Evaluación y optimización de rendimiento y consumo energético de aplicaciones paralelas a nivel de tareas sobre arquitecturas asimétricas

Trabajo de Fin de Máster realizado por

Luis María Costero Valero

Dirigido por los Doctores

Francisco Daniel Igual Peña **Katzalin Olcoz Herrero**

Calificación: 9,5

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid

Septiembre 2016

Autorización de difusión

Luis María Costero Valero

Madrid, a 5 de septiembre de 2016

El abajo firmante, matriculado en el Máster en Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: *"Evaluación y optimización de rendimiento y consumo energético de aplicaciones paralelas a nivel de tareas sobre arquitecturas asimétricas"*, realizado durante el curso académico 2015-2016 bajo la dirección de Francisco Daniel Igual Peña y Katzalin Olcoz Herrero, del Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en internet y garantizar su preservación y acceso a largo plazo.

Fdo.: Luis María Costero Valero

Índice general

Índice de figuras

Índice de tablas

Índice de listados

Resumen

Las arquitecturas asimétricas, formadas por varios procesadores con el mismo repertorio de instrucciones pero distintas características de rendimiento y consumo, ofrecen muchas posibilidades de optimización del rendimiento y/o el consumo en la ejecución de aplicaciones paralelas. La planificación de tareas sobre dichas arquitecturas de forma que se aprovechen de manera eficiente los distintos recursos, es muy compleja y se suele abordar utilizando modelos de programación paralelos, que permiten al programador especificar el paralelismo de las tareas, y entornos de ejecución que explotan dinámicamente dicho paralelismo.

En este trabajo hemos modificado uno de los planificadores de tareas más utilizados en la actualidad para intentar aprovechar todos los recursos al máximo, cuando el rendimiento así lo necesite, o para conseguir la mejor eficiencia energética posible, cuando el consumo sea más prioritario. También se ha utilizado una biblioteca desarrollada específicamente para la arquitectura asimétrica objeto de estudio en la Universidad de Texas, Austin.

Para obtener el máximo rendimiento se han agrupado los núcleos del sistema en dos niveles: hay un cluster simétrico de núcleos virtuales idénticos, cada uno de los cuales está compuesto por un conjunto de núcleos asimétricos. El planificador de tareas asigna trabajo a los núcleos virtuales, de manera idéntica a como lo haría en un sistema multinúcleo simétrico, y la biblioteca se encarga de repartir el trabajo entre los núcleos asimétricos. El trabajo ha consistido en integrar dicha biblioteca con el planificador de tareas.

Para mejorar la eficiencia energética se han incluido en el planificador de tareas políticas de explotación de los modos de bajo consumo de la arquitectura y también de apagado o no asignación de carga de trabajo a algunos de los núcleos, que se activan en tiempo de ejecución cuando se detecta que la aplicación no necesita todos los recursos disponibles en la arquitectura.

Palabras clave

Arquitecturas asimétricas, planificación, DVFS, mejora rendimiento, consumo energético.

Abstract

Asymmetric architectures, composed by multiple processors with the same instruction set but different performance and energy consumption characteristics, provide many optimisation possibilities in performance and/or energy consumption over the execution of parallel applications. Scheduling tasks in this kind of architectures in such a way that all the resources are used efficiently is a difficult task, and it is usually addressed using parallel programming models, allowing programmers to annotate the parallelism for each of the different tasks of the program, and runtime environments which dinamically take advantage of this kind of parallelism.

In this work we have modified one of the most used task schedulers nowadays to try to use all the resources efficiently if high performance is needed, or to achieve the best energy efficiency if consumption reduction is the priority. A math library developed specifically for this kind of asymmetric architectures in the University of Texas, Austin, has been used as well.

To obtain maximum performance, cores have been grouped into two different levels: a symmetric cluster with identical virtual cores where each of them is composed by a set of asymmetric cores. With this approach, schedulers can allocate tasks to the virtual cores in the same way as they would do in a symmetric multicore system, and the library is responsible for the allocation of tasks to the asymmetric cores. The work carried out consisted of integrating this library with the task scheduler.

To improve energy efficiency, new policies have been included into the scheduler. These policies have been designed to take advantage of the low consumption modes of the architecture and to disable or not assign tasks to some of the cores, activating them at run-time if the execution does not need all the available resources in the architecture.

Keywords

Asymmetric Architectures, scheduling, DVFS, performance improvement, energy consumption.

Capítulo 1

Introducción

1.1. Motivación

En los últimos años, el consumo energético ha emergido como una de las principales barreras que ha frenado la mejora de rendimiento en los sistemas de cómputo, desde dispositivos móviles hasta grandes centros de procesamiento de datos (CPDs). Con el fin de mejorar la eficiencia energética, resulta necesario explotar mayores grados de especialización en el hardware. Las arquitecturas actuales proporcionan diversos tipos de heterogeneidad, que van desde configuraciones heterogéneas en CPDs, hasta procesadores con aceleradores hardware integrados en el propio chip, plataformas híbridas formadas por procesadores de propósito general (CPUs) y aceleradores de propósito específico (como pueden ser GPUs, Intel Xeon Phi, etc.), multicores asimétricos (AMPSs) con repertorio común de instrucciones o multicores heterogéneos con múltiples repertorios de instrucciones [19, 29].

La correcta explotación del rendimiento y la eficiencia energética ofrecidos por estas arquitecturas conlleva, de forma necesaria, un incremento en la complejidad del software que permita, de forma lo más transparente posible para el usuario final, la adaptación de aplicaciones ya existentes o de nuevo desarrollo para aumentar el rendimiento y reducir su consumo energético [40, 45]. Uno de los principales retos en este sentido es la correcta asignación de trabajo a los recursos computacionales existentes, de modo que, de forma simultánea, se optimice su grado de ocupación y la adaptación del recurso escogido a la tarea específica a desarrollar. En respuesta a estas necesidades, una de las soluciones propuestas por la comunidad científica es el desarrollo de modelos de programación que expongan el paralelismo a nivel de tareas, explotado en tiempo de ejecución por planificadores de tareas (comúnmente conocidos como runtimes [53]). Esta capa de software es capaz de gestionar de forma transparente al usuario los procesos de gestión de dependencias de datos entre tareas,

las transferencias de datos en entornos heterogéneos, y el mapeado eficiente entre tipos de tareas y de procesador, entre otros aspectos.

Atendiendo a las arquitecturas subyacentes, el trabajo se centra en una de las familias de arquitecturas con más perspectiva de futuro de entre las anteriormente mencionadas: procesadores multinúcleo asimétricos (AMPs) de bajo consumo, y en concreto, procesadores de la familia big.LITTLE de ARM. Este tipo de configuraciones ha suscitado gran interés en el mercado de las aplicaciones y ecosistemas móviles, debido a la especialización de cada tipo de procesador ante tareas concretas. Esta especialización se suele traducir en la activación/desactivación de cada tipo de núcleo en función de la demanda computacional o prioridad de las aplicaciones en ejecución, o de las limitaciones energéticas ligadas a cada escenario concreto de ejecución.

1.2. Objetivos

Desde el punto de vista de la computación de altas prestaciones (HPC), el uso conjunto de todos los recursos computacionales desde una misma aplicación multihebra resulta de especial interés. Aunque algunas de las técnicas ya desarrolladas en bibliotecas o planificadores de tareas existentes pueden ser aplicadas directamente en este tipo de arquitecturas, éstas exponen nuevos retos y problemáticas que hay que abordar. Este trabajo se centra en dos de estos retos:

- Integración y combinación de bibliotecas específicas conscientes de la asimetría con *runtimes* convencionales (no conscientes de la asimetría), y comparativa de este enfoque con un diseño de *runtime* sí consciente con la asimetría de la arquitectura.

- Diseño de técnicas de reducción de consumo energético. Aplicación de técnicas de escalado dinámico de frecuencia (DVFS) o desactivación de núcleos en distintas fases de la ejecución de una determinada aplicación, en función de las demandas de cómputo y grado de paralelismo disponible, y evaluación del impacto de este tipo de técnicas en el tiempo de ejecución y consumo energético global.

1.3. Metodología y plan de trabajo

Para alcanzar los objetivos descritos en el apartado anterior, se han abordado las siguientes tareas específicas:

T1. Estudio del estado del arte en sistemas asimétricos desde el punto de vista arquitectural, y del soporte software disponible para explotar sus características, a nivel de sistema operativo, *runtime* o aplicación.

T2. Estudio, análisis y evaluación del rendimiento de familias de bibliotecas y planificadores de tareas en tiempo de ejecución no conscientes de la asimetría de la arquitectura, así como soluciones ya adaptadas a esta familia de arquitecturas. En el ámbito del álgebra lineal, utilizado como hilo conductor del trabajo, se ha utilizado la biblioteca BLIS [55].

T3. Integración de las anteriores bibliotecas sobre *runtimes* no conscientes de la asimetría ya existentes, para explotar de forma transparente la asimetría de las arquitecturas sin modificaciones en los planificadores, para un conjunto de operaciones matemáticas de interés.

T4. Diseño de heurísticas, políticas de asignación de recursos y explotación de los modos de bajo consumo proporcionados por las arquitecturas, que permitan un aprovechamiento óptimo de los recursos a nivel del planificador de tareas en *runtimes* existentes, con el objetivo de explotar de manera energéticamente eficiente todos los recursos computacionales existentes.

T5. Evaluación del rendimiento y eficiencia energética de las nuevas políticas desarrolladas. Comparación de los resultados con soluciones actuales conscientes de la asimetría de la arquitectura.

1.4. Estructura del documento

La presente memoria se estructura como sigue. El Capítulo 2 ofrece una descripción general de las arquitecturas heterogéneas y asimétricas actualmente más utilizadas en el ámbito de la computación de altas prestaciones, haciendo especial hincapié en el paradigma big.LITTLE de ARM utilizado en el trabajo realizado. Además, detalla las características específicas del entorno experimental empleado a nivel hardware y software. El Capítulo 3 introduce dos de las estrategias de paralelización utilizadas en el desarrollo del trabajo: extracción de paralelismo a nivel de tareas, y extracción de paralelismo a nivel de datos, centrándose en las características de cada uno, así como en sus ventajas e inconvenientes. Los Capítulos 4 y 5 describen en detalle el trabajo realizado, dividido en la propuesta y evaluación de técnicas para la mejora de rendimiento en planificadores a nivel de tareas, y en el desarrollo de soluciones para la mejora de la eficiencia energética, respectivamente.

Finalmente, el Capítulo 6 incluye una serie de conclusiones generales y líneas de trabajo futuras.

Introduction

Motivation

In recent years, energy consumption has been revealed as one of the barriers that has stopped performance improvements in all computer systems, from mobile devices to big data centres. In order to improve the energy efficiency, it is necessary to exploit greater specialisation levels in the hardware. Nowadays, architectures provide different types of heterogeneity, from heterogeneous configurations for data centres, to processors with hardware accelerators inside the chip, hybrid platforms composed by general purpose processors (CPUs) and specific purpose accelerators (like GPUs, Intel Xeon Phi, etc.), asymmetric multicores (AMPSs) with the same instruction set, or heterogeneous multicores with different instruction sets [19, 29].

The correct exploitation of performance and energy efficiency offered by these architectures leads to an increase in complexity of the software that permits, in the most transparent way for the end user, the adaptation of already existing applications or newly developed applications to increase performance and reduce the energy consumption [40, 45]. One of the main challenges in this topic is the right assignment of work to the existent computational resources, which should be done in such a way that, simultaneously, its activity degree and the adaptation between the task and the resources are optimised. In response to these needs, one of the scientific community solutions proposed is the development of programming models which expose the task parallelism, exploited in execution time by task schedulers (usually known as runtimes [53]). This software layer can manage, in a transparent user way, the process of managing the dependencies between tasks, data transfers between heterogeneous environments, and the efficient assignment between types of tasks and processors, as many others aspects.

Regarding underlying architectures, the work focuses on one of the family architectures with more future perspective between the previously cited: asymmetric multicore architectures (AMPs) with low energy consumption and, specifically, processors of ARM's big.LITTLE family. This kind of configurations has caused a high interest in the application market and mobile ecosystems, because of the specialisation of each kind of processor to specific tasks. This kind of specialisation is translated into the activation/deactivation

of each kind of core depending of the computational demand or application priorities in execution, or the energy limitations of each execution situation.

Objectives

From a high performance computing (HPC) point of view, using all computational resources together from one multithreaded application has a special interest. Although some techniques developed on existing libraries and task schedulers can be used in this kind of architectures, these expose new challenges to face. This work focus on two of these challenges:

- Integration and combination of specific asymmetry-aware libraries with conventional *runtimes* (non conscious of the asymmetry), and comparative of this approach with a *runtime* asymmetry-aware design.
- Developing of new energy consumption reduction techniques. Applying dynamic frequency scaling (DVFS) techniques, or deactivating cores in different moments during the application execution depending on the computing demands and parallelism degree available, and evaluation of the impact of these techniques in run-time and global energy consumption.

Methodology and work plan

In order to achieve the objectives described in the previous section, the next specific tasks have been done:

T1. Study of the state of the art of asymmetric systems from an architectural point of view, and of the software support for exploiting its characteristics, from the operative system level, *runtime* level or application level.

T2. Study, analysis and evaluation of performance of families of libraries and schedulers non conscious of the architecture's asymmetry, and also of solutions adapted for this kind of architectures. In the linear algebra scope, used as the main thread of this work, the BLIS library [55] has been used.

T3. Integration of the previous libraries over existing asymmetry-aware non conscious schedulers, to exploit the asymmetry of the architecture without modifying the actual schedulers in a transparent way, specially over a set of mathematical interesting operations.

T4. Design of heuristics, resource scheduling policies and exploitation of low power consuming modes provided by this kind of architectures, which allow an optimal use of the resources in existing schedulers, with the objective of exploiting every existing computational resource in an energy efficient way.

T5. Evaluation of performance and energy efficiency of the new policies developed. Comparison between these results with current asymmetry-aware solutions.

Document structure

This document is organised as follow: Chapter 2 shows a general description of the heterogeneous and asymmetric architectures most used nowadays, focusing on ARM's big.LITTLE architectures used in this work. Also, specific characteristics of the environment used for testing (including hardware and software) are described. Chapter 3 introduces the two parallel strategies used in this work: task-level parallelism and data-level parallelism, focusing on the characteristics of each one of them and also on their advantages and disadvantages. Chapters 4 and 5 show a complete description of the work developed, splitted into the proposed techniques and their evaluation for obtaining a performance gain over task-level schedulers, and techniques developed for obtain at least some energy efficiency gain, respectively. Finally, chapter 6 shows a set of general conclusions and future work.

Capítulo 2

Arquitecturas asimétricas y heterogéneas

2.1. Descripción general del paradigma big.LITTLE

2.1.1. Introducción a las arquitecturas asimétricas

Desde la aparición de los procesadores multinúcleo, y tras su auge durante la pasada década como respuesta a las limitaciones en el incremento de la frecuencia de reloj impuestas por la tecnología, este tipo de arquitecturas se han convertido en un pilar clave en el desarrollo de arquitecturas de altas prestaciones, reuniendo a la vez una gran capacidad de cálculo y una eficiencia energética notable.

Históricamente, los procesadores multinúcleo han estado formados por varias Unidades de Procesamiento (PUs), típicamente con idénticas características entre ellas. Sin embargo, a medida que la ley de escalabilidad de Dennard [13] comienza a ver su fin, los sistemas heterogéneos se han convertido en las arquitecturas más atractivas en el campo de la computación de altas prestaciones. Este tipo de sistemas, típicamente equipados con núcleos de procesamiento de propósito general y aceleradores hardware de propósito específico, exhiben características, tanto desde el punto de vista de las prestaciones como del consumo energético, que responden a las necesidades actuales exigidas por los grandes problemas que surgen en diversos ámbitos de la ciencia y la ingeniería.

Dentro de los sistemas heterogéneos es posible realizar una segunda clasificación en función de que los distintos núcleos dispongan de distintos repertorios de instrucciones (*heterogeneous-ISA*) o utilicen el mismo repertorio de instrucciones (*single-ISA*). Estos últimos se denominan habitualmente procesadores asimétricos (*asymmetric single-ISA CMP o ASISA-CMP*) [36].

En un procesador multinúcleo asimétrico (AMP), las diferencias entre los distintos nú-

cleos se deben a distintas microarquitecturas (asimetría física) o se trata de procesadores con la misma microarquitectura pero distinta frecuencia o tamaño de cache (asimetría virtual). En ocasiones, se usan técnicas de escalado dinámico de voltaje y frecuencia (DVFS) para emular procesadores asimétricos a partir de núcleos simétricos, consiguiendo así un sistema con asimetría virtual. Las ventajas de este tipo de arquitecturas dependen del escenario en el que son utilizadas; en la actualidad, el uso de arquitecturas asimétricas es una tendencia en alza en el mercado de dispositivos móviles, en el que el tipo de tareas a ejecutar y su criticidad es altamente heterogéneo, y el tiempo de vida de las baterías es un bien escaso y muy a tener en cuenta.

Dentro de las arquitecturas multinúcleo asimétricas, tienen especial relevancia los procesadores de la familia ARM, que se denominan big.LITTLE para reflejar que incluyen procesadores potentes (*big*) junto con procesadores de menor rendimiento y mucho menor consumo (*LITTLE*). Por ejemplo, se pueden combinar núcleos Cortex-A15 (big) con Cortex-A7 (LITTLE), si nos referimos al repertorio de instrucciones de 32 bits (ARMv7), o Cortex A57 (big) con A53 (LITTLE) para el caso de 64 bits (ARMv8). Uno de los ejemplos reales más utilizados a día de hoy es el SoC Samsung Exynos (véase Figura 2.1) en sus versiones de 32 y 64 bits, ampliamente utilizado en dispositivos móviles de última generación.

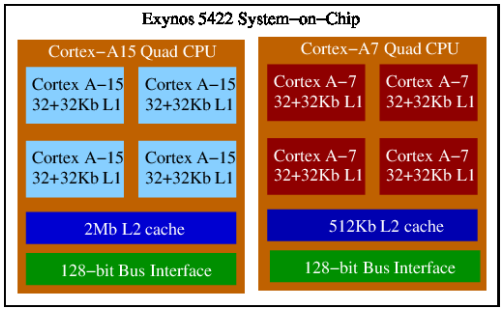


Figura 2.1: Diagrama de bloques del SoC Samsung Exynos 5422.

En los últimos años, además, la utilización de este tipo de arquitecturas de muy bajo consumo se ha convertido en un campo de investigación en pleno auge en la ruta hacia la construcción de futuros supercomputadores que alcancen la barrera del Exaflop [44, 46]. La construcción de este tipo de computadores y su posible idoneidad de cara a llegar a (y superar) dicha barrera los ha convertido en un candidato más que factible a la hora de desarrollar técnicas y códigos que exploten de manera eficiente todos los recursos computacionales que ofrecen.

2.1.2. Soporte en el kernel para arquitecturas big.LITTLE

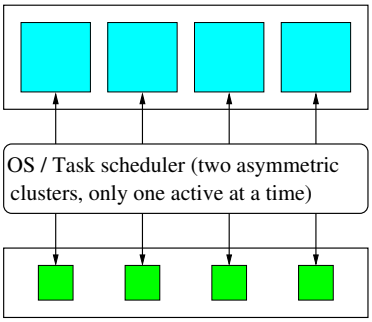
Las arquitecturas big.LITTLE modernas ofrecen un conjunto de distintos modelos de ejecución soportados por el sistema operativo (algunos de ellos exigen soporte también a nivel de hardware):

Cluster Switching Mode (CSM) El procesador se divide de forma lógica en dos clusters, uno conteniendo los núcleos rápidos, y otro conteniendo los núcleos lentos, pero sólo uno de ellos es utilizable simultáneamente en tiempo de ejecución. El sistema operativo activa/desactiva los clusters de forma transparente, en función de la carga de trabajo, para equilibrar el rendimiento y la eficiencia energética.

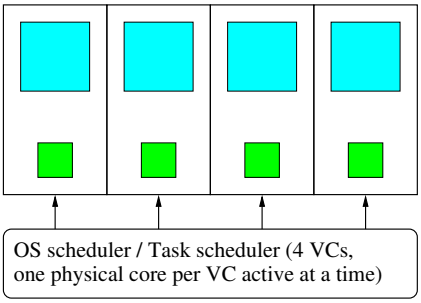
CPU migration (CPUM) Los núcleos físicos se agrupan en pares, cada uno formado por un núcleo rápido y otro lento, construyendo *Núcleos Virtuales (VC)*, a los que el sistema operativo mapea hebras. En un instante determinado, sólo un núcleo físico está activo por VC, dependiendo de los requisitos exigidos por la carga computacional activa. En aquellas implementaciones big.LITTLE en las que el número de núcleos lentos y rápidos no es el mismo, cada VC puede estar formado por diferente número de núcleos de cada tipo. La solución implementada por Linaro en el kernel de Linux se conoce como *In-Kernel Switcher (IKS)* [26].

Global Task Scheduling (GTS) Se trata del modelo más flexible. Todos los núcleos (lentos y rápidos) están disponibles para la planificación de hebras, y el sistema operativo las mapea en función de la naturaleza de la carga computacional asociada a cada uno de ellos y la disponibilidad de núcleos.

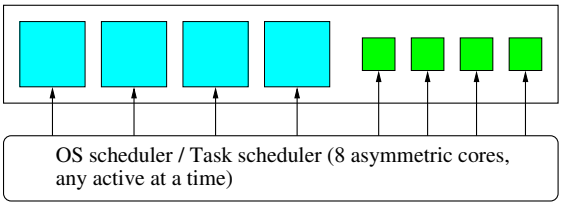
La Figura 2.2 ofrece una visión esquemática de estos tres modelos de ejecución sobre arquitecturas big.LITTLE modernas. GTS es la solución más flexible, ya que permite al planificador del sistema operativo asignar hebras a cualquiera de los núcleos disponibles, es decir, todos los núcleos disponibles se ofrecen al sistema operativo como candidatos a ejecutar el código de cualquier hebra, independientemente de su naturaleza o características. Esta característica permite migrar de forma muy sencilla aplicaciones multihebra ya existentes, incluyendo planificadores de tareas en tiempo de ejecución, y explotar todos los recursos computacionales en este tipo de procesadores asimétricos, utilizando cualquier mecanismo estándar de creación y gestión de hebras (por ejemplo, pthreads u OpenMP). Sin embargo, conseguir prestaciones óptimas no resulta tan sencillo, especialmente en aplicaciones multihebra en las que el desequilibrio de carga introducido por la asimetría de la arquitectura



(a) CSM



(b) CPUM



(c) GTS

Figura 2.2: Modos de operación de las arquitecturas big.LITTLE actuales.

puede penalizar el rendimiento obtenido. Solucionar este problema es uno de los objetivos de este trabajo.

