instruction Set Architecture . . . 14
KB Knowledge Base . . . 148
LCT Learning Classifier Table . . . 16
LR Low Resolution . . . 106
MAL Multi-Agent Learning . . . 9
MDP Markov Decision Problem . . . 81
ML Machine Learning . . . 16
OP Operating Point . . . 123
OpenMP Open Multi-Processing . . . 27
OS Operating System . . . 41
PSNR Peak Signal-to-Noise Ratio . . . 85
QL Q-Learning . . . 16
QoE Quality of Experience . . . 4
QoS Quality of Service . . . 4
QP Quantization Parameter . . . 87
RAPL Running average power limit . . . 2
RL Reinforcement Learning . . . 16