De forma alternativa, CPUM propone una visión pseudosimétrica del procesador big.LITTLE; por ejemplo, en el caso del SoC Exynos 5422, los 8 núcleos asimétricos son vistos de forma lógica como 4 grupos de pares de procesadores, convirtiendo así la arquitectura en un sistema simétrico formado por 4 núcleos virtuales (VC), que son expuestos de esta manera al sistema operativo.

En la práctica, los planificadores de tareas en tiempo de ejecución pueden aproximar a nivel software cualquiera de estos tres paradigmas. Un modelo trivial puede seguir las directivas de GTS para asignar cualquier tarea lista para su ejecución a cualquiera de los núcleos disponibles en el sistema. Con esta solución, el desequilibrio de carga podría resolverse desarrollando políticas de planificación a nivel de tarea específicas (conscientes de la asimetría), para asignar tareas al recurso más apropiado en función de sus características.

Sin embargo, este trabajo propone un enfoque alternativo, en el que el SoC asimétrico es visto de forma lógica como un conjunto *simétrico* de núcleos virtuales, cada uno de ellos compuesto internamente por distintos tipos de núcleo, pero considerado, de cara al planificador, como un sistema simétrico. Esto facilita el desarrollo de políticas de planificación, como se verá más adelante, y permite incluso reutilizar planificadores ya existentes sobre este tipo de arquitecturas.

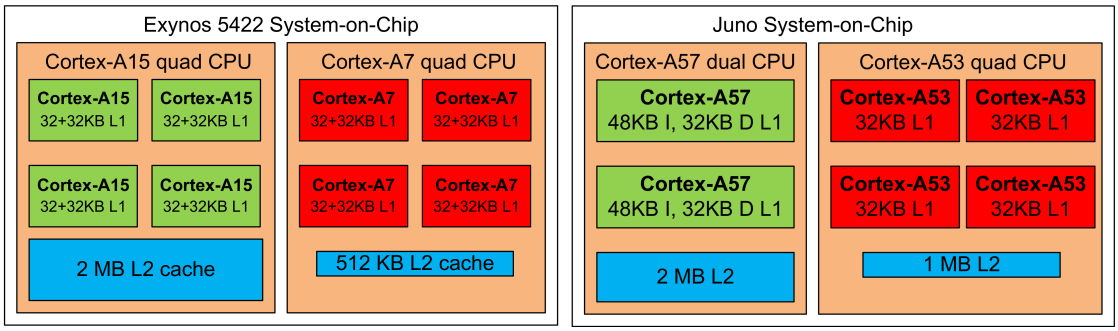
2.2. Descripción del entorno experimental

Durante el presente trabajo se han utilizado dos plataformas distintas basadas en el paradigma big.LITTLE de ARM. Se describen a continuación sus principales características, así como detalles adicionales acerca del software utilizado y el entorno experimental para la medición de consumo usado en parte del trabajo.

2.2.1. Plataformas hardware evaluadas

Odroid

Odroid-XU3 es una placa de desarrollo desarrollada por la empresa *Hardkernel*, equipada con un SoC Samsung Exynos 5422, que implementa una arquitectura big.LITTLE de 32 bits, compuesta por cuatro núcleos ARM Cortex-A15 funcionando a una frecuencia máxima de 1.3 GHz (formando el cluster de núcleos big), y cuatro núcleos ARM Cortex-A7 funcionando a una frecuencia máxima de 1.3 GHz (formando el cluster de núcleos LITTLE).



(a) Odroid (b) Juno

Figura 2.3: Diagramas de bloque para las plataformas Juno y Odroid.

Ambos tipos de núcleos implementan la microarquitectura ARMv7a, con procesamiento fuera de orden (en el caso del Cortex-A15) o en orden (en el caso del Cortex-A7). A nivel de jerarquía de memoria, los núcleos de cada cluster comparten una cache L2 de 2 Mbytes en el caso del cluster big y 512 Kbytes para el cluster LITTLE. En ambos casos, cada núcleo contiene una cache L1 de 64 Kbytes, divididos en 32 Kbytes de datos y 32 Kbytes de instrucciones. Por último, la jerarquía de memoria se completa con una memoria principal DDR3 con una capacidad de 2 Gbytes. La figura 2.3a muestra un esquema del SoC descrito.

Juno

Juno ARM Development Platform es una placa de desarrollo desarrollada por ARM, equipada con un SoC que implementa una arquitectura big.LITTLE de 64 bits, compuesta por dos núcleos ARM Cortex-A57 con una frecuencia máxima de 1.1 GHz (los cuales forman el cluster big), y cuatro núcleos ARM Cortex-A53 funcionando a una frecuencia máxima de 800 MHz (formando el cluster de núcleos LITTLE). Ambos tipos de núcleos implementan la microarquitectura ARMv8-A, donde los núcleos big realizan una ejecución fuera de orden con una profundidad del *pipeline* de 15 etapas, mientras que los núcleos LITTLE realizan una ejecución en orden con un *pipeline* de 8 etapas. Respecto a la jerarquía de memoria, cada cluster posee una cache L2 de tamaño 2 MBytes para el cluster big y de 1 MByte para el cluster LITTLE. Cada núcleo del cluster big posee una cache L1 divida en 48 Kbytes de instrucciones y 32 Kbytes de datos. Los núcleos del cluster LITTLE poseen todos ellos una cache L1 de 32 KBytes. Por último, la jerarquía se completa con una memoria principal DDR3 de tamaño 8 GBytes. El diagrama que describe esta arquitectura se muestra en la figura 2.3b.

2.2.2. Software utilizado

A continuación se muestra el software utilizado en cada una de las plataformas para realizar los distintos experimentos. El software base instalado en cada una de las máquinas es el siguiente:

	Odroid	Juno
Kernel	3.10.51+	3.10.63
Compilador GCC/G++	4.8	4.9.1

Adicionalmente al sistema base, se ha utilizado el siguiente software con las mismas versiones en ambas plataformas:

- Modificación del *runtime* Nanos++ [37] para incorporar los distintos experimentos realizados. Las modificaciones se han realizado sobre la versión 0.10a.
- Compilador fuente-a-fuente Mercurium [35] versión 2.0.0, que permite traducir el código anotado a un binario ejecutable por Nanos++.
- Biblioteca Extrae [17] para obtener trazas de ejecución de código anotado. Versión 3.2.1.
- Herramienta Paraver [41] para visualizar las trazas extraídas por Extrae. Versión 4.5.6.

2.2.3. Entorno de medición de consumo

Para el desarrollo del Capítulo 5 se han utilizado dos entornos experimentales de medición de consumo energético sobre las plataformas Odroid y Juno. Ambos entornos se basan en sendas adaptaciones del software `pmlib` [4], desarrollado por la Universidad Jaume I de Castellón; dicha herramienta, desarrollada en lenguaje Python, implementa un paradigma cliente/servidor, en el que la parte servidora se encarga de recoger continuamente muestras de potencia instantánea disipada por cierto dispositivo, mientras que la utilización de una API propia permite instrumentar los códigos a perfilar desde el punto de vista energético, que actúan como cliente realizando peticiones al servidor `pmlib`.

Ambos entornos difieren en la forma en la que las muestras de consumo energético son recogidas desde el sistema objetivo:

Odroid: El servidor `pmlib` ha sido adaptado para interactuar con los sensores de consumo energético integrados en la placa y basados en resistencias de *shunt*. Dichos sensores

ofrecen lecturas independientes para el consumo energético del cluster formado por los Cortex-A7, Cortex-A15, GPU y RAM, con una frecuencia de refresco de cuatro muestras por segundo.

JUNO: El servidor `pmlib` ha sido adaptado para recoger los datos de consumo desde un DAQ fabricado por National Instruments. Dicho dispositivo, a su vez, recoge muestras de potencia instantánea a partir de las resistencias de *shunt* integradas en la plataforma, con lecturas actualizadas con una frecuencia de 1000 muestras por segundo e independientes para el cluster formado por los Cortex-A53, Cortex-A57, GPU y resto del sistema.

Capítulo 3

Estrategias para la extracción de paralelismo en arquitecturas heterogéneas y asimétricas

En este capítulo se describen distintos enfoques software a la hora de explotar el paralelismo existente en las arquitecturas paralelas más extendidas actualmente, incluyendo sistemas multinúcleo y arquitecturas heterogéneas con memoria compartida, haciendo especial hincapié en los mecanismos existentes a la hora de explotar sistemas asimétricos. Se describen las dos soluciones con mayor aceptación a día de hoy. La Sección 3.1 introduce los mecanismos e implementaciones más extendidas para la extracción y explotación de paralelismo a nivel de tareas. La Sección 3.2 incide en el desarrollo de bibliotecas paralelas, extrayendo paralelismo a nivel de datos. En ambos casos, se tomará como hilo conductor un ámbito concreto: el desarrollo de códigos matemáticos dentro del ámbito del álgebra lineal, ya que serán estos códigos los utilizados en la descripción del trabajo realizado en próximos capítulos. Finalmente, la Sección 3.3 realiza una breve comparativa entre ambos enfoques, destacando las ventajas e inconvenientes de cada uno de ellos desde el punto de vista del rendimiento y de la facilidad de programación.

3.1. Modelos de programación basados en paralelismo a nivel de tareas

El DAG (*directed acyclic graph*) asociado con un algoritmo o rutina es una representación gráfica del paralelismo de tareas asociado a ella, e, idealmente, un planificador de tareas

en tiempo de ejecución podría explotar esta información para determinar distintas planificaciones (esto es, ordenación de tareas y asignación de las mismas a recursos computacionales disponibles) que satisfacen las dependencias fijadas en el DAG.

3.1.1. Introducción al paralelismo a nivel de tareas. Estado del arte

Los modelos de programación basados en la extracción de paralelismo a nivel de tareas han demostrado su utilidad, particularmente en el campo del álgebra lineal [1]. En general, estos modelos proporcionan al programador en primer lugar un mecanismo para expresar el paralelismo potencial a nivel de tareas existente en la aplicación, normalmente en forma de grafo de tareas, y en segundo lugar un planificador de tareas (o runtime) que realice una planificación dinámica de las mismas, respetando las dependencias de datos entre ellas y a la vez permitiendo un correcto aprovechamiento de las unidades de procesamiento disponibles.

Entre las propuestas e implementaciones más extendidas para este tipo de modelo de programación cabe destacar, entre otros, Cilk Plus, StarPU, Superglue, QUARK, Kaapi y OmpSs.

Cilk Plus [8, 11, 18, 32] es un lenguaje diseñado exclusivamente para programación paralela basado en C, con extensiones que permiten expresar tanto el paralelismo a nivel de tareas como a nivel de datos. Cilk incorpora un componente software para la planificación de tareas en tiempo de ejecución que aprovecha el paralelismo expuesto por el programador. Fue desarrollado en los años 90 en el MIT, distribuido de forma comercial más tarde por Intel, y liberado e integrado en proyectos de software libre (por ejemplo, GCC) más tarde. En su versión actual, Cilk no incorpora ningún soporte específico para arquitecturas heterogéneas ni asimétricas.

StarPU [2, 3, 6, 48] implementa un planificador de tareas en tiempo de ejecución y da soporte de cara al programador a la hora de anotar tareas mediante pequeños fragmentos de código (*codelets*). Aunque el runtime se ha aplicado frecuentemente a la resolución de algoritmos numéricos, es de carácter generalista, y gestiona las transferencias de datos entre espacios de memoria (en arquitecturas heterogéneas) mediante una biblioteca que fuerza la coherencia de memoria e incluye prebúsqueda automática de datos.

Superglue [49–51] es también un modelo basado en tareas, en el que las dependencias de datos se representan usando versiones en lugar de grafos de dependencias. El runtime realiza la planificación de las tareas considerando que todos los procesadores son idénticos, pero tiene en cuenta el uso de recursos de cada tarea para evitar problemas de contención.

Quark (*QUeing And Runtime for Kernels*) [24, 42, 56] es un runtime que permite la ejecución dinámica de tareas con dependencias de datos en sistemas multinúcleo con memoria

compartida; forma parte de la biblioteca de álgebra lineal PLASMA [1].

Kaapi (*Kernel for Adaptive, Asynchronous Parallel and Interactive Programming*) [20, 21, 28] es una biblioteca C++ para programación basada en tareas similar a Cilk, desarrollada en el INRIA. Existen versiones de su planificador tanto para sistemas multihebra como multi-GPU (XKaapi), cuyo modelo de programación permite crear tareas de forma similar a Cilk, StarSs u OmpSs. El paralelismo es explícito y las dependencias entre tareas y transferencias entre espacios de memoria son gestionadas automáticamente por el planificador.

3.1.2. OmpSs. Modelo de programación y planificación de tareas

OmpSs [16, 38] es uno de los modelos de programación paralela a nivel de tareas más extendidos a día de hoy. A grandes rasgos, el diseño del modelo de programación se basa en la inclusión de directivas (o `pragma`s) similares a las utilizadas en otros modelos de programación paralela, como OpenMP. Sin embargo, dichas directivas se limitan en su mayoría, a la anotación de ciertos bloques de código (típicamente funciones) para informar de su carácter de *tareas*, es decir, unidades básicas de planificación a los recursos computacionales disponibles. Dicha planificación se difiere hasta el momento de la ejecución del código, en el que un *planificador de tareas* o *runtime* (en el caso de OmpSs, llamado Nanox) se encarga de mapear, de forma eficiente, su ejecución al mejor recurso computacional disponible en un momento dado.

El modelo de programación OmpSs

Con el fin de ayudar al programador en la construcción de un DAG completo y correcto partiendo de un código secuencial, OmpSs proporciona mecanismos sencillos y no invasivos. En dicho modelo, el programador utiliza directivas similares a OpenMP (`pragma`s) para anotar rutinas existentes en el código como tareas, indicando a la vez la direccionalidad de sus operandos (entrada, salida o entrada/salida) a través de cláusulas adicionales. A grandes rasgos, el planificador de OmpSs descompone el código (transformado previamente a través del compilador fuente-a-fuente Mercurium [35]) en un conjunto de tareas, identificando las dependencias entre ellas, y lanzando a ejecución únicamente *tareas listas* (es decir, aquellas cuyas dependencias han sido satisfechas) para su ejecución en los distintos núcleos computacionales del sistema. Adicionalmente a las directivas proporcionadas para la creación de tareas, OmpSs proporciona un conjunto de directivas orientadas a la sincronización de las distintas tareas.

Planificación de tareas en OmpSs. El *runtime* Nanox

Una vez que un código anotado ha sido compilado con el compilador de OmpSs Mercurium, éste puede ser ejecutado a través del *runtime* de OmpSs denominado Nanox, consistente en una serie de librerías encargadas de controlar la ejecución del programa e intentar finalizar la ejecución de la manera más eficiente posible. Para ello, Nanox está compuesto por un núcleo principal encargado de la gestión de los distintos hilos y estructuras internas, y un conjunto de módulos encargados de distintas tareas como puede ser el módulo encargado de la planificación de las tareas sobre los hilos, el módulo encargado de comprobar las dependencias entre tareas, o el módulo encargado de obtener distintas trazas durante la ejecución.

Una vez que un programa anotado es lanzado sobre Nanox, éste realiza una serie de pasos para inicializar el *runtime* antes de comenzar la ejecución del programa original. Entre estos pasos, uno de ellos es la creación de un número fijo de hilos denominados *worker threads*, los cuales van a ser los encargados de ejecutar las distintas tareas que estén listas para ejecución (el número de hilos creados es escogido por el usuario en el momento de iniciar la ejecución). El primero de estos *worker threads* creados, a parte de ejecutar tareas igual que el resto de *worker threads*, también es el encargado de ejecutar el código secuencial del programa, por lo que mientras que el resto de hilos pueden estar ociosos en el caso de no existir ninguna tarea lista para ejecución, este hilo siempre estará ocupado. Notar que al contrario que en el modelo de ejecución usado por OpenMP, Nanox no crea un nuevo hilo por cada tarea a ejecutar, sino que crea un conjunto de hilos al inicio de la aplicación, y son éstos los encargados de ejecutar todas las tareas del programa, finalizando su vida al acabar la ejecución de la misma.

Cuando un *worker thread* finaliza la ejecución de una tarea, éste realiza dos acciones sobre el *runtime*:

1. El *worker thread* informa al *runtime* de la finalización de la tarea. Al finalizar la tarea, el módulo encargado de comprobar las dependencias comprueba qué nuevas tareas están listas para ser ejecutadas ya que se han satisfecho todas sus dependencias. Cuando se determina que una tarea está lista para ejecución, se informa al módulo encargado del planificador para que la almacene y trate de acuerdo a la política que implemente el módulo cargado.

2. El *worker thread* solicita una nueva tarea para ejecutar al módulo planificador. El módulo planificador, en función de la política que se haya decidido usar, y del *worker thread* que solicite la tarea, decidirá qué tarea debe ejecutar. En caso de no existir

ninguna tarea lista para ejecutar, o de que el *worker thread* no sea el idóneo para ejecutar ninguna de las tareas listas existentes, éste pasa a estado inactivo hasta la creación de nuevas tareas, evitando así consumir recursos de manera innecesaria.

La creación de nuevas tareas tiene lugar en el momento en el que un *worker thread* en ejecución se encuentra una anotación para la creación de una nueva tarea. En este momento, el *worker thread* informa al *runtime* para que cree la tarea correspondiente y la inserte al DAG del problema, esperando a que todas sus dependencias sean satisfechas y la nueva tarea esté lista para su ejecución.

3.1.3. Ejemplo: paralelización de la factorización de Cholesky

A continuación describimos los mecanismos utilizados para extraer paralelismo a nivel de tareas durante la ejecución de una operación específica de álgebra lineal densa, utilizando la Factorización de Cholesky como hilo conductor. Esta operación en particular es representativa de muchas otras factorizaciones utilizadas en la resolución de sistemas lineales, por lo que gran parte de las conclusiones extraídas sobre ella pueden ser aplicadas a otro tipo de operaciones similares ampliamente utilizadas en ciencia e ingeniería. La Factorización de Cholesky descompone una matriz simétrica definida positiva A en el producto $A = U^T U$, donde el factor de Cholesky U de dimensión $n \times n$ es una matriz triangular superior [22].

Listado 3.1: Implementación en lenguaje C de la factorización de Cholesky orientada a bloques.

```
1  void cholesky (double *A[s][s], int b, int s)
2  {
3     for (int k = 0; k < s; k++) {
4
5        po_cholesky (A[k][k], b, b);            // Fact. Cholesky
6                                                // (bloque diagonal)
7        for (int j = k + 1; j < s; j++)
8           tr_solve (A[k][k], A[k][j], b, b);  // Sol. Sist. Triangular
9
10       for (int i = k + 1; i < s; i++) {
11          for (int j = i + 1; j < s; j++)
12             ge_multiply (A[k][i], A[k][j],
13                          A[i][j], b, b);      // Mult. Matrices
14          sy_update (A[k][i], A[i][i], b, b); // Act. Simetrica rango-b
15       }
16
17    }
18 }
```

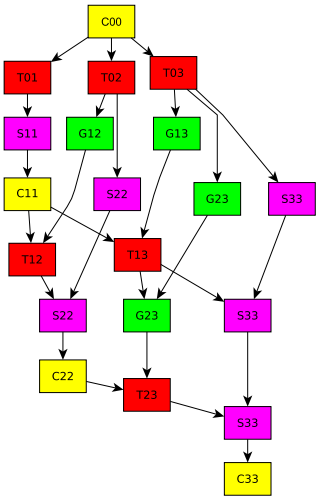


Figura 3.1: DAG con las tareas y dependencias de datos resultantes de la aplicación del código del Listado 3.1 sobre una matriz compuesta por 4×4 bloques (`s`=4). Las etiquetas especifican el tipo de kernel/tarea con la siguiente correspondencia: "C" para la Factorización de Cholesky, "T" para la resolución de sistema triangular, "G" para la multiplicación de matrices, y "S" para la actualización simétrica de rango-`b`. Los subíndices (comenzando en 0) especifican la submatriz que la tarea correspondiente actualiza.

El código del Listado 3.1 muestra un código simplificado en lenguaje C para la factorización de una matriz `A` de dimensiones `n`×`n`, almacenada como un conjunto de `s`×`s` submatrices (bloques de datos) de dimensión `b`×`b`.

Esta rutina orientada a bloques descompone la operación global en una colección de kernels u operaciones fundamentales: `po_cholesky` (Factorización de Cholesky), `tr_solve` (resolución de un sistema triangular), `ge_multiply` (multiplicación de matrices), y `sy_update` (actualización simétrica de rango-`b`), que operan sobre cada bloque de datos de la matriz, y forman parte de las rutinas proporcionadas por los estándares LAPACK [30] y BLAS [39].

El orden en el que dichos kernels son invocados durante la ejecución de la rutina, así como las submatrices o bloques que cada kernel lee o escribe, genera un grafo acíclico dirigido (DAG) que refleja las dependencias entre tareas (esto es, entre instancias de los kernels) y, por tanto, el paralelismo de tareas potencial de la operación. Por ejemplo, la Figura 3.1 muestra el DAG con las tareas (nodos) y las dependencias de datos (arcos) intrínsecas a la ejecución del código del Listado 3.1 cuando éste se ejecuta sobre una matriz compuesta por 4×4 submatrices (esto es, cuando `s`=4).

El código del Listado 3.2 muestra las anotaciones necesarias por parte del programador para explotar el paralelismo a nivel de tareas inherente a la factorización de Cholesky usan-

do OmpSs; cabe destacar las líneas etiquetadas con anotaciones (directivas) "`#pragma omp task`". Las cláusulas `in`, `out` e `inout` denotan la direccionalidad de los datos, y ayudan al planificador de tareas a mantener coherentemente las dependencias entre tareas durante la ejecución. En esta implementación, los cuatro kernels anotados son implementados como invocaciones a kernels computacionales básicos de álgebra lineal proporcionados por LAPACK (`dpotrf`) y BLAS (`dtrsm`, `dgemm` y `dsyrk`).

Listado 3.2: Tareas etiquetadas necesarias para la factorización de Cholesky por bloques.

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

3.1.4. Adaptación de OmpSs a arquitecturas asimétricas

En servidores equipados con uno o más aceleradores –por ejemplo, procesadores gráficos–, versiones especializadas de los planificadores de tareas que acompañan a OmpSs, StarPU, MAGMA, Kaapi y `libflame` [57] son capaces de planificar tareas a núcleos de propósito general (CPUs en adelante) o a aceleradores (por ejemplo, procesadores gráficos –GPUs– o Intel Xeon Phi), asignando tareas a cada tipo de recurso en función de sus propiedades, y aplicando técnicas como caches gestionadas por software o mapeado de tareas explotando la localidad de datos (véanse, entre otros, [6, 7, 21, 43, 52]). Este tipo de adaptaciones, además, permiten la gestión transparente de transferencias de datos entre distintos espacios de memoria, característica principal en sistemas heterogéneos modernos.

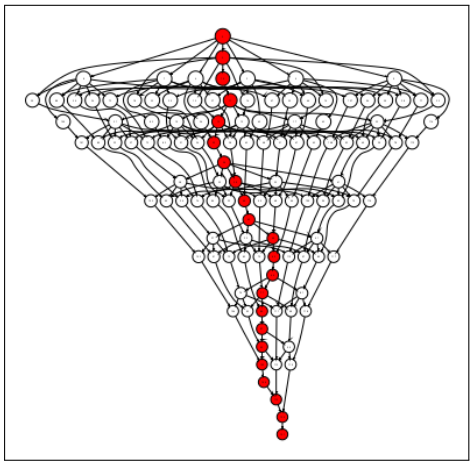
Con el reciente auge de las arquitecturas asimétricas en el mundo HPC, el equipo de desarrollo de OmpSs ha introducido recientemente un nuevo planificador denominado *Bottom level-aware scheduler* (Botlev) [10] específico para este nuevo tipo de arquitecturas. Botlev recoge las ideas de los planificadores tradicionales basados en arquitecturas heterogéneas, distinguiendo únicamente dos tipos de nodos de cómputo (un nodo rápido formado por los núcleos de tipo big, y un nodo de cómputo lento formado por los núcleos de tipo LITTLE) y eliminando el cálculo de los costes asociados a la transferencia de datos.

Una técnica utilizada para obtener un mejor rendimiento en programas paralelos basados en tareas es intentar que las tareas críticas finalicen su ejecución en el menor tiempo posible. Se denominan tareas críticas de un DAG a aquellas tareas que en caso de retrasar su ejecución, producen un retraso en la ejecución del problema. El planificador Botlev persigue este objetivo intentando calcular de manera dinámica qué tareas pertenecen al camino crítico del DAG asociado al problema, y ejecutar estas tareas sobre núcleos rápidos con el objetivo de finalizar su ejecución cuanto antes. Además, en caso de existir más tareas no críticas, las tareas críticas tendrán preferencia frente a las tareas no críticas. El objetivo de garantizar que las tareas del camino crítico se ejecutan en los núcleos rápidos cuanto antes es asegurarse que al ejecutar las tareas críticas, éstas liberarán dependencias con nuevas tareas que pasarán a estar listas para ejecución, y así intentar conseguir que nunca haya *worker threads* ociosos a causa de que no existan tareas listas para ejecutar, lo cual disminuiría el rendimiento global de la aplicación. La principal diferencia entre Botlev y los planificadores tradicionales para sistemas heterogéneos es que Botlev toma las decisiones de forma dinámica sin necesidad de conocer de antemano información sobre las tareas, o la forma del árbol de dependencias.

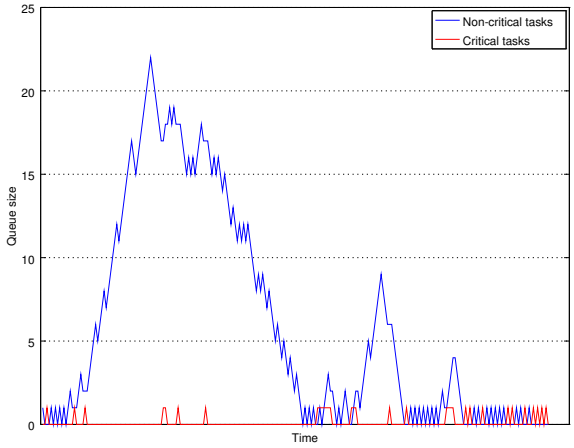
Para determinar si una tarea pertenece al camino crítico o no en tiempo de ejecución,

Botlev asigna una prioridad a cada tarea en el momento de la inserción en el grafo de dependencias, y la actualiza con la creación de nuevas tareas. Una vez que la tarea está lista para ser ejecutada (es decir, cuando todas las tareas que tenían relación con la tarea mediante el grafo de dependencias han finalizado su ejecución), a partir de su prioridad se toma la decisión de si la tarea es considerada crítica o no.
La prioridad de una tarea viene dada por un número entero positivo, el cual representa la longitud del camino más largo desde la tarea actual hasta una tarea hoja del grafo de dependencias. Cuando una tarea es introducida en el DAG asociado al problema, se le asigna una prioridad 0 ya que la longitud del camino más largo desde la tarea a un nodo hoja (ella misma) es 0. Además, cuando una nueva tarea es insertada en el árbol, se actualizan todas las tareas predecesoras, ya que las prioridades de éstas pueden haber cambiado: por cada tarea predecesora, se intenta aumentar la prioridad en 1 unidad (el camino es un nodo más largo), siempre que no tuviera una prioridad mayor antes (este es el caso de que la tarea pertenezca a un camino más largo en el que la nueva tarea no esté involucrada). El proceso de actualización finaliza cuando se actualiza la tarea raíz del árbol, o cuando se alcanza una tarea que pertenece a otro camino más largo que el actual. En la Figura 3.2a se puede ver el camino crítico detectado para una factorización de Cholesky sobre una matriz dividida en 8×8 bloques. Hay que destacar que esta forma de calcular el camino crítico detecta las tareas que pertenecen al camino más largo, pero eso no implica que sean aquellas que más van a retrasar la ejecución en caso de que no se ejecuten de manera prioritaria. Para calcular esto, sería necesario conocer de antemano las necesidades de cada tarea, o ir almacenando resultados parciales de ejecución de manera dinámica para tomar decisiones en el futuro.

Una tarea puede ser ejecutada cuando todas sus tareas predecesoras han finalizado la ejecución. Cuando esto ocurre, Botlev inserta la tarea en una cola de tareas pendientes para ir ejecutándolas en función de la prioridad asignada, o en orden de inserción en caso de empate. Estas tareas son asignadas a los diferentes núcleos según vayan finalizando la ejecución de las tareas previas. Según la prioridad de la tarea, Botlev distingue dos tipos de colas en las que insertar una tarea: la cola con las tareas pertenecientes al camino crítico, y la cola con el resto de tareas. Una tarea se considera que pertenece al camino crítico si posee una prioridad mayor a cualquiera de las prioridades de las tareas anteriores, o si es hija directa de una tarea que se ha considerado crítica y tiene una prioridad de una unidad menor (es decir, es el siguiente nodo del camino crítico). En la Figura 3.2b se puede ver la evolución del tamaño de las colas para una factorización de Cholesky sobre una matriz de 8×8 bloques. La línea azul muestra el número de tareas no críticas listas para ser ejecutadas en función del momento de la ejecución del problema, mientras que la línea roja muestra



(a) Camino crítico en BOTLEV para una factorización de Cholesky de 8×8 bloques. Fuente: [10]



(b) Evolución del tamaño de las colas de tareas listas.

Figura 3.2: Cálculo de tareas críticas en Botlev y asignación a núcleos.

el número de tareas críticas. Como se puede apreciar, a mitad de ejecución el número de tareas listas para ejecución disminuye considerablemente, y hasta que un conjunto de tareas detectadas como críticas son ejecutadas, el número de tareas listas no vuelve a aumentar. Este es un ejemplo de cómo retrasar la ejecución de una tarea crítica supone retrasar la creación de tareas, y por tanto, tener un impacto negativo en el rendimiento.

Cuando un core big finaliza la ejecución de una tarea, éste ejecutará la siguiente tarea de la cola de tareas críticas, mientras que un core LITTLE ejecutará la primera tarea de la cola de tareas no críticas. De esta forma se consigue asignar las tareas críticas a núcleos big, mientras que las tareas no críticas serán ejecutadas por núcleos lentos. Para grafos de dependencias muy anchos, como puede ser el asociado a una factorización de Cholesky, el número de tareas críticas es mucho menor que el número de tareas no críticas, dando lugar a que los núcleos rápidos estén la mayor parte de tiempo ociosos. Para evitar esta situación, si un core big no dispone de tareas críticas que ejecutar, ejecutará la siguiente tarea de la lista de tareas no críticas. El robo de tareas en sentido contrario (es decir, que los núcleos LITTLE ejecuten tareas críticas) es una parámetro adicional que se puede seleccionar en tiempo de ejecución, pero que se encuentra desactivado por defecto.

3.2. Implementación paralela de bibliotecas matemáticas

3.2.1. BLIS: implementación multihebra del estándar BLAS

A la hora de explotar el paralelismo potencial de una arquitectura, una segunda alternativa al enfoque basado en planificación a nivel de tareas (es decir, explotando paralelismo a nivel de tareas) consiste en el uso de kernels ya paralelizados, es decir, implementaciones de rutinas que particionan estáticamente el trabajo entre los recursos computacionales existentes, o lo hacen aprovechando mecanismos sencillos de planificación como aquellos ofrecidos por OpenMP. Desde este punto de vista, el paralelismo no se extrae ya a nivel de tarea, sino a nivel de datos, de forma interna a cada tarea. Por ejemplo, en el ámbito del álgebra lineal, donde las dependencias de datos internos a cada tarea son sencillas, o cuando el número de núcleos en el sistema es reducido, esta opción típicamente conlleva menos sobrecarga en tiempo de ejecución, y por tanto proporciona una solución más eficiente desde el punto de vista del rendimiento. A día de hoy, ésta es la opción preferida y adoptada en bibliotecas comerciales y de software libre, como por ejemplo AMD ACML [5], IBM ESSL [25], Intel MKL [27], GotoBLAS [23], OpenBLAS [39] o BLIS [55]. Todas estas bibliotecas son implementaciones optimizadas de los estándares BLAS [14, 15, 31] y LAPACK [34].

A modo de ejemplo, y dado que utilizaremos BLIS como biblioteca subyacente para la ejecución de tareas individuales durante el resto del trabajo, se describe brevemente su funcionamiento, así como la posibilidad de adaptar su paralelización a una arquitectura asimétrica.

BLIS implementa el enfoque algorítmico introducido por la biblioteca de álgebra lineal GotoBLAS, ampliamente utilizada a día de hoy en multitud de arquitecturas. Este enfoque plantea la implementación de todas las rutinas BLAS de nivel 3 (incluyendo el producto de matrices, GEMM) como tres bucles anidados sobre dos rutinas de empaquetado de datos, cuya finalidad es dirigir bloques de los operandos fuente a través de la jerarquía de memoria, y un *macro-kernel* cuya función es la ejecución de las operaciones aritméticas asociadas a cada operación. Internamente, BLIS implementa dicho macro-kernel como dos bucles adicionales sobre un *micro-kernel* que, a su vez, está formado por un bucle sobre una actualización de rango 1.

Para la siguiente descripción, únicamente consideraremos los tres bucles externos en la implementación de BLIS para GEMM, para el producto $C := C + A \cdot B$, donde A, B, C son matrices de dimensión $m \times k$, $k \times n$ y $m \times n$ respectivamente, almacenadas en los arrays `A`, `B` y `C`; véase el código del Listado 3.3. En dicho código, `mc`, `nc`, `kc` son parámetros de configuración que necesitan ser sintonizados para cada arquitectura teniendo en cuenta,

entre otros, factores como la latencia de las unidades de punto flotante, número de registros vectoriales y tamaño/grado de asociatividad de cada nivel de cache [33].

Listado 3.3: Implementación de altas prestaciones de GEMM en BLIS.

```
1  void gemm (double A[m][k], double B[k][n], double C[m][n],
2             int m,  int n,  int k, int mc, int nc, int kc)
3  {
4     double *Ac = malloc (mc * kc * sizeof (double)),
5            *Bc = malloc (kc * nc * sizeof (double));
6
7     for (int jc = 0; jc < n; jc+=nc) {              // Bucle 1
8        int jb = min(n-jc+1, nc);
9
10       for (int pc = 0; pc < k; jc+=kc) {           // Bucle 2
11          int pb = min(k-pc+1, kc);
12
13          pack_buffB (B[pc][jc], Bc, kb, nb);       // Empaquetar A->Ac
14
15          for (int ic = 0; ic < m; ic+=mc) {        // Bucle 3
16             int ib = min(m-ic+1, mc);
17
18             pack_buffA (A[ic][pc], Ac, mb, kb);    // Empaquetar A->Ac
19
20             gemm_kernel (Ac, Bc, C[ic][jc],
21                          mb, nb, kb, mc, nc, kc);  // Macro-kernel
22          }
23       }
24    }
25 }
```

3.2.2. Implementación multihebra consciente de la asimetría

La implementación de GEMM en BLIS obtiene un rendimiento cercano al rendimiento pico de la arquitectura, tanto en sistemas multinúcleo como sistemas con aceleradores [47, 54]. Estos estudios han derivado en versiones de la implementación conscientes de la asimetría sobre arquitecturas ARM big.LITTLE bajo el modelo de ejecución GTS. Concretamente, la versión multihebra consciente de la asimetría descrita en [9] integra las siguientes tres técnicas:

- Un particionado dinámico 1-D del espacio de iteraciones para distribuir la carga de los bucles 1 y 3 sobre los dos clusters (big y LITTLE).
- Un particionado estático 1-D del espacio de iteraciones para distribuir la carga de uno de los bucles internos entre los núcleos del mismo cluster.

- Una modificación de los árboles de control que gobiernan la paralelización multihebra en BLIS para utilizar distintos *strides* en los bucles, de modo que éstos se adapten a cada una de las arquitecturas subyacentes.

En general, esta estrategia puede ser aplicada a AMPs genéricos, consistentes en cualquier combinación de núcleos lentos y rápidos, y a cualquiera de las rutinas BLAS implementadas en la biblioteca. Así, BLIS se convierte en una biblioteca que implementa rutinas básicas de álgebra lineal que, internamente, explotan la asimetría de la arquitectura subyacente; esta característica será aprovechada en el resto del trabajo en combinación con planificadores de tareas "clásicos" (no conscientes de la asimetría), como se describe en la siguiente sección.

3.3. Paralelismo a nivel de tareas y a nivel de datos: ventajas e inconvenientes

Existen pues dos alternativas principales a la hora de portar una determinada implementación (en nuestro caso, de una operación de álgebra lineal) a una plataforma asimétrica:

1. Utilizar un planificador de tareas consciente de la asimetría, en el que, por ejemplo, las tareas pertenecientes al camino crítico sean asignadas a núcleos de procesamiento rápidos. De este modo, la unidad básica de planificación de tareas es el núcleo individual, las tareas son invocaciones a versiones secuenciales de una determinada biblioteca, y es el planificador de tareas el encargado de explotar en tiempo de ejecución, sin la intervención del programador, los núcleos asimétricos subyacentes.

2. Utilizar una implementación de rutinas de biblioteca conscientes de la asimetría, sin extraer paralelismo a nivel de tareas. Una determinada invocación a una de estas rutinas se ejecutará eficientemente de forma paralela teniendo en cuenta las características de la arquitectura asimétrica subyacente, repartiendo su espacio de iteraciones, por ejemplo, según las capacidades de cómputo de cada tipo de núcleo disponible.

Las ventajas de la utilización de un planificador de tareas son obvias: menor intervención por parte del programador o usuario, adaptación a pequeños cambios en el comportamiento dinámico de la arquitectura ante la ejecución de un código, o modificación de las políticas de planificación dinámicamente, entre otras. Sin embargo, en muchas ocasiones, el sobrecoste de este tipo de mecanismos no es desdeñable; además, su adaptación a arquitecturas heterogéneas o asimétricas, como es el caso, requiere un desarrollo específico de nuevas políticas de planificación que exploten los recursos disponibles de manera eficiente.

Una biblioteca de funciones específicamente desarrollada para una arquitectura asimétrica (por ejemplo, BLIS en el ámbito del álgebra lineal) elimina gran parte de este sobrecoste y suele conseguir mejores y más predecibles rendimientos, especialmente en arquitecturas no homogéneas. Sin embargo, su desarrollo conlleva implementaciones específicas para cada tipo de arquitectura, lo que supone un esfuerzo no despreciable.

En el siguiente capítulo, se propone una solución que combina las ventajas de cada uno de los dos enfoques: combinando un planificador no consciente de la asimetría (es decir, convencional) con una biblioteca para la ejecución de tareas optimizada para arquitecturas asimétricas, se consigue aunar facilidad de programación y uso –debido a la utilización de un planificador de tareas estándar– con altas prestaciones –derivadas de la utilización de una biblioteca desarrollada ad-hoc para este tipo de arquitecturas–.

Capítulo 4

Optimización del rendimiento de OmpSs sobre arquitecturas asimétricas

4.1. Planteamiento y objetivos generales

Este capítulo presenta un nuevo acercamiento hacia la utilización de planificadores de tareas no conscientes de la asimetría (en adelante nos referiremos a ellos como *convencionales*) sobre arquitecturas asimétricas, explotando toda la capacidad de cómputo de las mismas sin que ello suponga una adaptación específica de las políticas de planificación de los mismos. Para ello, se aprovechan implementaciones de tareas conscientes de la asimetría, abstrayendo al runtime de dicha característica.

En primer lugar, se llevará a cabo una evaluación inicial de la implementación por bloques de la factorización de Cholesky mostrada anteriormente en los Listados 3.1 y 3.2, utilizando el planificador convencional de OmpSs y una versión secuencial de BLIS. Las observaciones de dicho estudio servirán como base para la propuesta de la solución introducida, consistente en el uso de un planificador convencional enlazado con una versión asimétrica de BLIS. A continuación, se comparan los resultados obtenidos con los de los mismos códigos utilizando una versión de OmpSs consciente de la arquitectura; finalmente, estos resultados son analizados en detalle a través de trazas de ejecución.

4.1.1. Evaluación de runtimes convencionales en AMPs

La Figura 4.1 muestra el rendimiento, en términos de GFLOPS (miles de millones de operaciones en coma flotante por segundo), mostrado por el runtime convencional de OmpSs para la factorización de Cholesky, variando el número de *worker threads* entre 1 y 8 sobre

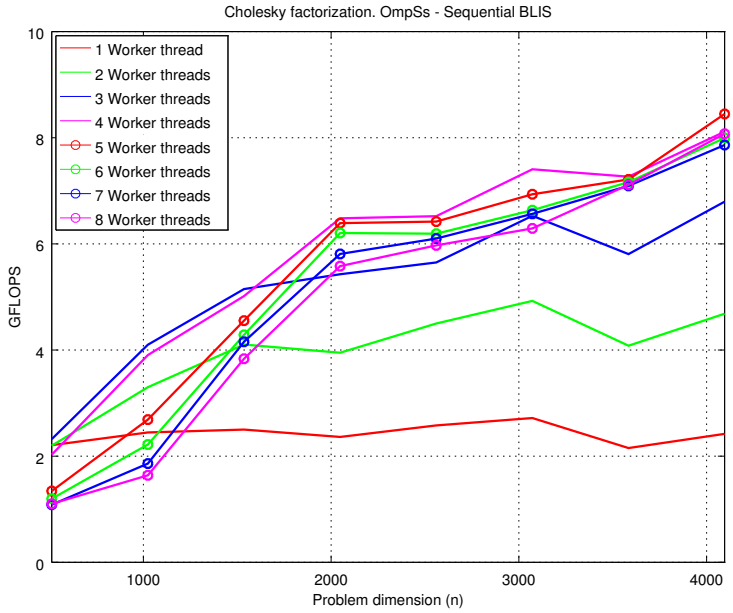


Figura 4.1: Rendimiento de la factorización de Cholesky utilizando el runtime OmpSs convencional y una implementación secuencial de BLIS sobre el SoC Exynos 5422.

la plataforma asimétrica ODROID; en el experimento, se delega la asignación de *worker threads* a núcleos al planificador del sistema operativo. En otras palabras, se está utilizando una versión no consciente de la asimetría del planificador de tareas en OmpSs. Para este experimento, se ha evaluado un rango lo suficientemente amplio de tamaños de bloque (`b` en el código del Listado 3.1), aunque por simplicidad se reportan únicamente los resultados obtenidos para el valor de `b` que optimiza el ratio de GFLOPS para cada dimensión del problema[1].

Los resultados experimentales revelan un incremento en el rendimiento a medida que el número de *worker threads* aumenta entre 1 y 4, casos en los que el planificador del sistema operativo asigna su ejecución a los núcleos rápidos (Cortex-A15) disponibles. Sin embargo, cuando el número de *worker threads* excede la cantidad de núcleos rápidos, el sistema operativo se ve forzado a asignar *worker threads* a núcleos lentos (Cortex-A7), en cuyo caso el aumento en prestaciones desaparece, e incluso éstas disminuyen drásticamente a medida que el número de *worker threads* aumenta. La principal causa de este decremento en las prestaciones es el desequilibrio de carga, ya que tareas con granularidad uniforme, posiblemente pertenecientes al camino crítico, son asignadas a núcleos lentos.

Este experimento revela la principal motivación por la que es necesario un planificador

[1]En este capítulo, todos los experimentos se han realizado utilizando doble precisión.

de tareas consciente de la arquitectura y adaptado a ella [10]. Sin embargo, en el presente capítulo, se desarrollará un enfoque alternativo a la hora de explotar la asimetría de la arquitectura subyacente: se utilizará un planificador de tareas convencional en combinación con una biblioteca asimétrica para la ejecución de tareas, como se describe a continuación.

4.2. Combinación de runtimes convencionales con bibliotecas asimétricas

4.2.1. Visión general de la propuesta

La propuesta de operación introducida en el presente capítulo se ejecuta bajo el modelo GTS, pero está en cierto modo inspirada en el modelo CPUM (véase Sección 2.1.2). Más concretamente, el objetivo es que el planificador de tareas considere la arquitectura ODROID (por ejemplo) como una arquitectura verdaderamente simétrica, formada por cuatro VCs (núcleos virtuales, o *Virtual Cores*), cada uno de ellos compuesto por un núcleo rápido y un núcleo lento. Para ello, a diferencia del modelo CPUM, *ambos* núcleos físicos dentro de cada VC se mantendrán activos y colaborarán en la ejecución de una determinada tarea. Así, la propuesta explota dos niveles de concurrencia: el *paralelismo a nivel de tareas* es extraído por el planificador para asignar tareas a cada uno de los cuatro VCs idénticos disponibles en el sistema, e internamente, cada tarea/kernel divide su carga de forma correcta para exponer *paralelismo a nivel de datos*, distribuyendo la carga de trabajo entre los dos núcleos físicos asimétricos dentro del VC que está a cargo de la ejecución de dicha tarea.

Esta solución requiere únicamente un planificador de tareas convencional (es decir, no consciente de la asimetría de la arquitectura, o lo que es lo mismo, orientado a arquitecturas SMT), como por ejemplo el planificador de tareas convencional proporcionado por OmpSs, donde, en lugar de crear un *worker thread* por núcleo físico del sistema, se sigue una filosofía similar a la propuesta por el modelo CPUM, creando *un único worker thread por VC*. Internamente, cuando una tarea es elegida para ser ejecutada por un *worker thread* disponible, dicha tarea se ejecutará, de forma ya no secuencial, sino paralela sobre cada uno de los núcleos que componen el VC, explotando, en este caso, la asimetría interna de dicha abstracción. Por ejemplo, en el caso de ODROID, equipada con cuatro núcleos rápidos y cuatro lentos, la propuesta desarrollada desplegará únicamente cuatro *worker threads*; para cualquier tarea básica o kernel de álgebra lineal a ejecutar, dicha rutina explotará la asimetría internamente, siempre bajo la condición de que existe una implementación de la misma optimizada para este tipo de paralelismo asimétrico.

Siguiendo esta idea, la arquitectura expuesta al planificador de tareas es completamente *simétrica*, y los kernels en BLIS (o en cualquier otra biblioteca utilizada para la ejecución de las tareas) configuran una "caja negra" que abstrae del carácter asimétrico al planificador.

En resumen, si en una configuración convencional el núcleo es el recurso mínimo de computación para el planificador de tareas, y éstas son totalmente secuenciales, en la solución propuesta el VC es el recurso básico de computación de cara al planificador, mientras que la implementación de las tareas pasa a ser no sólo paralela, sino adaptada para la correcta explotación del paralelismo existente dentro de cada VC.

4.2.2. Comparación y ventajas frente a otras alternativas

La solución desarrollada reúne un conjunto de ventajas de cara al desarrollador:

- El planificador de tareas no es consciente de la asimetría, por lo que cualquier versión convencional del mismo podrá trabajar sobre este tipo de sistemas sin modificaciones específicas.
- Cualquier política de planificación (por ejemplo, *work stealing*, planificación consciente de la localidad de datos, implementación de caches por software, ...) ya desarrollada para arquitecturas simétricas, o cualquier futura mejora, tendrá también un impacto directo en el rendimiento sobre el sistema asimétrico.
- Cualquier mejora en la implementación de las bibliotecas asimétricas subyacentes (por ejemplo, BLIS) tendrá un impacto directo sobre el AMP. Esta observación se aplica a arquitecturas con distinto número de núcleos lentos/rápidos dentro del VC, ratios en las frecuencias de funcionamiento, o incluso ante la introducción de más niveles de asimetría (esto es, núcleos con rendimiento intermedio).

Obviamente, existe una dificultad adicional intrínseca a la solución propuesta, y que se convierte en un requisito fundamental de cara a su viabilidad: debe existir, necesariamente, una versión consciente de la asimetría para cada tarea a ejecutar (por ejemplo, una implementación completa de las rutinas BLAS). En el ámbito del álgebra lineal densa, este requisito se cumple a través de la versión asimétrica de BLIS, aunque en otros ámbitos, el desarrollo específico de este tipo de implementaciones es todavía escaso.

4.2.3. Requisitos a nivel de tarea en el ámbito del álgebra lineal

Por último, cabe destacar que son necesarios ciertos requisitos adicionales en la implementación multihebra de BLIS que opera sobre el modelo propuesto. Considérese el kernel

GEMM y la descripción de alto nivel proporcionada en el Listado 3.3. Como se ha descrito anteriormente, resulta necesario distribuir el espacio de iteraciones de alguno de los bucles entre los dos tipos de núcleos que conforman un VC; siguiendo las directivas de paralelización descritas en [47], se distinguen a continuación dos escenarios de reparto posibles:

- En arquitecturas donde cada VC está compuesto por igual número de núcleos de cada tipo (por ejemplo, ODROID) realizar un reparto de los bucles externos (1 ó 3) acorde a las características de cada tipo de procesador dentro de un VC resulta suficiente, debido a que ambos clusters no comparten cache L2.
- En arquitecturas donde cada VC está compuesto por distinto número de núcleos de cada tipo (por ejemplo, JUNO) realizar un reparto de los bucles externos (1 ó 3) entre clusters, y un reparto homogéneo de alguno de los bucles internos (4 ó 5) para realizar una división final del espacio de iteraciones entre aquellos núcleos replicados dentro de un mismo cluster.

4.3. Resultados experimentales

4.3.1. Estudio experimental del tamaño de bloque óptimo

En tiempo de ejecución, OmpSs descompone la rutina para la implementación de la factorización de Cholesky en una colección de tareas que operan en submatrices (bloques) con una granularidad que viene definida por el tamaño de bloque `b`, véase el código del Listado 3.1. Estas tareas típicamente se reducen a invocaciones a kernels elementales a nivel de BLAS (en nuestro caso, BLIS) o LAPACK, véase el código proporcionado en el Listado 3.2.

El primer paso en nuestra evaluación radica en proporcionar una estimación realista de la ganancia de rendimiento potencial proporcionada por la solución propuesta (en caso de haberla). Un factor crítico desde este punto de vista es el rango de tamaños de bloque que resultan óptimos para el runtime convencional de OmpSs ante una determinada operación (en adelante b^{opt}). En particular, la eficiencia de la solución propuesta vendrá determinada por el rendimiento obtenido por las implementaciones BLIS de cada una de las tareas que componen la operación, comparada con su implementación secuencial equivalente para dimensiones del problema `n` en el orden de dimensión *bopt*.

La Tabla 4.1 muestra los tamaños de bloque óptimos b^{opt} para la factorización de Cholesky y tamaños de problema crecientes, utilizando el planificador OmpSs convencional enlazado con una versión secuencial de BLIS y un número creciente de *worker threads* entre 1

Tabla 4.1: Tamaños de bloque óptimos para la factorización de Cholesky utilizando el planificador convencional de OmpSs y una implementación secuencial de BLIS sobre ODROID.

	Tamaño del problema (n)														
	512	1,024	1,536	2,048	2,560	3,072	3,584	4,096	4,608	5,120	5,632	6,144	6,656	7,168	7,680
1 WT	192	384	320	448	448	448	384	320	320	448	448	448	448	384	448
2 WT	192	192	320	192	448	448	384	320	320	448	448	448	448	384	448
3 WT	128	192	320	192	384	448	320	320	320	448	448	448	448	384	448
4 WT	128	128	192	192	192	320	320	320	320	448	320	448	448	384	448

y 4. Obsérvese como, excepto para los menores tamaños de problema, los tamaños de bloque óptimos están en el rango entre 192 y 448. Estos tamaños de bloque resultan óptimos al ofrecer un equilibrio entre el paralelismo a nivel de tareas potencial y la eficiencia interna de las ejecuciones secuenciales de cada tarea individual.

La principal conclusión extraída tras el análisis de los resultados es la siguiente: para elevar el rendimiento de un planificador de tareas combinado con una versión asimétrica de BLIS, cada uno de los kernels que componen la operación implementados en dicha versión asimétrica debe exhibir mayor rendimiento que las respectivas implementaciones secuenciales, para dimensiones de matrices que estén en el orden de los tamaños de bloque mostrados en la Tabla 4.1.

La Figura 4.2 muestra el rendimiento alcanzado para las tres rutinas básicas BLAS que componen la factorización de Cholesky (GEMM, SYRK y TRSM) para el rango de dimensiones de interés sobre la plataforma ODROID y que operan sobre los distintos bloques de la matriz. El experimento compara la versión secuencial de BLIS sobre un único núcleo Cortex-A15 con la versión asimétrica de BLIS que combina un Cortex-A15 y un Cortex-A7. Es decir, compara la ejecución de las tareas que componen la factorización sobre un único núcleo físico (rápido) y sobre un VC (combinando un núcleo rápido y otro lento).

En general, las tres rutinas BLAS muestran una tendencia similar: los kernels de la versión secuencial de BLIS consiguen mejor rendimiento que sus respectivas implementaciones asimétricas para tamaños de problema pequeños (hasta aproximadamente $\mathtt{m}$, $\mathtt{n}$, $\mathtt{k} = 128$); sin embargo, a partir de dicha dimensión, el uso de un núcleo lento comienza a hacer que el rendimiento mejore. El aspecto más importante radica en el hecho de que el punto de corte entre ambas curvas está en el rango (e incluso es típicamente menor) de b^{opt}, véase Tabla 4.1. Esto implica que la versión asimétrica de BLIS puede, potencialmente, mejorar el rendimiento general de la factorización, incluso usando un planificador de tareas convencional. Además, la mejora en rendimiento aumenta con el tamaño de problema, estabilizándose para dimensiones alrededor de $\mathtt{m}$, $\mathtt{n}$, $\mathtt{k} \approx 400$. Dado que este valor está en el rango del tama-

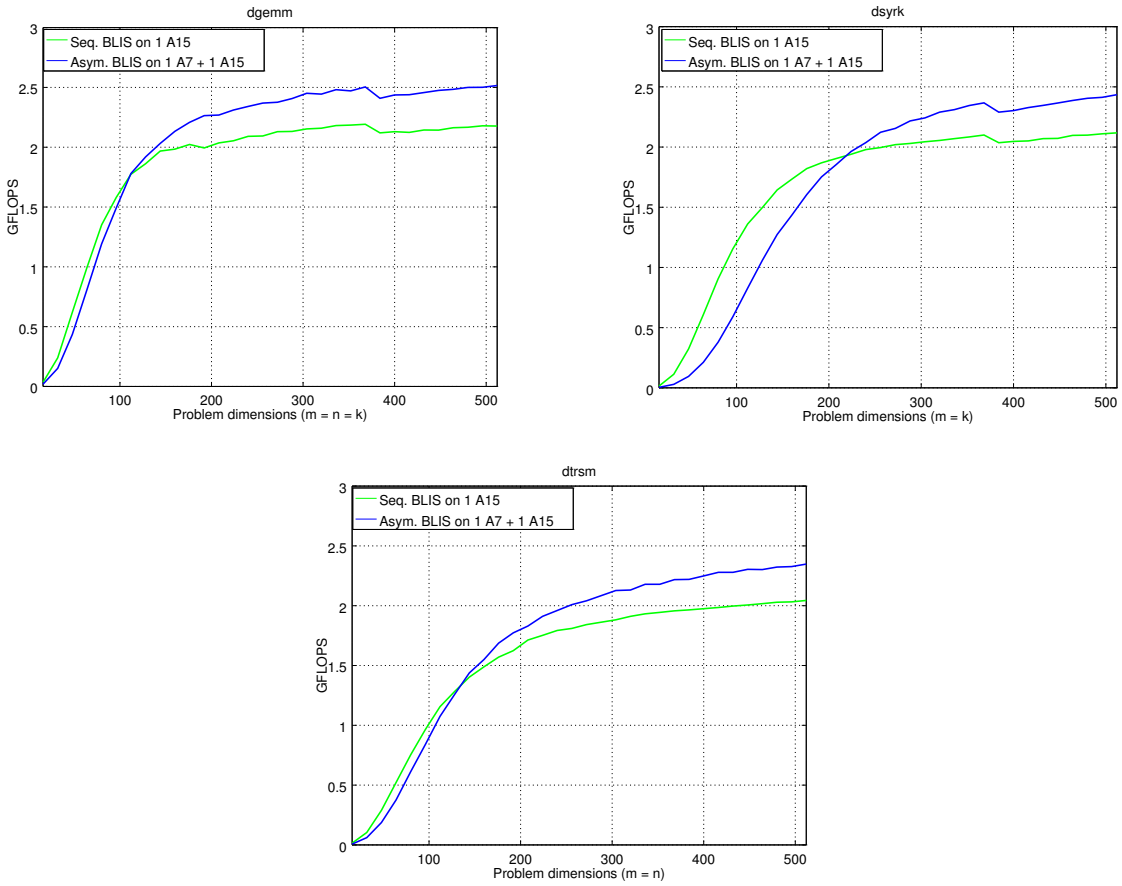


Figura 4.2: Rendimiento de los kernels BLAS-3 en las implementaciones secuencial y asimétrica de BLIS, utilizando, respectivamente un núcleo Cortex-A15 y un núcleo Cortex-A15 más un núcleo Cortex-A7 sobre ODROID.

ño de bloque óptimo para la factorización de Cholesky, sobre esta arquitectura es esperable una mejora en el orden de 0.3–0.5 GFLOPS por núcleo lento añadido.

4.3.2. Integración de BLIS asimétrico con un planificador convencional

Con el fin de analizar los beneficios reales de la solución propuesta en términos de rendimiento, comparamos a continuación el rendimiento del planificador OmpSs enlazado con *(a)* una versión secuencial de BLIS, y *(b)* con la versión asimétrica de BLIS. En todos los experimentos, los kernels BLIS en la primera configuración utilizarán exclusivamente un núcleo Cortex-A15, mientras que en el segundo caso se utilizara un núcleo Cortex-A15 más un núcleo Cortex-A7 para la ejecución de las tareas. La Figura 4.3 muestra los resultados obte-

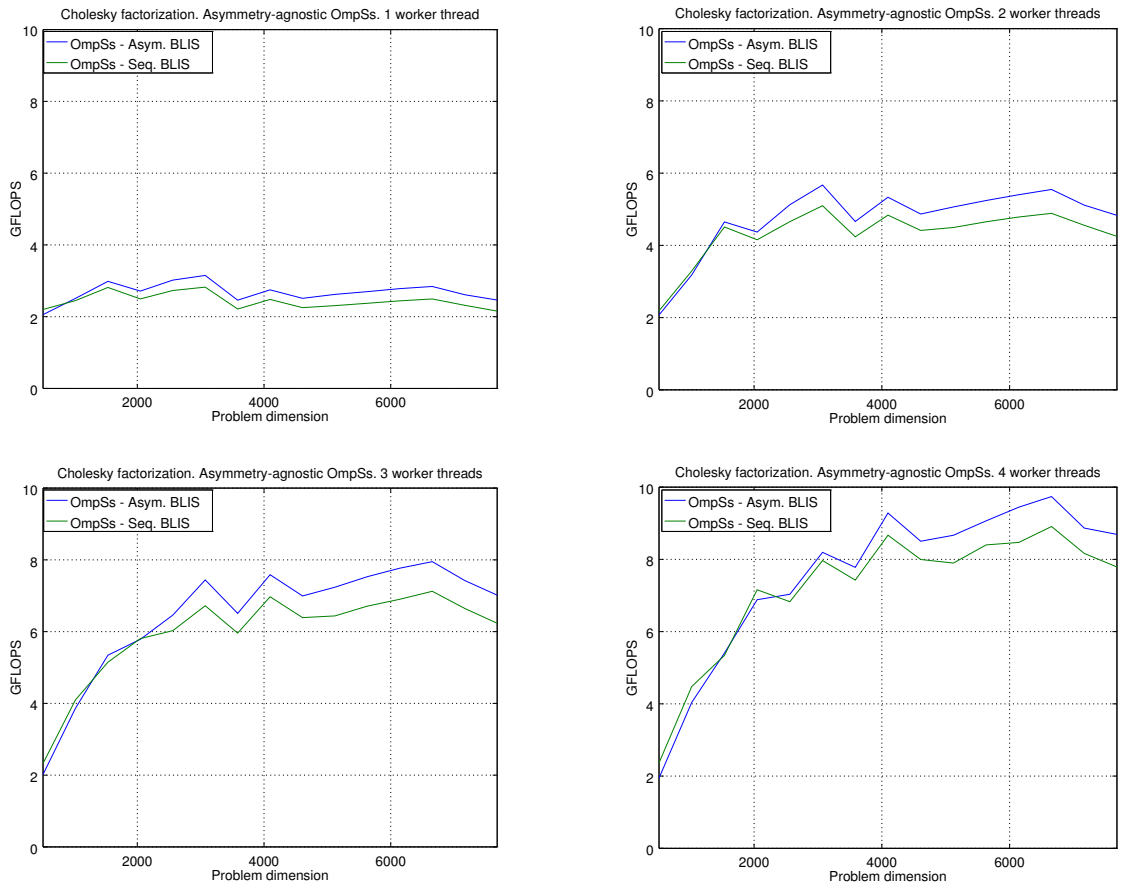


Figura 4.3: Rendimiento de la factorización de Cholesky utilizando el planificador convencional de OmpSs, enlazado con la versión secuencial y asimétrica de BLIS.

nidos para ambas configuraciones, utilizando un número creciente de *worker threads*, entre 1 y 4. Por simplicidad, únicamente se reportan los resultados obtenidos considerando el tamaño de bloque óptimo para cada tamaño de problema. En todos los casos, la solución basada en la utilización de una biblioteca asimétrica mejora el rendimiento de la implementación secuencial para matrices relativamente grandes (típicamente con dimensiones $\texttt{n} > 2{,}048$) mientras que, para tamaños de problema menores, el ratio de GFLOPS obtenido en ambos casos es similar. La razón de este comportamiento viene marcada por el tamaño de bloque óptimo reportado en la Tabla 4.1 y el rendimiento de BLIS mostrado en la Figura 4.2: para dicho rango de dimensiones de problema, el tamaño óptimo de bloque es significativamente menor, y ambas implementaciones BLIS obtienen resultados de rendimiento similares.

La diferencia cuantitativa en términos de rendimiento entre ambos enfoques se muestra en las Tablas 4.2 y 4.3. La primera tabla muestra la diferencia absoluta en rendimiento, mientras que la segunda muestra la diferencia en rendimiento por cada núcleo lento (Cortex-

Tabla 4.2: Mejora de rendimiento absoluta (en GFLOPS) para la factorización de Cholesky utilizando el planificador OmpSs convencional enlazado con una versión asimétrica de BLIS, con respecto al mismo planificador enlazado con la versión secuencial de BLIS sobre Odroid.

	Dimensión del problema (n)											
	512	1,024	2,048	2,560	3,072	4,096	4,608	5,120	5,632	6,144	6,656	7,680
1 wt	-0.143	0.061	0.218	0.289	0.326	0.267	0.259	0.313	0.324	0.340	0.348	0.300
2 wt	-0.116	-0.109	0.213	0.469	0.573	0.495	0.454	0.568	0.588	0.617	0.660	0.582
3 wt	-0.308	-0.233	-0.020	0.432	0.720	0.614	0.603	0.800	0.820	0.866	0.825	0.780
4 wt	-0.421	-0.440	-0.274	0.204	0.227	0.614	0.506	0.769	0.666	0.975	0.829	0.902

Tabla 4.3: Mejora de rendimiento por núcleo lento (en GFLOPS) para la factorización de Cholesky utilizando el planificador OmpSs convencional enlazado con una versión asimétrica de BLIS, con respecto al mismo planificador enlazado con la versión secuencial de BLIS sobre Odroid.

	Dimensión del problema (n)											
	512	1,024	2,048	2,560	3,072	4,096	4,608	5,120	5,632	6,144	6,656	7,680
1 wt	-0.143	0.061	0.218	0.289	0.326	0.267	0.259	0.313	0.324	0.340	0.348	0.300
2 wt	-0.058	-0.054	0.106	0.234	0.286	0.247	0.227	0.284	0.294	0.308	0.330	0.291
3 wt	-0.102	-0.077	-0.006	0.144	0.240	0.204	0.201	0.266	0.273	0.288	0.275	0.261
4 wt	-0.105	-0.110	-0.068	0.051	0.056	0.153	0.126	0.192	0.166	0.243	0.207	0.225

A7) introducido en el experimento. Consideremos, por ejemplo, el tamaño de problema $n = 6{,}144$. En este caso, el rendimiento mejora en 0.975 GFLOPS cuando se añaden los 4 núcleos lentos para dar soporte a los 4 núcleos Cortex-A15. Este hecho se traduce en una ganancia de rendimiento de 0.243 GFLOPS por core lento, ligeramente por debajo de la mejora que podría esperarse a partir de los resultados experimentales observados en la anterior sección. Nótese, sin embargo, como el rendimiento por Cortex-A7 añadido se reduce desde 0.340 GFLOPS al incorporar únicamente un núcleo, hasta 0.243 GFLOPS, al utilizar los cuatro núcleos lentos, por lo que el rendimiento por core lento añadido decrece al aumentar el número de *worker threads*. Esperamos averiguar las causas en un futuro.

4.3.3. Rendimiento frente a un planificador consciente de la asimetría

El último conjunto de experimentos tiene como objetivo ofrecer una visión global de las ventajas en términos de rendimiento de distintas configuraciones en la ejecución paralela a nivel de tareas de la factorización de Cholesky utilizando el planificador de OmpSs. Más concretamente, se consideran a continuación las siguientes alternativas:

1. El planificador convencional OmpSs enlazado con una implementación secuencial de BLIS ("OmpSs - Seq. BLIS").

2. El planificador convencional OmpSs enlazado con la versión asimétrica de BLIS que considera el SoC como cuatro *núcleos virtuales (VC)* ("OmpSs - Asym. BLIS").

3. La versión consciente de la asimetría (*criticality-aware, o botlev*) de OmpSs, enlazada con la versión secuencial de BLIS ("Botlev-OmpS - Seq. BLIS").

En las ejecuciones, se utilizan los cuatro núcleos Cortex-A15 y se evalúa el impacto de añadir núcleos Cortex-A7 a la configuración (entre 1 y 4) para el planificador Botlev.

La Figura 4.4 muestra el rendimiento obtenido por cada una de las configuraciones anteriormente descritas sobre la plataforma Odroid. Los resultados pueden ser analizados si se dividen en tres grandes grupos, atendiendo a la dimensión del problema:

- Para matrices pequeñas (`n` = 512, 1,024), el planificador convencional utilizando exclusivamente cuatro núcleos rápidos (esto es, enlazado con la versión secuencial de BLIS para la ejecución de tareas) obtiene los mejores resultados en términos de rendimiento. Esta es una observación esperada y ya fue observada en la Figura 4.3; la principal causa radica en el pequeño tamaño óptimo de bloque para este rango de dimensiones de problema, necesario para exponer el suficiente paralelismo a nivel de tareas. Esta necesidad invalida el uso de la implementación asimétrica de BLIS dado su bajo rendimiento sobre matrices de muy reducidas dimensiones; véase Figura 4.2. Además, cabe destacar que el planificador OmpSs adaptado a la asimetría (Botlev) no obtiene ratios de rendimiento elevados para este rango de dimensiones, independientemente del número de núcleos Cortex-A7 introducidos en el experimento.

- Para matrices de tamaño intermedio (`n` = 2,048), la diferencia de rendimiento entre los diferentes enfoques se reduce. Para este rango de dimensiones, Botlev-OmpSs es competitivo, y también obtiene mejores rendimientos que la configuración convencional.

- Para matrices de grandes dimensiones (`n` = 4,096, 6,144, 7,680) la anterior tendencia se consolida, y ambos enfoques conscientes de la arquitectura muestran ganancias de rendimiento considerables. Comparando ambos enfoques conscientes de la arquitectura, nuestra solución obtiene mejor rendimiento, incluso considerando la utilización de todos los núcleos lentos para el planificador Botlev-OmpSs.

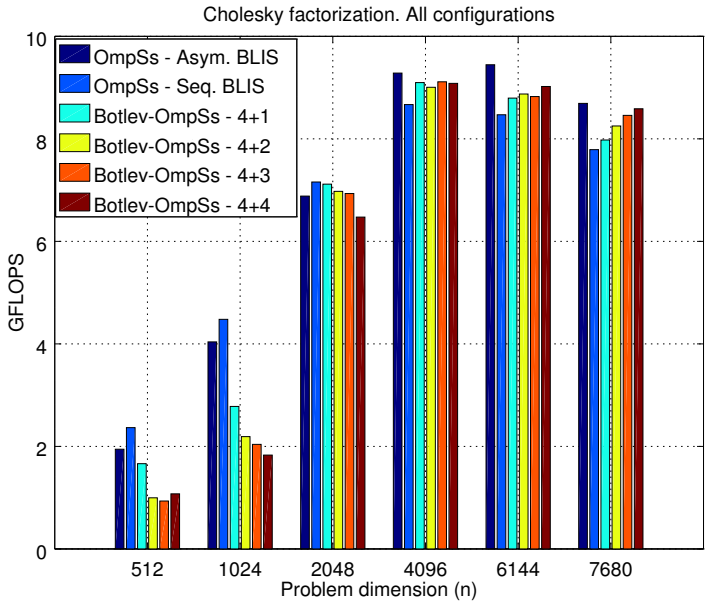


Figura 4.4: Rendimiento (en GFLOPS) para la factorización de Cholesky utilizando el planificador convencional de OmpSs enlazado con la versión secuencial de BLIS, la implementación asimétrica de BLIS, y la implementación consciente de la arquitectura Botlev de OmpSs enlazado con la versión secuencial de BLIS sobre ODROID. Las etiquetas con la forma "4+x" indican una ejecución con 4 núcleos Cortex-A15 utilizando x núcleos Cortex-A7.

En conclusión, nuestro enfoque para explotar la asimetría mejora tanto la portabilidad como la facilidad de programación y desarrollo de planificadores de tareas, evitando el desarrollo de políticas específicas de planificación conscientes de la asimetría. Además, el rendimiento obtenido es comparable con éstos para cualquier tamaño de problema; para matrices de tamaño medio/grande, mejora además de forma sustancial el rendimiento conseguido por un planificador no consciente de la asimetría.

4.3.4. Análisis detallado de rendimiento

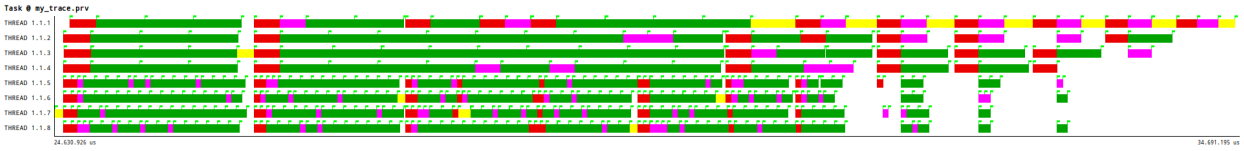
A continuación, se proporcionan más detalles sobre el comportamiento, en términos de rendimiento, de cada una de las configuraciones anteriormente descritas. Las trazas de ejecución mostradas en esta sección han sido extraídas con la herramienta de instrumentación `Extrae` [17] y visualizadas a través de la herramienta `Paraver` [41]. Los resultados corresponden a un caso concreto de la factorización de Cholesky, con matriz de entrada de dimensión `n` = 6,144 y tamaño de bloque `b` = 448.

Visión general de la ejecución de tareas

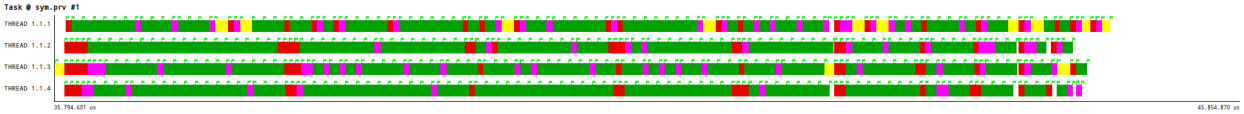
La Figura 4.5 muestra una traza completa de ejecución para cada configuración del planificador OmpSs. A grandes rasgos, es posible extraer un conjunto de observaciones generales:

- Desde el punto de vista del tiempo total de ejecución, el planificador convencional de OmpSs combinado con una versión asimétrica de BLIS obtiene los mejores resultados, seguido por la implementación Botlev del planificador. Es necesario destacar que la versión no consciente de la asimetría de OmpSs, con 8 *worker thread* y sin ninguna modificación adicional, obtiene los peores resultados con diferencia. En este caso, el desequilibrio de carga y los largos periodos de inactividad en la ejecución de tareas, especialmente a medida que el paralelismo de tareas disponible disminuye (en las últimas etapas de la ejecución) conllevan una penalización de rendimiento muy considerable.

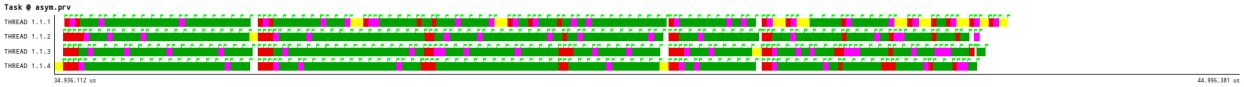
- Las marcas de inicio/finalización de cada tarea revelan que la versión asimétrica de BLIS (que utiliza los recursos combinados en cada VC) requiere menor tiempo de ejecución por tarea que las dos alternativas basadas en BLIS secuencial. Un efecto a observar es la lógica diferencia en rendimiento en la versión Botlev del planificador en la ejecución de las tareas sobre núcleos rápidos (*worker threads* entre el 5 y el 8) y lentos (*worker threads* entre el 1 y el 4). Este fenómeno no se observa en nuestra



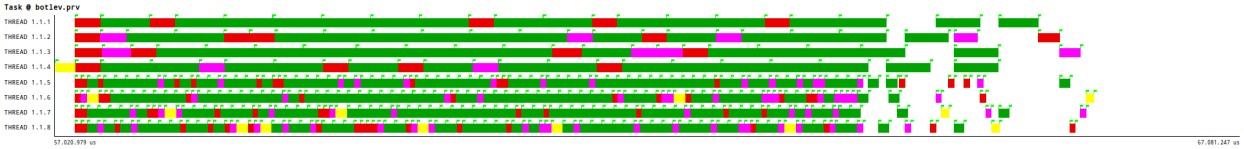
(a) OmpSs - BLIS secuencial (8 worker threads)



(b) OmpSs - BLIS secuencial (4 worker threads)



(c) OmpSs - BLIS asimétrico (4 worker threads)



(d) Botlev-OmpSs - BLIS secuencial (8 worker threads, 4+4)

Figura 4.5: Trazas de ejecución para las tres configuraciones estudiadas en la factorización de Cholesky (`n` = 6,144, `b` = 448). La línea de tiempo en cada fila recoge las tareas ejecutadas por un único *worker thread*. Las tareas se han coloreado siguiendo la convención de la Figura 3.1; las fases coloreadas en color blanco representan etapas sin actividad. Las marcas en color verde denotan puntos de inicialización de la ejecución de cada tarea.

solución, ya que, para el planificador, cualquiera de los 4 *worker threads* disponibles observa unidades de procesamiento (VCs) homogéneos.

- El planificador Botlev-OmpSs incluye políticas complejas de planificación que incluyen el tratamiento de prioridades, avanzando la ejecución de tareas en el camino crítico y, cuando sea posible, asignando éstas a núcleos rápidos (véanse, por ejemplo, las tareas correspondientes a factorizaciones de los bloques diagonales, coloreadas en amarillo). Esta política induce una planificación más compacta durante las primeras fases de la ejecución paralela, pero con mayores problemas a medida que el grado de concurrencia disminuye (en las últimas iteraciones de la factorización). Aunque es posible, no se ha activado este tipo de gestión de prioridades en la versión convencional de OmpSs utilizada en el resto de experimentos, y su interacción con una implementación asimétrica de las tareas se plantea como trabajo futuro.

Se proporciona a continuación un análisis cuantitativo de la duración temporal de las tareas y un estudio más detallado de la estrategia de planificación integrada en cada confi-

guración.

Duración de las tareas

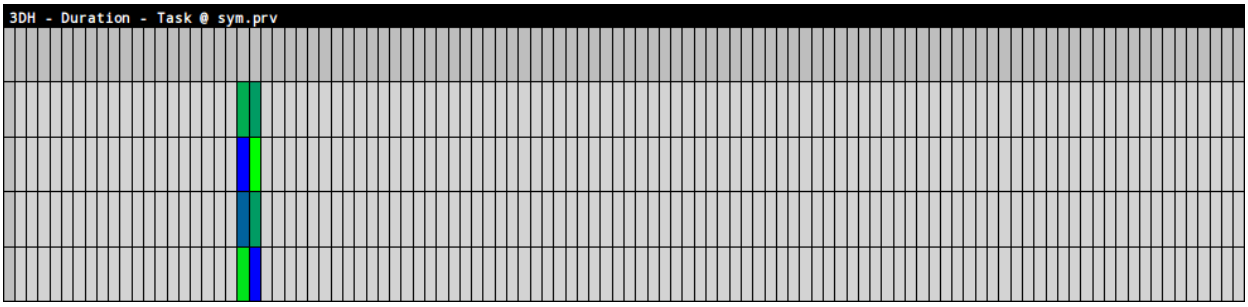
La Tabla 4.4 muestra el tiempo medio de ejecución por tipo de tarea para cada *worker thread*. Los resultados muestran como el tiempo de ejecución para cada tipo de tarea es considerablemente menor en la versión asimétrica de BLIS que en las alternativas basadas en ejecuciones secuenciales de las tareas. La única excepción es la factorización de los bloques diagonales (rutina `dpotrf`), ya que se trata de una rutina perteneciente a LAPACK, y por tanto no disponible en BLIS y no paralelizada de forma consciente de la asimetría). Inspeccionando la duración de las tareas en la configuración Botlev-OmpSs, se observa una notable diferencia en función del tipo de núcleo sobre el que el planificador mapea las tareas. Por ejemplo, el tiempo medio de ejecución para `dgemm` varía entre más de 400 ms en un núcleo lento, hasta prácticamente 90 ms en un núcleo rápido. Este comportamiento se reproduce para todos los tipos de tareas.

Para ilustrar más claramente estas observaciones, la Figura 4.6 representa un histograma detallado de los tiempos de ejecución para las distintas instancias de la tarea `dgemm` durante la ejecución paralela. Cada casilla corresponde al número de tareas con un tiempo de ejecución dado, mientras que las filas corresponden a cada *worker thread* en ejecución. Comparando las configuraciones que utilizan planificadores convencionales (trazas (a) y (b)), existe una clara desviación en el tiempo medio de ejecución hacia mayores rendimientos al utilizar BLIS asimétrico, esto es, el tiempo medio de ejecución de una tarea es claramente menor en este caso. El histograma para Botlev (traza (c)) muestra dos zonas diferenciadas, que corresponden a tareas ejecutadas sobre núcleos rápidos y núcleos lentos, respectivamente. Nótese que las tareas ejecutadas sobre núcleos rápidos obtienen el mismo rendimiento que las equivalentes en una configuración convencional usando BLIS secuencial. Se ha observado un comportamiento similar para el resto de tareas BLAS ejecutadas durante el experimento.

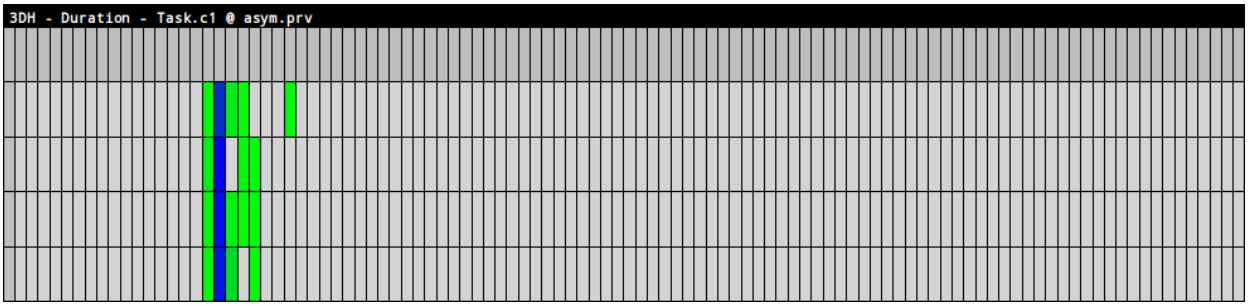
Políticas de planificación y tiempos sin actividad

La Figura 4.7 ilustra el orden de ejecución de tareas determinado por el planificador de OmpSs. En ella, las tareas se muestran utilizando un gradiente de color, atendiendo exclusivamente al orden en el que son encontradas en el código secuencial (Figura 3.1), desde la primera hasta la última.

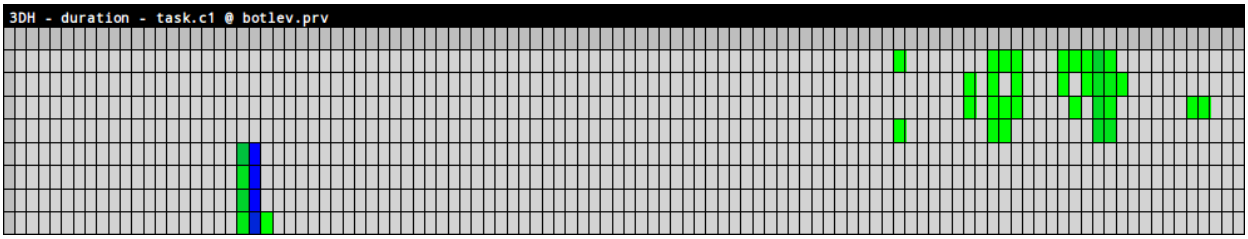
En tiempo de ejecución, el planificador de tareas Botlev-OmpSs lanza tareas a ejecución fuera de orden, dependiendo de su criticalidad, y las asigna, a ser posible, sobre núcleos



(a) OmpSs + BLIS simétrico.



(b) OmpSs + BLIS asimétrico.



(c) Botlev-OmpSs - 4+4 hebras.

Figura 4.6: Histograma de duración de tareas para la tarea `dgemm` sobre las tres configuraciones de planificador utilizadas para la factorización de Cholesky (`n` = 6,144, `b` = 448). Las filas corresponden a los *worker threads* en ejecución. Las columnas corresponden a intervalos de tiempo de ejecución medio. Los colores (en gradiente) indican el número de tareas en el intervalo correspondiente (desde verde claro hasta azul oscuro, en un rango de menor a mayor número de tareas).

Tabla 4.4: Tiempo medio (en ms) por tarea y *worker thread* en la factorización de Cholesky (`n` = 6,144, `b` = 448) para las tres configuraciones de runtime.

	OmpSs - Seq. BLIS (4 worker threads)				OmpSs - Asym. BLIS (4 worker threads)				Botlev-OmpSs - Seq. BLIS (8 worker threads, 4+4)			
	`dgemm`	`dtrsm`	`dsyrk`	`dpotrf`	`dgemm`	`dtrsm`	`dsyrk`	`dpotrf`	`dgemm`	`dtrsm`	`dsyrk`	`dpotrf`
WT 0	89.62	48.12	47.14	101.77	79.82	42.77	44.42	105.77	406.25	216.70	–	–
WT 1	88.96	48.10	47.14	–	78.65	42.97	44.56	76.35	408.90	207.41	212.55	–
WT 2	89.02	48.36	47.18	87.22	79.14	43.14	44.60	85.98	415.31	230.07	212.56	–
WT 3	90.11	48.51	47.42	–	79.28	43.10	44.59	67.73	410.84	216.95	216.82	137.65
WT 4	–	–	–	–	–	–	–	–	90.97	48.97	48.36	–
WT 5	–	–	–	–	–	–	–	–	90.61	48.86	48.16	90.78
WT 6	–	–	–	–	–	–	–	–	91.28	49.43	47.97	89.58
WT 7	–	–	–	–	–	–	–	–	91.60	49.49	48.62	95.43
AVG.	89.43	48.27	47.22	94.49	79.22	42.99	44.54	83.96	250.72	133.49	119.29	103.36

rápidos. De acuerdo con esta estrategia, la ejecución fuera de orden se revela más frecuentemente en las líneas de tiempo para los núcleos rápidos que para los lentos. Con el planificador convencional, la ejecución fuera de orden sólo viene dictada por el cumplimiento de las dependencias de datos en tiempo de ejecución.

A la vista de las trazas de ejecución, es posible observar como el planificador Botlev-OmpSs muestra una penalización en el rendimiento muy apreciable debida a la existencia de periodos ociosos en la parte final de la factorización, cuando la concurrencia disponible disminuye. Este problema no se da utilizando políticas de planificación convencionales. Sin embargo, en las primeras fases de la factorización, el uso de una política consciente de la criticalidad de las tareas (implementada en Botlev-OmpSs), reduce de forma efectiva los períodos de tiempo en los que los *worker threads* permanecen ociosos.

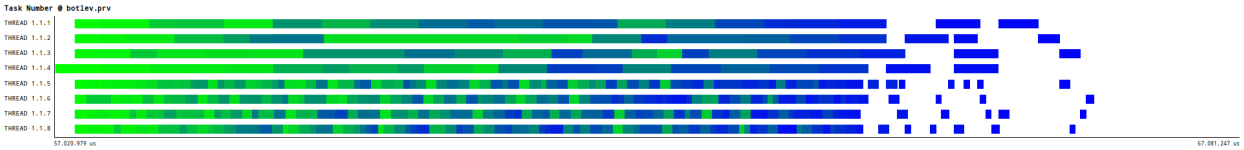
La Tabla 4.5 muestra el porcentaje de tiempo en el que cada *worker thread* permanece en estado `running` (ejecutando tareas) o `idle` (sin ejecutar tareas). En general, la cantidad de tiempo transcurrido en estado `idle` es mucho mayor para Botlev-OmpSs que para las implementaciones convencionales (17 % contra 5 %, respectivamente). Nótese también la notable diferencia en el porcentaje de tiempo ocioso entre núcleos rápidos y lentos (20 % y 13 %, respectivamente), lo que lleva a la conclusión de que los núcleos rápidos deben esperar a la finalización de tareas ejecutadas en núcleos lentos. En otras palabras, en estas configuraciones, los núcleos lentos "frenan" a los rápidos. Este hecho es también constatable en las etapas finales de la traza obtenida con la configuración Botlev-OmpSs.

Las anteriores observaciones sugieren la combinación de distintas configuraciones sobre una misma ejecución si se utilizan procesadores asimétricos; en este enfoque, la asimetría podría ser explotada a través de políticas de planificación conscientes de la arquitectura

(a) OmpSs - BLIS secuencial (8 worker threads)

(b) OmpSs - BLIS secuencial (4 worker threads)

(c) OmpSs - BLIS asimétrico (4 worker threads)



(d) Botlev-OmpSs - BLIS secuencial (8 worker threads, 4+4)

Figura 4.7: Orden de ejecución de tareas para las tres configuraciones estudiadas sobre la factorización de Cholesky ($\mathtt{n} = 6{,}144$, $\mathtt{b} = 448$). En las trazas, las tareas se ordenan según su orden de aparición en el código secuencial, y son mostradas usando un gradiente de color, correspondiendo el color verde claro a tareas tempranas, y azul oscuro a tareas tardías.

durante las primeras fases de la factorización –cuando el paralelismo de tareas disponible es elevado–, y reemplazar el enfoque por el uso de tareas conscientes de la arquitectura y VCs en las fases finales de la ejecución, cuando la concurrencia es escasa. Ambos enfoques no son mutuamente exclusivos, sino complementarios en función del nivel de concurrencia disponible en un punto determinado de la ejecución. Estas ideas se plantean como líneas de investigación futuras al presente trabajo.

Tabla 4.5: Porcentaje de tiempo por *worker thread* en estado `idle` o `running` para distintas configuraciones de planificador para la factorización de Cholesky (`n` = 6,144, `b` = 448). Nótese que WT 0 es el hilo principal, y por tanto nunca permanece en estado `idle` dadas las características de OmpSs; para él, el resto del tiempo hasta el 100 % se dedica a sincronización, planificación y creación de la hebra. Para el resto de las hebras, esta cantidad de tiempo se considera sobrecoste de planificación (`runtime overhead`).

	OmpSs - Seq. BLIS (4 worker threads)		OmpSs - Asym. BLIS (4 worker threads)		Botlev-OmpSs - Seq. BLIS (8 worker threads, 4+4)	
	`idle`	`running`	`idle`	`running`	`idle`	`running`
WT 0	–	98.41	–	97.85	–	86.53
WT 1	5.59	94.22	5.51	94.29	13.63	86.28
WT 2	3.14	96.67	5.27	94.53	13.94	85.98
WT 3	5.77	94.07	5.17	94.62	13.43	86.47
WT 4	–	–	–	–	19.26	80.51
WT 5	–	–	–	–	21.12	78.69
WT 6	–	–	–	–	20.84	78.97
WT 7	–	–	–	–	20.09	79.70
AVG.	4.84	95.89	5.32	94.90	17.47	82.89

Capítulo 5

Optimización de la eficiencia energética de OmpSs sobre arquitecturas asimétricas

5.1. Descripción de la estrategia de optimización

Las arquitecturas asimétricas permiten optimizar la eficiencia energética asignando las tareas a núcleos de distintas características en función de los requisitos de rendimiento y consumo de cada tarea. Pero además, la arquitectura permite ajustar el rendimiento y consumo de cada núcleo utilizando las técnicas de escalado de frecuencia que se encuentran disponibles en los procesadores modernos.

Los planificadores de tareas utilizados en la actualidad no realizan este tipo de optimizaciones energéticas. Por eso, en este capítulo se presentan las modificaciones realizadas al planificador consciente de la asimetría Botlev, ya presentado en capítulos anteriores, para incluir en el mismo diversas técnicas de mejora de eficiencia energética.

5.1.1. DVFS sobre la arquitectura big.LITTLE

Las siglas DVFS (*Dynamic voltage frequency scaling*) representan a un conjunto de técnicas caracterizadas por variar la frecuencia o el voltaje del procesador de manera dinámica en tiempo de ejecución con el objetivo de obtener una mejora energética o disminuir la potencia instantánea consumida. En la actualidad, esta técnica ha cobrado especial relevancia en los dispositivos móviles, donde la duración de la batería es un factor muy importante a tener en cuenta. Sin embargo, esta técnica no se limita solamente a este tipo de dispositivos,

Cluster	Frecuencias soportadas
Cortex-A7	{800MHz, 900MHz, 1GHz, 1.1GHz, 1.2GHz, 1.3GHz}
Cortex-A15	{800MHz, 900MHz, 1GHz, 1.1GHz, 1.2GHz, 1.3GHz}
Cortex-A53	{450MHz, 575MHz, 700MHz, 775MHz, 850MHz}
Cortex-A57	{450MHz, 625MHz, 800MHz, 950MHZ, 1100MHz}

Tabla 5.1: Conjunto de frecuencias válidas para los clusters utilizados.

sino a cualquier tipo de procesador que lo soporte. Como ejemplo del uso de esta técnica, destacar la implementación comercial de Intel para sus CPUs denominada *Intel SpeedStep*, o las de AMD denominadas *AMD Cool'n'Quiet* para procesadores destinados a servidores y equipos de sobremesa, y *AMD PowerNow!* para procesadores orientados a dispositivos móviles. Además, destacar que este tipo de técnicas también se pueden aplicar a procesadores de propósito específico, como puede ser la implementación de AMD denominada *AMD PowerTune* para GPUs y APUs (*Accelerated Processing Unit*).

Los procesadores de la familia big.LITTLE de ARM poseen soporte para realizar escalado de frecuencia, aunque únicamente se permite realizar el escalado a nivel de cluster y no de núcleo, lo que implica que todos los núcleos del cluster siempre van a estar ejecutando tareas a la misma frecuencia. Además, las frecuencias a las que puede operar el cluster se encuentran limitadas a un conjunto cerrado impuesto por el diseño interno. Para las dos plataformas sobre las que se han desarrollado los experimentos, la tabla 5.1 muestra el conjunto de frecuencias en las que cada cluster puede trabajar.

El acceso a esta característica de los procesadores en el kernel Linux se realiza a través del subsistema *cpufreq*, el cual proporciona la librería `libcpufreq` y los ejecutables `cpufreq-set` y `cpufreq-info` que permiten obtener y modificar la frecuencia de cada núcleo en tiempo de ejecución.

5.1.2. Evaluación de rendimiento/eficiencia energética de las tareas

En un sistema asimétrico, se pueden identificar tres variables que afectan al rendimiento y al consumo energético de una aplicación:

- El tipo de núcleo usado para ejecutar las distintas tareas que componen el problema. Ejecutar una tarea sobre un núcleo del cluster big va a generar un mayor rendimiento que si se ejecuta sobre un núcleo del cluster LITTLE.

- El número de núcleos activos durante la ejecución del problema. Un número bajo de núcleos activos puede suponer una pérdida de rendimiento global considerable, sin embargo también supone un ahorro en el consumo energético del procesador.

- La frecuencia a la que trabajan los distintos clusters. Frecuencias bajas pueden afectar al rendimiento de la aplicación al aumentar el tiempo de ejecución, pero pueden suponer un descenso de la potencia instantánea.

A partir de estas tres dimensiones, existen muchas formas de combinar los distintos factores para conseguir adaptar el rendimiento y el consumo energético a la aplicación a ejecutar. En las siguientes secciones se exploran las técnicas principales para obtener la mejora en rendimiento y energía, adaptadas para este tipo de plataformas.

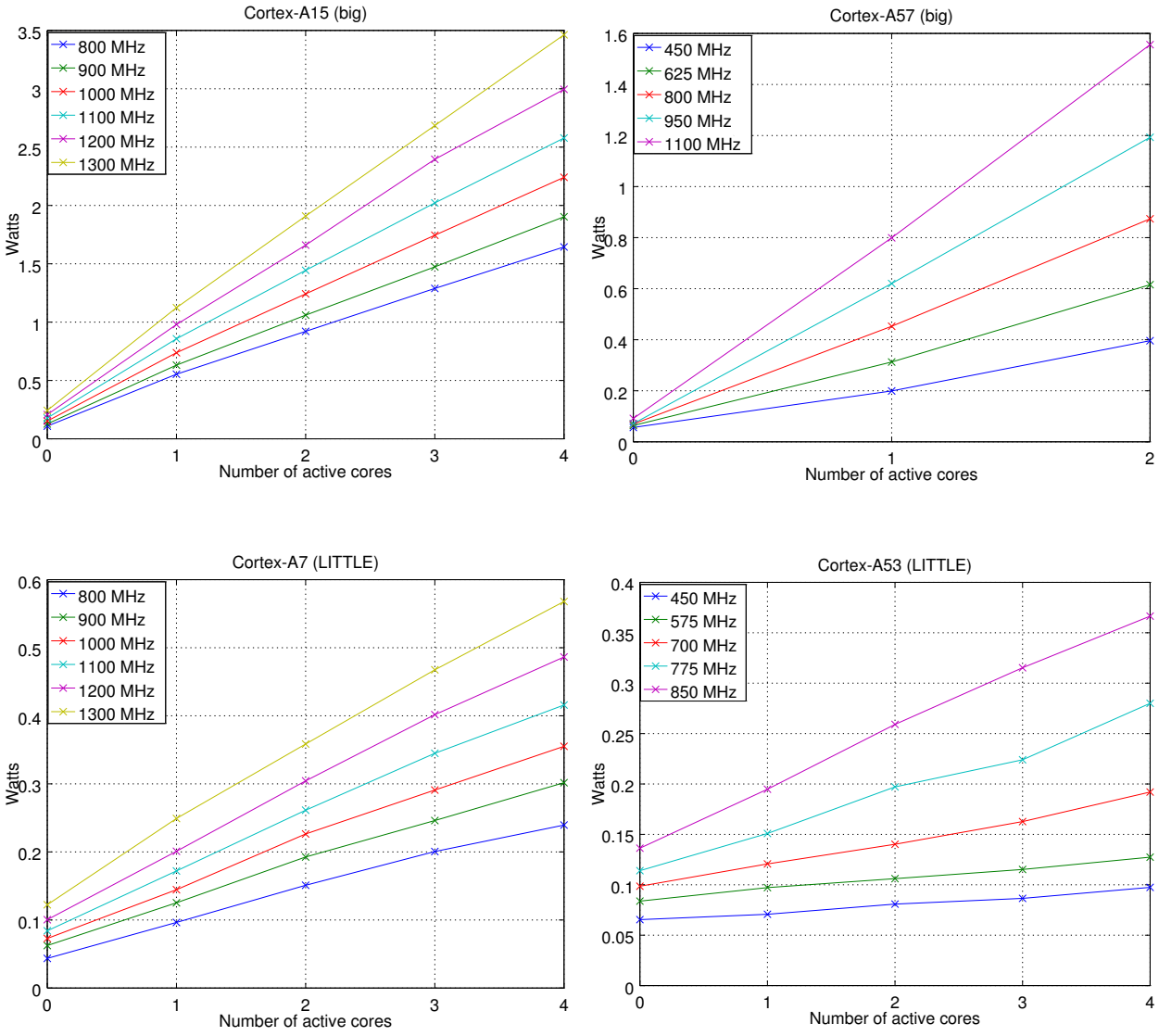
La Figura 5.1 relaciona dos de los puntos anteriores, mostrando la potencia instantánea consumida por cada cluster que compone la arquitectura en función del número de núcleos activos y la frecuencia de ejecución de los mismos. Aunque el experimento se ha realizado ejecutando varias llamadas en paralelo para realizar una multiplicación de matrices `gemm` (una por cada núcleo activo), se ha comprobado experimentalmente que todas las tareas que forman parte de la factorización de Cholesky utilizan el 100 % de la capacidad del núcleo correspondiente, independientemente de a qué cluster o plataforma pertenezca, haciendo que los resultados mostrados sean válidos para cualquier tarea de las que componen la factorización.

5.2. Políticas de reducción de consumo

En esta sección se detallan las diversas políticas desarrolladas para reducir el consumo energético en una arquitectura asimétrica. Las políticas se encuentran divididas en dos grandes grupos: un primer grupo formado por políticas basadas en *escalado de frecuencia*, y un segundo grupo formado por políticas encargadas de modificar la *asignación de las tareas* a los diferentes *worker threads* generados por el *runtime* (y por tanto, a los diferentes núcleos que componen la arquitectura).

5.2.1. Políticas basadas en escalado de frecuencia

A la hora de aplicar técnicas de escalado de frecuencia (DVFS) sobre un problema, la técnica presenta de manera simplificada dos grandes dimensiones en las que tomar decisiones sobre los parámetros de configuración para conseguir el objetivo de reducir el consumo total:



(a) Odroid (b) Juno

Figura 5.1: Medidas de consumo energético por núcleo y plataforma

(a) una primera dimensión que determina *qué frecuencias utilizar* durante los diferentes cambios, y *(b)* una segunda dimensión que determina *en qué momentos de la ejecución se debe modificar la frecuencia.*

La primera dimensión normalmente viene determinada por la propia arquitectura sobre la que se ejecuta el problema, ya que es común que el procesador o el sistema operativo no puedan seleccionar cualquier frecuencia de funcionamiento, sino solamente una serie de frecuencias predeterminadas. Esto hace que tomar la decisión de a qué frecuencia se desea que el procesador trabaje se reduzca a una elección sobre un conjunto cerrado y finito de frecuencias. Cabe destacar que, en ocasiones, puede resultar conveniente descartar algunas de estas frecuencias y no tenerlas en consideración, ya sea porque impactan de manera muy negativa en el rendimiento, o no tienen un impacto significativo en la mejora de consumo.

La segunda dimensión está estrechamente relacionada con el problema a ejecutar y el conocimiento que se tenga de él. Las decisiones que se pueden tomar varían desde decisiones de grano grueso, como por ejemplo considerar el nivel de carga de trabajo de cada procesador y variar la frecuencia en función de dicho parámetro, hasta decisiones de grano más fino, por ejemplo conocer el comportamiento de cada una de las tareas a ejecutar y el árbol de dependencias de antemano, y así tomar decisiones en función de estos parámetros. Por ejemplo, si una tarea es crítica, es posible aumentar la frecuencia del núcleo asociado para así liberar nuevas tareas cuanto antes; o si se sabe que la ejecución va a estar bloqueada hasta que finalice una tarea en concreto, es posible disminuir la frecuencia del resto de núcleos hasta que esta tarea finalice, y así disminuir el consumo energético. Hay que destacar que aunque disminuir la frecuencia del procesador implique conseguir una potencia instantánea menor, esto no siempre implica que la energía final consumida sea también menor: al bajar la frecuencia, el tiempo empleado para finalizar la ejecución de una tarea puede aumentar lo suficiente para provocar un consumo de energía global mayor. Por tanto, cualquier decisión que afecte a la variación de frecuencia se debe considerar como un compromiso entre rendimiento y consumo energético; la clave aquí radica en optimizar dicho compromiso.

Adicionalmente a estas dos dimensiones, los sistemas heterogéneos en general, y las arquitecturas asimétricas en particular, presentan una dimensión extra sobre la que tomar decisiones: *(c)* decidir *sobre qué elementos de cálculo* aplicar el escalado de frecuencia. En el caso de las arquitecturas asimétricas, esta decisión se reduce a decidir si aplicar el escalado de frecuencias al cluster de núcleos big, o al cluster de núcleos LITTLE, dadas las limitaciones en la selección de frecuencias mencionadas en la Sección 5.1.1.
Las siguientes políticas desarrolladas contemplan estos aspectos, adaptándolos para la ejecución sobre una arquitectura big.LITTLE y un modelo de paralelismo basado en tareas.

Política P1: Tareas limitadas por el camino crítico

Durante la ejecución de un programa paralelo mediante un paradigma basado en tareas, es común que, en cierto punto de la ejecución paralela, la mayor parte de tareas listas para ser ejecutadas sean tareas pertenecientes al *camino crítico*, provocando que nuevas tareas no puedan pasar a considerarse listas para ejecución hasta que las tareas críticas finalicen.

Este comportamiento puede observarse en una aplicación cuyo grafo de dependencias se encoja muy rápidamente en algún punto intermedio de la ejecución, para luego volverse a expandir rápidamente (un árbol con forma de diábolo). En este tipo de árbol, las tareas "centrales" serán críticas, ya que son prioritarias para que la ejecución paralela prosiga, y además, en el momento que esto ocurra, un gran número de tareas serán liberadas para ser ejecutadas y poder continuar la ejecución del problema.

Este problema, aplicado al planificador BOTLEV, (descrito en la sección 3.1.4) supondría que en el momento en el que el árbol se estreche, la cola de tareas no críticas tendría un tamaño muy inferior a la de tareas críticas, y por tanto los núcleos LITTLE tendrían mucha menor carga de trabajo que los núcleos big. Este fenómeno se da mientras los núcleos big finalizan la ejecución de sus tareas críticas y se liberan más tareas para ejecutar en los núcleos LITTLE.

Una forma de intentar paliar este problema consiste en forzar a que los núcleos lentos también ejecuten tareas críticas; esta decisión, sin embargo, puede provocar un retraso en la finalización de las tareas críticas, ya que aunque su ejecución puede comenzar antes (pues se disponen de más núcleos para distribuir el mismo número de tareas), el rendimiento de los núcleos LITTLE es muy inferior que el de los núcleos big, provocando retrasos en la ejecución e incluso llegando a agravar el cuello de botella anteriormente mencionado.

La política P1 intenta solventar este problema desde un enfoque distinto. La idea principal consiste en aprovechar los períodos de tiempo en los que la carga de trabajo sobre los núcleos lentos es menor para disminuir el consumo energético, *forzando una reducción de frecuencia* sobre el cluster LITTLE. Como se podía ver en la Figura 5.1, reducir la frecuencia al cluster LITTLE implica que la potencia instantánea disipada disminuye. Aunque es cierto que al reducir la frecuencia de los núcleos el tiempo empleado en ejecutar una tarea aumenta, esta técnica únicamente se aplica en aquellas fases de la ejecución paralela en las que la ejecución está limitada por el gran número de tareas críticas y el bajo número de tareas no críticas; por tanto, es esperable que el impacto final en el rendimiento no sea elevado, y sí lo sea la reducción de consumo energético.

La forma en la que se ha aplicado esta política sobre BOTLEV consiste en monitorizar constantemente el número de tareas, tanto críticas como no críticas, listas para ser ejecutadas

Listado 5.1: Fragmento de código esquemático para la política P1.

```
1  int P1(int bigQueueSize, int littleQueueSize){
2    int nxtFreq;
3
4    if( littleQueueSize==0 ) nxtFreq = FreqCfg::minLittleFreq;
5    else if( bigQueueSize==0 ) nxtFreq = FreqCfg::maxLittleFreq;
6    else{
7      int idx = min((bigQueueSize / littleQueueSize),
8                    FreqCfg::maxIdxLittle );
9      nxtFreq = FreqCfg::littleFreqs[idx];
10   }
11   return changeLittleFreq(nxtFreq);
12 }
```

(reflejado en el tamaño de las colas internas del planificador), y actuar en función de la relación entre el tamaño de ambas colas. El Listado 5.1 muestra un fragmento esquemático del código encargado de realizar esta tarea. El método es invocado cada vez que el tamaño de una cola es modificado (ya sea porque una tarea comienza su ejecución, o porque una tarea se encuentra lista para ser ejecutada y es insertada en una cola). Como se puede observar en la línea 7, la frecuencia a la que cambiar el cluster se calcula como una proporción directa entre el tamaño de ambas colas; así, por ejemplo, si el número de tareas críticas es el doble que el de las tareas no críticas, la frecuencia se disminuye un "escalón"; si el tamaño es el triple, la frecuencia se disminuye hasta el tercer escalón de frecuencia, etc. Cabe recordar que los valores de frecuencia que puede tomar el cluster están limitados por el kernel del sistema operativo, como se mencionó en la Sección 5.1.1. La línea 8 asegura que la frecuencia final no es menor que la menor frecuencia soportada. Por tanto, en esta política, el cambio de frecuencia en el cluster se realiza de manera escalonada según varía el tamaño de las colas.

Adicionalmente al comportamiento general de esta política, hay que distinguir dos casos especiales que merecen ser mencionados: el caso en el que no exista ninguna tarea crítica lista para ser ejecutada, y el caso opuesto, en el que todas las tareas listas sean críticas y los núcleos lentos estén totalmente ociosos. En estos casos, las líneas 4-5 se encargan de aumentar la frecuencia al máximo directamente en caso de que todas las tareas sean no críticas, o de disminuir la frecuencia al mínimo en caso de que no exista ninguna tarea no crítica que ejecutar.

La Figura 5.2 muestra el comportamiento de esta política para una factorización de Cholesky sobre una matriz de dimensión 1024×1024 usando precisión simple y dividida en bloques de 64×64 elementos. La gráfica superior muestra el estado de las colas durante la

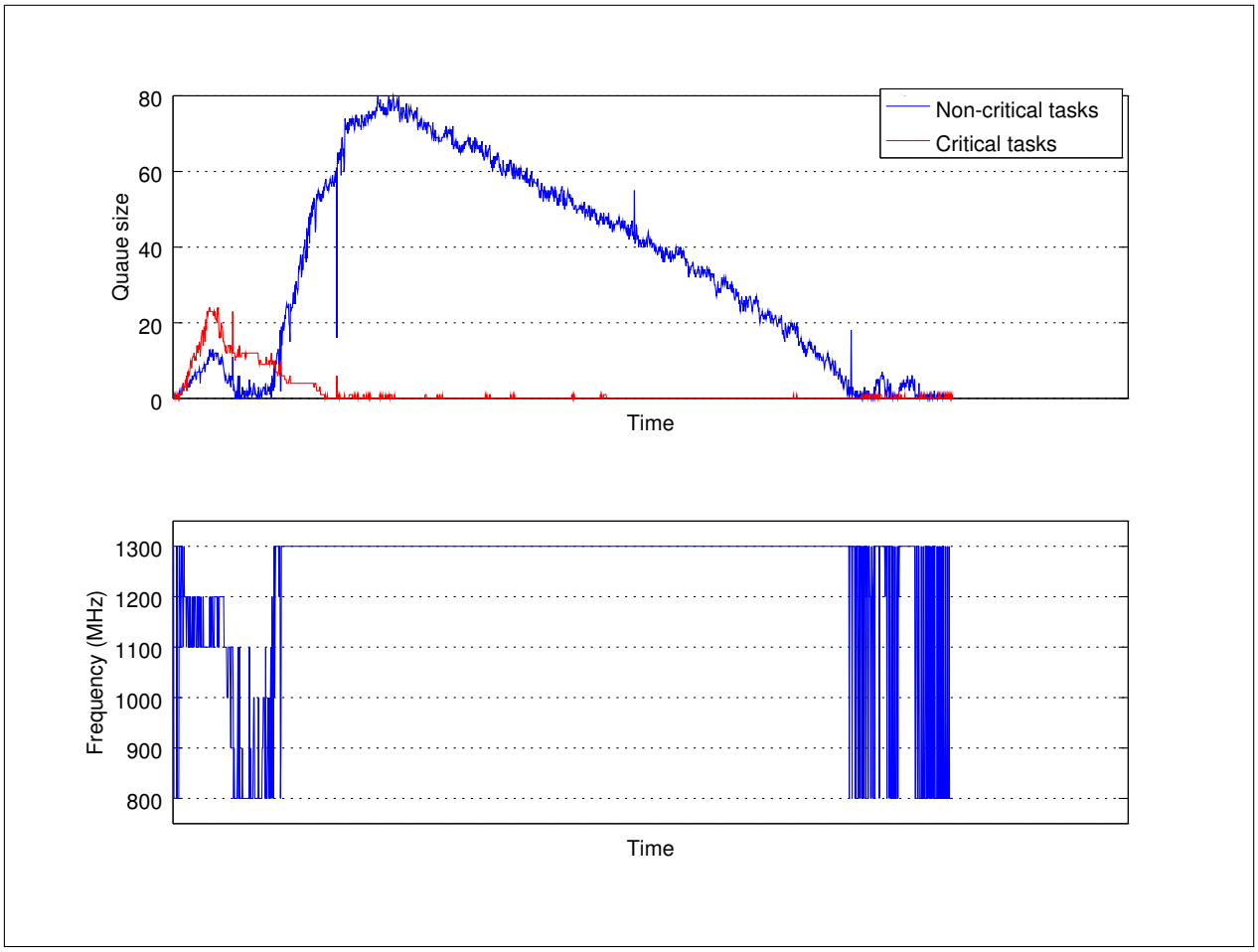


Figura 5.2: Cambio de frecuencias según la política P1 para una factorización de Cholesky sobre una matriz de 1024 elementos dividida en bloques de 64, ejecutada sobre la plataforma JUNO.

ejecución del problema, mientras que la gráfica inferior muestra la frecuencia del cluster de núcleos LITTLE en cada momento de la ejecución del problema. Como se puede apreciar, al inicio de la ejecución (cuando el número de tareas críticas es elevado), la frecuencia se reduce de manera escalonada en función de la relación entre el tamaño de ambas colas. Una vez el número de tareas no críticas aumenta, el cluster se mantiene a máxima frecuencia durante la mayor parte de la ejecución hasta que ésta se encuentra en las fases finales donde el número de tareas críticas vuelve a ser elevado respecto al número de tareas no críticas y la frecuencia vuelve a disminuirse.

Políticas P2 y P2': Escalado sobre el cluster LITTLE en función de la carga de trabajo

La primera intuición que surge al intentar aplicar un escalado de frecuencia sobre una arquitectura asimétrica es la de pensar que debido a la gran diferencia de rendimiento entre los núcleos big y LITTLE (como muestra la Figura 5.1), aplicar una reducción de frecuencia a los núcleos big puede suponer una pérdida significativa de rendimiento, lo cual puede provocar incluso un aumento en el consumo energético al acarrear un mayor tiempo global de ejecución; es decir, esta técnica podría llegar a empeorar tanto el rendimiento como el consumo energético. Para evitar que suceda este hecho, las políticas P2 y P2' han sido diseñadas para modificar la frecuencia exclusivamente sobre el cluster de núcleos LITTLE, con el fin de no causar gran impacto sobre el rendimiento global. Además, las políticas han sido desarrolladas sobre el planificador BOTLEV descrito en la sección 3.1.4, con el objetivo de asegurar que las tareas críticas se ejecutan en los núcleos big y así evitar que éstas retrasen su ejecución si se ejecutan en un núcleo LITTLE con la frecuencia disminuida.

Para determinar cuándo variar la frecuencia del cluster, las políticas P2 y P2' tienen en cuenta el número de tareas listas para ser ejecutadas; así, si existen suficientes tareas listas para ser ejecutadas, la frecuencia será alta para finalizar la ejecución cuanto antes, mientras que si existen pocas tareas listas para ser ejecutadas, la frecuencia será menor, intentando favorecer el ahorro energético. Este planteamiento es similar a tener en cuenta la carga de trabajo del sistema, pero medida en número de tareas pendientes en vez de ciclos ociosos/ocupados del procesador. Más concretamente, las políticas monitorizan el tamaño de la cola de tareas listas para ser ejecutadas, y determinan cuál es el tamaño máximo de la cola hasta el momento de manera dinámica. Si la cola posee un tamaño igual o superior al tamaño máximo conocido, significa que el número de tareas es elevado y por tanto la frecuencia debe ser elevada. Si el tamaño es menor, entonces se determina en qué porcentaje es menor, y en función de las frecuencias consideradas se toma la decisión de modificar la frecuencia actual o no.

En el Listado 5.2 se muestra el pseudocódigo asociado a estas políticas. El bloque `if` de la línea 3 es el encargado de determinar si el tamaño actual de la cola es mayor o igual que cualquier tamaño observado hasta el momento, y en caso de ser así, configurar el cluster para que funcione a máxima frecuencia. Esta comprobación es equivalente a determinar si hay un pico de carga de trabajo. Las líneas 10-11 determinan el número de tareas que separan una frecuencia de la otra. La forma de determinar esta cantidad consiste en repartir de manera equitativa el espacio máximo de tareas conocido hasta el momento entre todas las frecuencias, y asignar la frecuencia actual en función de qué tamaño posea la cola (línea 13).

Listado 5.2: Pseudocódigo para las políticas P2 y P2'.

```
int P2(int littleQueueSize, int bigQueueSize){
  //Comprobamos si estamos en un pico de carga
  if(littleQueueSize >= FreqCfg::maxQueueSize){
    FreqCfg::maxQueueSize = littleQueueSize;

    return changeLittleFreq(FreqCfg::maxLittleFreq);
  }

  //Tamanyo para cambiar de frecuencia
  float tamStep = (FreqCfg::maxQueueSize*1.0 /
                   (FreqCfg::maxIdxLittle+1)*1.0);
  //Escalon actual
  int step = (int) (littleQueueSize / tamStep);

  return changeLittleFreq(FreqCfg::littleFreqs[step]);
}
```

Por ejemplo, si el cluster es capaz de funcionar a 5 frecuencias distintas, y hasta el momento el número máximo de tareas preparadas para ser ejecutadas ha sido de 20 tareas, antes de cambiar de frecuencia se dispone de un margen de 4 tareas. Si el tamaño actual de la cola es de 3 tareas, la frecuencia del cluster será la mínima, mientras que si es de 5 tareas, la frecuencia será la frecuencia inmediatamente superior a la frecuencia mínima.

La diferencia entre las políticas P2 y P2' radica en el rango de frecuencias que se consideran para el cluster. Así, mientras la política P2 divide el tamaño de la cola entre todas las frecuencias posibles para el cluster, la política P2' solamente considera la frecuencia máxima y mínima del mismo.

Políticas P3: Escalado sobre el cluster big en función de la carga de trabajo

La política P3 es similar a la política anterior P2, pero realizando el escalado de frecuencias sobre el cluster de núcleos big en lugar de sobre los núcleos LITTLE. De manera similar a la anterior, la política ha sido implementada sobre el planificador Botlev implementado en OmpSs para minimizar el impacto negativo sobre las tareas críticas. La Figura 5.3 muestra los distintos cambios de frecuencia que se han realizado en función del número de tareas listas para ser ejecutadas sobre un ejemplo concreto. En la gráfica superior se muestra la evolución del número de tareas listas para ser ejecutadas según avanza la ejecución del problema, mientras que en la gráfica inferior se muestra la frecuencia del cluster big durante la ejecución del problema. Ambas gráficas poseen la misma escala en el eje x, permitiendo relacionarlas de manera visual. La gráfica corresponde a una ejecución sobre la platafor-

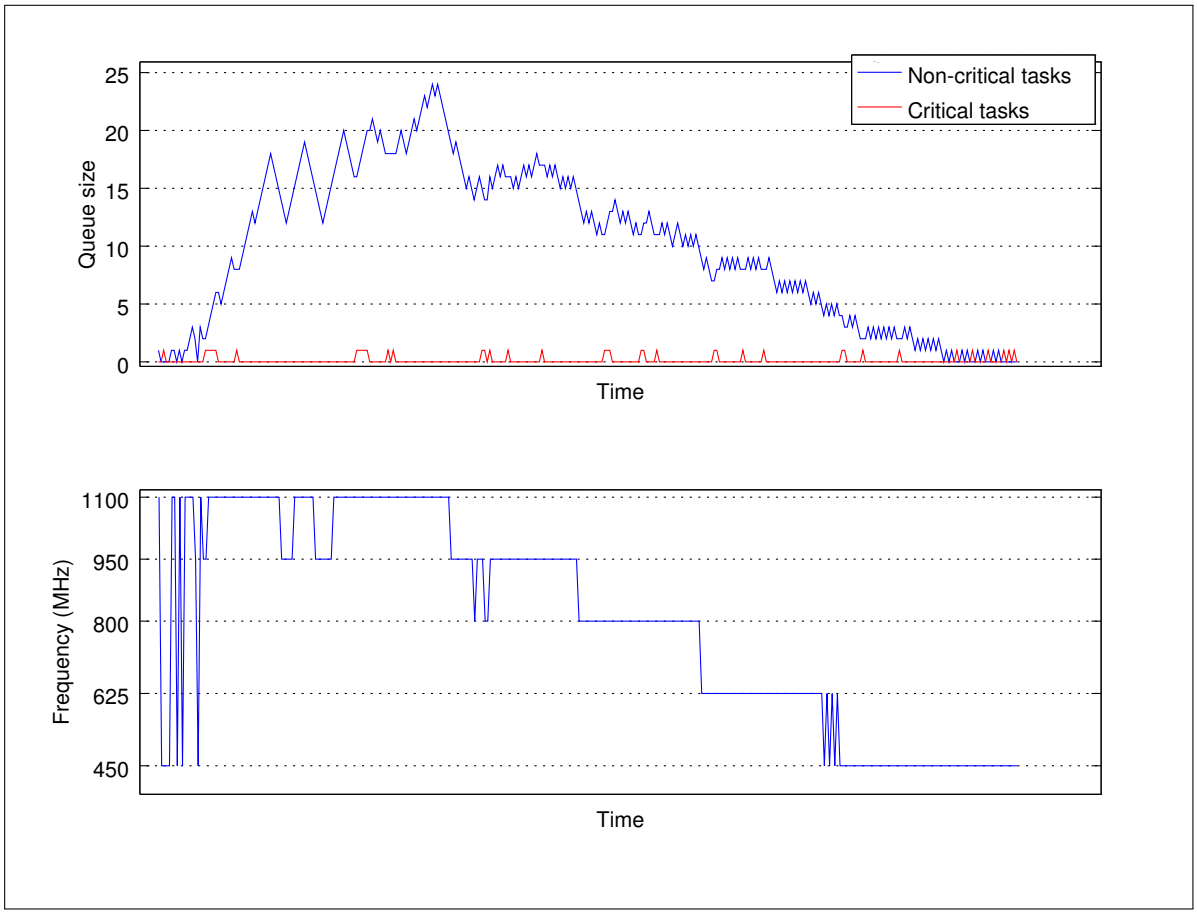


Figura 5.3: Escalado de frecuencia en función del número de tareas listas según la política P3. La ejecución corresponde a una factorización de Cholesky sobre una matriz de 4608 elementos en precisión simple, dividida en bloques de tamaño de 512 elementos, ejecutada sobre la plataforma de desarrollo JUNO (donde los núcleos big corresponden a núcleos ARM-A57).

ma JUNO para una factorización de Cholesky sobre una matriz de 4608×4608 elementos dividida en bloques de dimensión 512 en precisión simple.

Como se puede observar en la gráfica, la ejecución se puede dividir en dos partes bien diferenciadas: una primera parte hasta que la cola alcanza el tamaño máximo, y una segunda parte en la que el número de tareas listas desciende constantemente. Estas dos partes están totalmente relacionadas con el grafo de dependencias de una factorización de Cholesky, donde existe una primera fase en la que se crean un gran número de tareas y se expande el árbol, y una segunda fase donde las tareas se van ejecutando y el árbol se cierra paulatinamente.

Debido a que el cálculo del tamaño máximo de la cola se realiza de manera dinámica, en la primera fase se realizan cambios de frecuencia continuamente a causa de ligeros cambios en el tamaño de la cola hasta que se consigue estabilizar la cota superior, y poder aplicar la política

de manera correcta. Una vez alcanzado el tamaño máximo de la cola, se observa como el espacio está dividido en varios escalones, cada uno correspondiente a una frecuencia distinta, y según el tamaño de la cola va disminuyendo y cambiando de escalón, las frecuencias son modificadas. Puede darse el caso en el que el tamaño de la cola esté variando continuamente entre dos escalones, produciendo que la frecuencia oscile hasta que el tamaño de la cola se vuelva a estabilizar. Esta situación se puede apreciar al principio del escalón asociado a 950MHz, donde se aprecia como un ligero descenso del número de tareas produce que la frecuencia oscile entre 800MHz y 950MHz.

5.2.2. Políticas basadas en planificación de tareas

Las políticas descritas hasta el momento intentan obtener una mejora en el rendimiento energético a partir de un escalado de frecuencia en los distintos clusters de la arquitectura, prestando atención al número de tareas listas para ser ejecutadas como medida equivalente a detectar el nivel de carga de trabajo en un cierto momento. Estas políticas se han desarrollado prestando atención a la Figura 5.1, en la que se puede observar la disminución del consumo energético en función de la frecuencia. La siguiente idea que surge prestando atención a la misma figura consiste en disminuir el consumo al no utilizar algún cluster durante parte de la ejecución. Como se puede observar, el consumo energético es menor si el cluster no presenta carga, independientemente de la frecuencia utilizada. Las siguientes políticas han sido diseñadas con el objetivo de que, en ciertos momentos de la ejecución, se evite asignar tareas a cierto cluster del sistema, modificando el planificador de tareas. Es decir, estas políticas no están orientadas a realizar técnicas de escalado de frecuencia, sino que tratan con el problema de planificación de tareas a los distintos *worker threads* disponibles.

Políticas P4 y P5: Inhabilitación del cluster según la carga de trabajo

La política P4 monitoriza el tamaño de las colas de tareas listas y, en función del tamaño de la cola respecto al tamaño máximo (similar a las políticas P2 o P3), decide si asignar tareas listas al cluster de núcleos LITTLE o por el contrario no hacerlo. En caso de que el tamaño de la cola sea lo suficientemente pequeño, la política no asigna ninguna tarea lista al cluster, provocando que éste se encuentre ocioso. Cabe destacar que, aunque el planificador no asigne tareas al cluster, esto no implica que el sistema operativo no ejecute procesos sobre él; aún así, según se ha observado, el impacto final sobre el consumo energético es mínimo comparado con el de ejecutar una tarea.

Para realizar este proceso, cada vez que una tarea se inserta en la cola de tareas listas o

es extraída de ésta, se compara el tamaño actual de la cola con el tamaño máximo conocido hasta el momento, y se decide si se debe o no utilizar el cluster para ejecutar tareas. Una vez que un *worker thread* finaliza la ejecución de una tarea, éste solicita una nueva tarea a ejecutar al planificador. Si el *worker thread* se ejecuta sobre un núcleo del cluster LITTLE, y se ha decidido que el cluster no ejecute ninguna acción, al *worker thread* se le asigna una tarea nula, evitando así la ejecución de cualquier tarea en ese core asociado al *worker thread* y perteneciente al cluster inhabilitado. En caso de que el cluster no se encuentre desactivado se asigna la siguiente tarea de la cola de tareas listas para que la ejecute el *worker thread*.

Para que esta política funcione correctamente, es necesario que se verifiquen algunas condiciones cuando se ejecute la aplicación sobre el *runtime*:

- El número de *worker threads* lanzados debe coincidir con el número de núcleos del SoC. En el caso en el que el número de *worker threads* es superior, varios *worker threads* compiten por ejecutarse simultáneamente en el mismo núcleo. De manera contraria, si el número de *worker threads* es menor que el número de núcleos, se están desaprovechando recursos ya que no se utilizan todos los núcleos de manera simultánea. En NANOS++ esto se consigue mediante el argumento `-smp-workers=<n>`.

- Cada *worker thread* debe estar asignado a un núcleo único de la placa, provocando que un *worker thread* siempre ejecute tareas en el mismo núcleo durante todo el problema. Esta condición unida a la anterior implica que existe una relación directa (*afinidad*) entre cada núcleo y cada *worker thread* del *runtime*.

- Aunque se tenga control sobre qué tarea se ejecuta en qué *worker thread* (y por tanto en qué núcleo), existen ciertas tareas pertenecientes al propio *runtime* o al código no paralelo del programa sobre las que no se pueden tomar decisiones desde el planificador, siendo normal que este código se ejecute sobre el *worker thread* principal, normalmente ejecutado sobre el núcleo 0 de la máquina.

En las políticas anteriores, donde se disponía de un número limitado de frecuencias, el momento en el que aplicar el cambio de frecuencia venía determinado por el número de frecuencias posibles para el cluster, repartiendo el número máximo de tareas listas de manera equitativa entre todas ellas. Para dar mayor flexibilidad a esta política, el punto en el que tomar la decisión de apagar el cluster viene determinado por una variable de entorno elegida por el usuario a la hora de lanzar el experimento. Esto permite observar el comportamiento variando el momento en el que se desactiva el cluster, como se describe en la siguiente sección. Además, para evitar efectos rebote, en los que el cluster se esté activando

Listado 5.3: Fragmento de código para determinar si asignar una tarea a un *worker thread* o no.

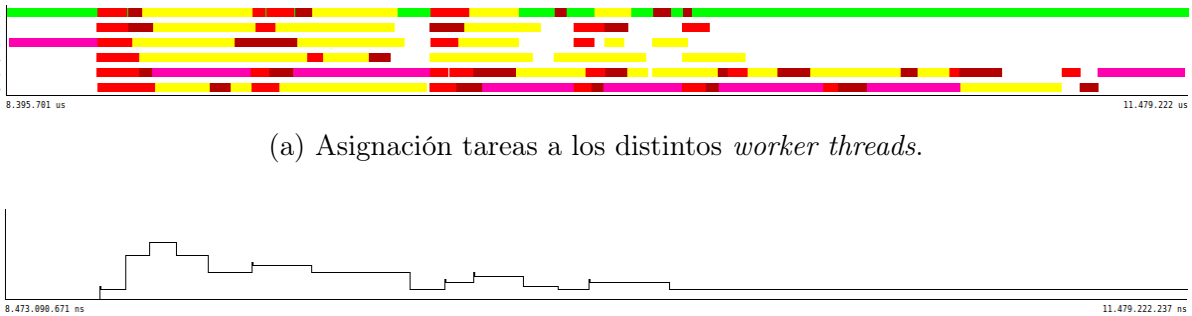
```
1  bool CoresCfg::puedoEjecutar(int coreId){
2    //core little
3    if(apagadoLittle)
4      for(int i=0; i<N_CORE_LITTLE; i++)
5        if(coresLittle[i] == coreId){
6          if (!coresApagados[coreId]){
7            FreqCfg::changeLittleFreq(FreqCfg::minFreqLittle);
8            coresApagados[coreId] = true;
9          }
10         return false;
11       }
```

y desactivando de manera continuada, mediante experimentación se ha decidido insertar un margen del 10 % del tamaño de la cola antes de tomar la decisión de apagar o encender el cluster.

El Listado 5.3 muestra un fragmento del código encargado de determinar si un *worker thread* debe ejecutar una tarea lista o no. La función recibe por parámetro el núcleo al que se encuentra asignado el *worker thread* y devuelve un valor *booleano* en función de si el cluster se encuentra desactivado o no. Para agilizar el proceso, la línea 3 filtra la ejecución y permite continuar con la ejecución únicamente en el caso de que el cluster se encuentre desactivado. La labor de determinar si el cluster debe estar activo o no se realiza en el momento en el que una tarea lista es insertada en la cola de tareas listas, o es extraída, y se realiza mediante un código similar al mostrado en el Listado 5.2. Si el cluster se encuentra desactivado, mediante las siguientes líneas se determina si el núcleo pertenece al cluster inactivo. En caso de pertenecer, se devuelve el valor de falso para indicar que el *worker thread* no debe ejecutar tareas, y además se reduce la frecuencia de núcleo al mínimo para intentar minimizar el consumo energético mientras no se utilice el cluster de núcleos LITTLE (línea 7).

La política P5 realiza el mismo procedimiento, pero para el cluster de núcleos big. Ambas políticas están implementadas sobre el planificador Botlev para así asegurar que la ejecución de tareas críticas sigue siendo en núcleos big. Sin embargo, al desactivar los núcleos big, hay que tener en cuenta que es necesario que las tareas críticas se ejecuten ahora en núcleos LITTLE si los big no están disponibles para que la ejecución no se quede bloqueada por las tareas críticas (la opción `-steal=1` en Botlev activa el *work-stealing* en Nanos++).

La Figura 5.4 muestra un ejemplo de esta política aplicada sobre el cluster de núcleos LITTLE sobre una factorización de Cholesky. La gráfica superior muestra la ejecución de las diferentes tareas por los distintos *worker threads*, cada uno asignado a un núcleo distinto



(a) Asignación tareas a los distintos *worker threads*.

(b) Evolución del tamaño de la cola de tareas listas.

Figura 5.4: Asignación de tareas en función del tamaño de la cola según la política P4. La gráfica superior muestras las distintas tareas ejecutadas por cada uno de los *worker threads* a lo largo del tiempo. Clave de colores: rojo=TRSM, rosa=POTRF, granate=SYRK, amarillo=GEMM, verde=MAIN. La ejecución corresponde a una factorización de Cholesky para una matriz de 4096 elementos dividido en bloques de tamaño 512 en precisión simple sobre una arquitectura JUNO, donde los 4 primeros *worker threads* corresponden a los núcleos lentos y los 2 últimos a los núcleos rápidos. La política está configurada para desactivar núcleos a partir del 25 % del tamaño máximo de la cola.

en orden de aparición. La ejecución ha sido realizada sobre la plataforma JUNO, por lo que los 4 primeros núcleos corresponden a núcleos LITTLE, mientras que los dos siguientes a núcleos big. La segunda gráfica muestra la evolución del tamaño de las colas en función del tiempo. Como se puede observar, la ejecución de las tareas se realiza de manera convencional hasta que el tamaño de la cola decrece por debajo del 25 % del tamaño máximo (en este caso, el experimento se configuró con este valor con fines meramente ilustrativos). En ese momento, los núcleos LITTLE dejan de recibir tareas mientras que los big continúan su ejecución habitual. Una vez que el número de tareas vuelve a incrementarse, los núcleos lentos vuelven a activarse y a ejecutar tareas, hasta que al final de la ejecución el tamaño de la cola vuelve a decrecer y el cluster se desactiva hasta finalizar la ejecución. Aunque el primer *worker thread* está asociado a un núcleo LITTLE, también es considerado como el *worker thread* principal del planificador, por lo que es el encargado de ejecutar la parte del código no paralela (en color verde). La mayor parte del tiempo que se encuentra ejecutando este código corresponde a tareas del propio planificador, o a la espera de que finalice la ejecución paralela (marcada por una etiqueta `#pragma omp taskwait` en el código secuencial), por lo que el impacto final sobre el consumo energético no es significativo.

Política P6: Desactivación de núcleos en función de la carga de trabajo

Una de las muchas funcionalidades que ofrece el kernel Linux es la de poder activar y desactivar núcleos bajo demanda y de manera dinámica. En cualquier momento de la ejecución, un núcleo puede ser desactivado, y éste desaparece del sistema, siendo totalmente transparente para cualquier aplicación en ejecución. Esto implica que si se decide apagar un núcleo, todos los procesos asociados a ese núcleo son migrados a otro de manera totalmente transparente al proceso en ejecución, el número total de núcleos disminuye y cualquier llamada a una función del kernel considerará la nueva configuración hardware creada como la configuración real de la máquina. Esta funcionalidad permite, por ejemplo, generar dinámicamente arquitecturas diferentes utilizando el mismo hardware.

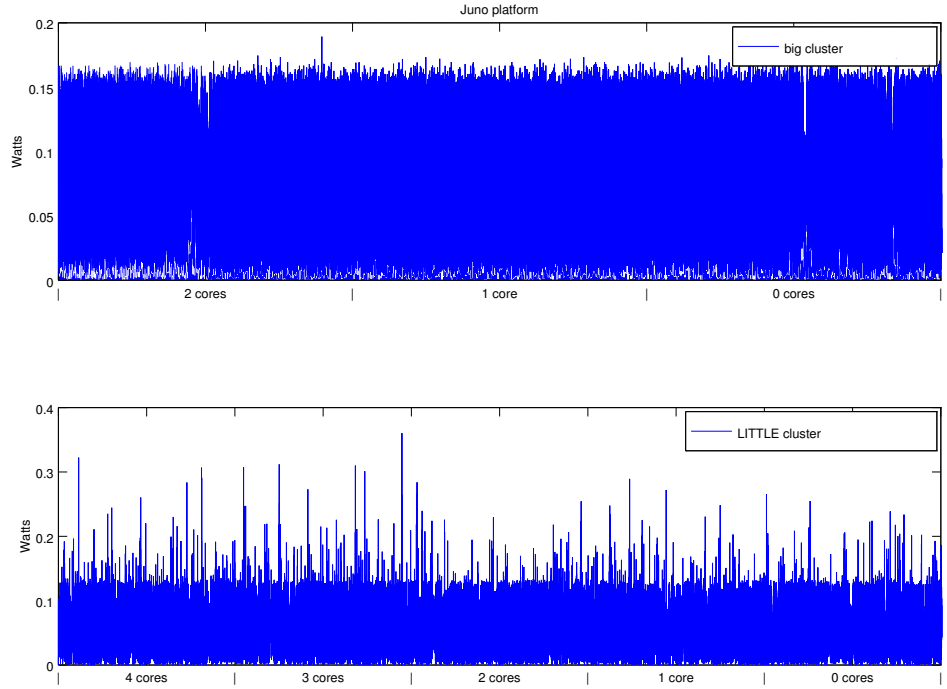
Esta funcionalidad aplicada a una arquitectura big.LITTLE teóricamente conlleva que en cualquier momento de la ejecución se pueda desactivar uno de los dos cluster de la máquina, convirtiendo la arquitectura en una plataforma totalmente simétrica [1].

Para realizar un estudio del efecto del apagado selectivo de núcleos sobre el consumo energético, se ha realizado el experimento de desactivar, uno a uno, los núcleos de un mismo cluster mientras se monitoriza el consumo. Además, para evitar errores en las medidas, se ha establecido un tiempo de espera entre apagado de dos núcleos, y adicionalmente se ha ejecutado el test asociado a un núcleo que no se fuera a apagar, evitando así problemas al migrar el proceso de un núcleo a otro. La Figura 5.5 muestra las mediciones del consumo energético en las dos plataformas: JUNO (gráfica superior) y ODROID (gráfica inferior).

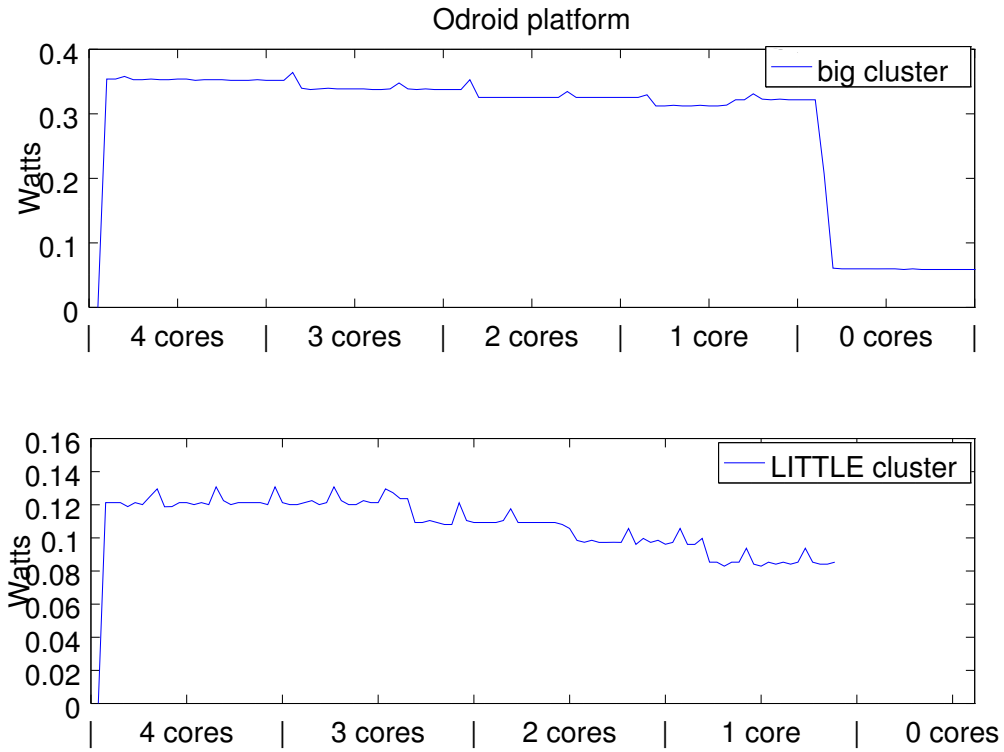
Las gráficas muestran la potencia instantánea observada (en Watios) durante todo el experimento en el eje vertical, mientras que el eje horizontal corresponde a las distintas fases por las que ha ido pasando el experimento. Por cada plataforma, la gráfica superior muestra el experimento apagando los núcleos del cluster big, indicándose en el eje x el número de núcleos activos del cluster en cada momento; la gráfica inferior muestra el mismo experimento para el cluster LITTLE. La diferencia de aspecto entre las gráficas de ambas plataformas se produce por la forma en la que se han realizado las medidas: mientras que en la plataforma ODROID los medidores poseen una frecuencia de muestreo máxima de 4 muestras por segundo (y por eso el aspecto *lineal* de la gráfica), el entorno de medición de la plataforma JUNO permite una mayor frecuencia de muestreo, por lo que la gráfica presenta un mayor ruido (esta es la causa de que la gráfica no presente una forma de línea, sino que los valores oscilan por encima y debajo del valor medio real).

Observando los resultados de las gráficas se pueden extraer las siguientes conclusiones:

[1] En la práctica, no todas las versiones del kernel permiten apagar el núcleo 0, por lo que en estos casos, no es posible apagar completamente el cluster que contiene a este núcleo



(a) Plataforma JUNO: dos núcleos big A57 y cuatro núcleos LITTLE A53.



(b) Plataforma ODROID: cuatro núcleos big A15 y cuatro núcleos LITTLE A7.

Figura 5.5: Consumo energético en reposo en función del número de núcleos activos del sistema y de la plataforma. La versión del kernel Linux instalada en la plataforma ODROID no permite apagar el cluster LITTLE de manera completa.

1. En la plataforma JUNO, desactivar núcleos es equivalente a no usarlos en lo referente al consumo energético.

2. Mientras que la versión del kernel Linux utilizada en la plataforma JUNO permite apagar todos los núcleos del sistema, la versión utilizada en la plataforma ODROID no permite apagar el núcleo 0, por lo que no es posible desactivar el cluster LITTLE completo.

3. En la plataforma ODROID, desactivar un cierto número de núcleos sin llegar a desactivar el cluster completo no tiene ningún impacto significativo en el consumo energético. Sin embargo, para esta plataforma, desactivar todos los núcleos del cluster big conlleva que la potencia instantánea caiga a niveles prácticamente nulos.

Aprovechando la caída de potencia cuando el cluster de núcleos big se desactiva por completo en la plataforma ODROID, la política P6 ha sido desarrollada como una modificación de la política anterior, pero además de no asignar tareas a estos núcleos, realiza un apagado controlado de los mismos. Para realizar esta modificación, el cambio introducido sobre el código del Listado 5.3 afecta a la línea 7, en la que en vez de disminuir la frecuencia del cluster, se realiza una llamada al código encargado de realizar el apagado de los distintos núcleos, mostrado en el Listado 5.4. Como se puede observar, el apagado del núcleo se realiza escribiendo el valor 0 en el fichero del sistema (SYSFS) que se encuentra en la ruta `/sys/devices/system/cpu/cpu<cpuId>/online`. Esta operación se ha protegido mediante un cerrojo para evitar problemas de concurrencia durante la ejecución.

5.3. Resultados experimentales

5.3.1. Análisis de la mejora energética para las políticas P1-P3

El primer conjunto de experimentos que se muestran a continuación tiene como objetivo principal comprobar el nivel de eficiencia energética desde la política P1 hasta la política P3, las cuales corresponden a políticas que siguen una estrategia puramente DVFS como se ha descrito anteriormente[2]. Los experimentos corresponden a distintas ejecuciones de una factorización de Cholesky para distintas configuraciones de tamaño de matriz y bloque. Cada configuración del problema ha sido lanzada 10 veces, y obtenido la media de los resultados para intentar disminuir los posibles errores que hubiera en las mediciones. Para comprobar si estas políticas presentan una mejora energética real, los resultados son comparados con

[2]En este capítulo todos los experimentos han sido realizados utilizando precisión simple.

Listado 5.4: Fragmento de código para el apagado de núcleos de manera dinámica.

```
1  char path[50];
2  sprintf(path, "/sys/devices/system/cpu/cpu%d/online", coreId);
3
4  FILE* f;
5  int d, a;
6
7  (CoresCfg::_lock[coreId])->acquire();
8  {
9    f = fopen(path, "r+");
10   a = fscanf(f, "%d", &d);
11
12   if(d!=0 && a!=0){
13     fseek(f, 0, SEEK_SET);
14     fprintf(f, "0"); //Apagado
15   }
16
17   fclose(f);
18 }
19 (CoresCfg::_lock[coreId])->release();
```

una ejecución convencional utilizando el planificador BOTLEV sobre el *runtime* OmpSs (esta ejecución se denomina P0 en los experimentos realizados). Para poder observar el comportamiento con detalle y poder extraer mejores conclusiones, de todos los experimentos se extraen las tres siguientes medidas:

1. Rendimiento computacional medido en GFLOPS[3]. Distintos rendimientos sobre la misma configuración del problema sirven para indicar diferencias en el tiempo de ejecución.

2. Muestreo de la potencia media (en Watios). Los medidores permiten extraer el muestreo de potencia diferenciado por clusters, aunque en este caso se muestra agregada para ambos cluster.

3. Eficiencia o rendimiento energético medido en GFLOPS/Watio, como resultado de combinar las dos medidas anteriores.

En las Figuras 5.6 y 5.7 se muestran de manera conjunta los resultados obtenidos para la plataforma JUNO y para la plataforma ODROID, respectivamente. Las gráficas representan de manera conjunta las tres métricas mencionadas anteriormente para las políticas P1 a P3 para un amplio conjunto de tamaños de matriz y bloque.

A grandes rasgos, es posible extraer un conjunto de observaciones generales:

[3] 1 GFLOP equivale a $1e9$ operaciones en punto flotante por segundo.

- Atendiendo exclusivamente al tamaño de problema, se pueden distinguir dos grupos bien diferenciados: un primer grupo formado por las matrices de tamaño pequeño ($m \leq 2048$), y un segundo grupo formado por las matrices de mayor tamaño. La principal diferencia radica en el rendimiento de las distintas políticas respecto a la ejecución normal sobre BOTLEV: mientras que en el segundo grupo el rendimiento se aproxima al de una ejecución normal con BOTLEV (aunque casi siempre inferior), en el primer grupo la diferencia de rendimiento es muy elevada, siempre a favor de BOTLEV. Esta diferencia provoca no solo un rendimiento computacional menor, sino una eficiencia energética considerablemente inferior. Además, hay que destacar que esta diferencia es más elevada en la plataforma ODROID que en la plataforma JUNO.

- El comportamiento de todas las políticas es similar en ambas plataformas, dando una mayor validez a los datos. Además, la proporción entre el número de núcleos big y LITTLE varía entre una plataforma y otra, dando mayor relevancia a los datos.

- Mientras que las políticas P1, P2 y P2' obtienen un rendimiento energético muy similar a la ejecución convencional sin aplicar ninguna política, la política P3 obtiene un rendimiento energético mucho mayor que la ejecución con BOTLEV sobre OmpSs.

Prestando más atención a la política P1, se observa que no presenta una gran diferencia respecto a la ejecución convencional utilizando BOTLEV. La causa de este hecho está relacionada con el DAG asociado a la factorización de Cholesky, el cual se caracteriza por una rápida explosión de tareas al comienzo de la ejecución, provocando que el número de tareas críticas sea inferior al número de tareas no críticas durante casi toda la ejecución. Como trabajo futuro se considera experimentar esta política sobre otros problemas donde su DAG se abra y cierre sucesivamente durante la ejecución.

Respecto a las políticas P2 y P2', no presentan una gran diferencia entre ellas en los resultados, tanto en el rendimiento en GFLOPS, como el consumo de energía. La razón de este parecido radica en que las políticas son prácticamente iguales, únicamente variando el conjunto de frecuencias sobre las que elegir para realizar el escalado de frecuencia. Respecto a la mejora del rendimiento energético, que es el objetivo final buscado, no se consigue superar el rendimiento alcanzado por una ejecución normal de BOTLEV; sin embargo, se observa que la potencia instantánea media consumida ha disminuido. Este dato, junto con el hecho de que el rendimiento energético (GFLOPS/watio) no disminuye considerablemente respecto a la ejecución normal, hacen que las políticas P2 y P2' representen una posible solución para arquitecturas cuya potencia instantánea máxima está limitada por motivos de diseño o por el entorno de ejecución (por ejemplo, sistemas alimentados por baterías). La

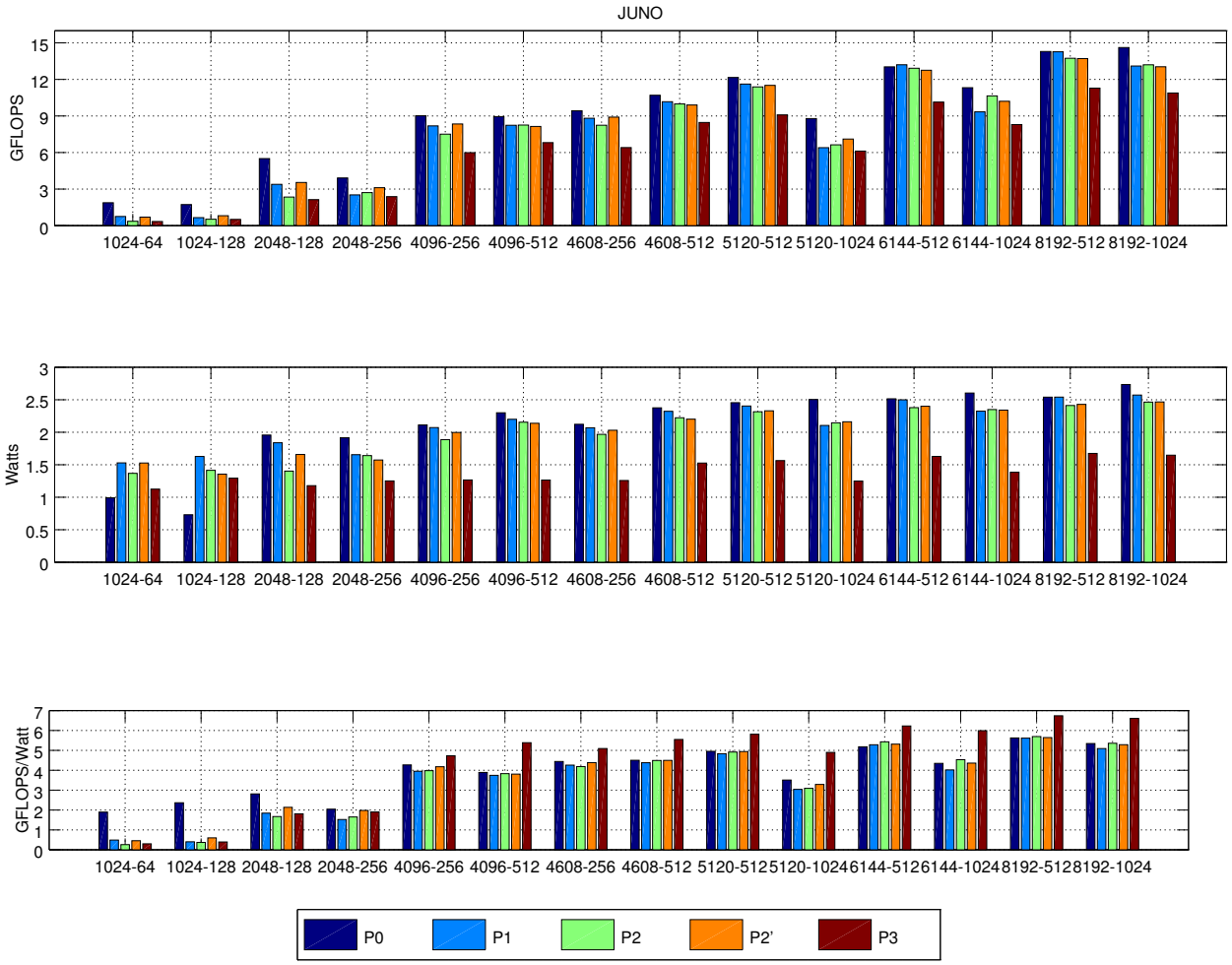


Figura 5.6: Medidas experimentales para las políticas desde P1 a P3 para la plataforma Juno. La política P0 representa a una ejecución normal con Botlev. Las etiquetas del eje x representan el tamaño de la matriz y de bloque utilizado para el experimento.

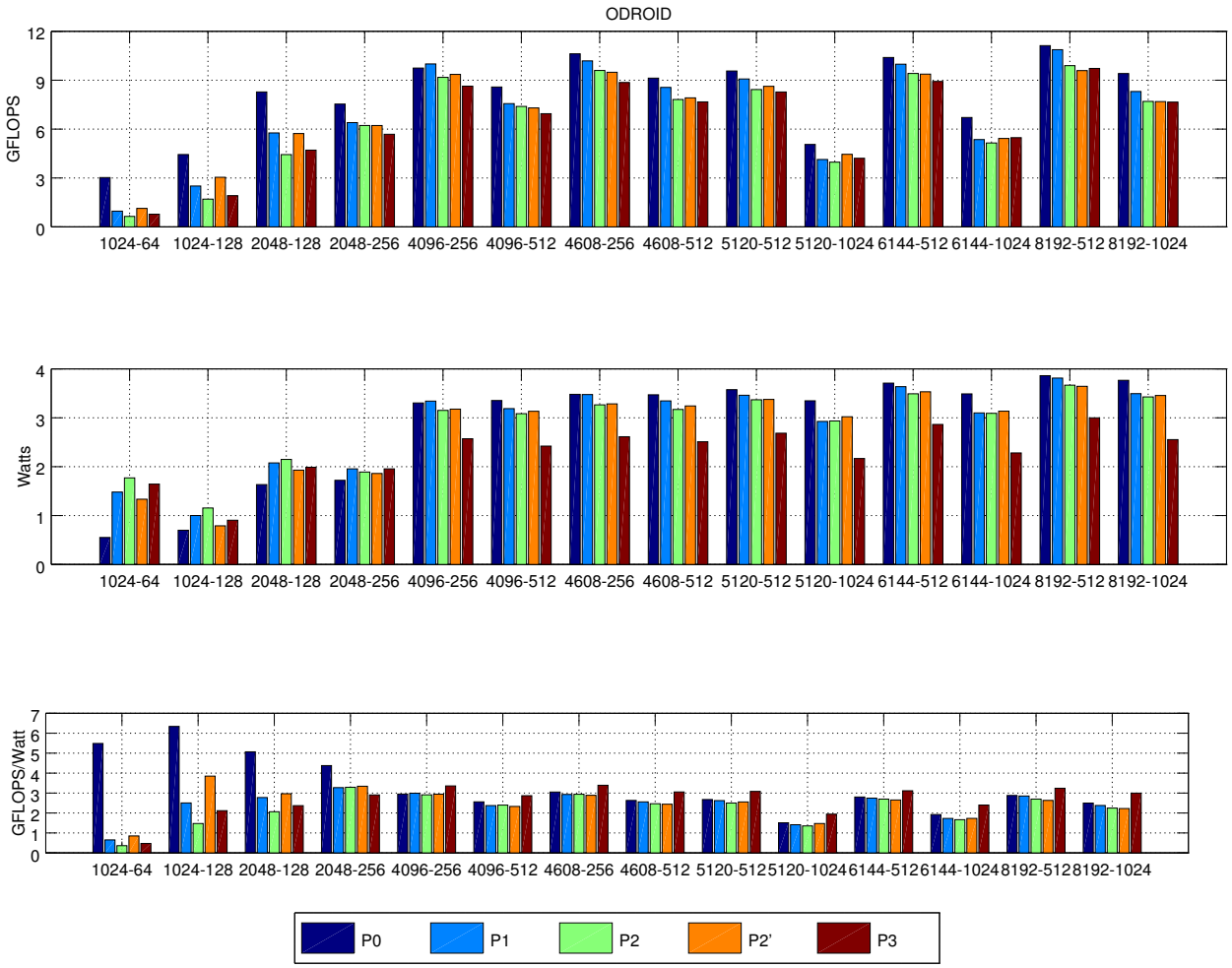


Figura 5.7: Medidas experimentales para las políticas desde P1 a P3 en la plataforma Odroid. La política P0 representa a una ejecución normal con Botlev. Las etiquetas del eje x representan el tamaño de la matriz y de bloque utilizado para el experimento.

Tabla 5.2: Mejora de la potencia instantánea media (en watios) para las políticas P1-P3.

	Tamaño de la matriz (m) y de bloque (b).													
(m)	1024		2048		4096		4608		5120		6144		8192	
(b)	64	128	128	256	256	512	256	512	512	1024	512	1024	512	1024
P1	-0.538	-0.897	0.118	0.260	0.040	0.100	0.053	0.054	0.051	0.399	0.016	0.279	-0.001	0.162
P2	-0.377	-0.684	0.555	0.274	0.226	0.145	0.155	0.153	0.139	0.359	0.136	0.254	0.129	0.271
P2'	-0.536	-0.625	0.299	0.342	0.115	0.162	0.092	0.175	0.125	0.343	0.115	0.263	0.111	0.268
P3	-0.134	-0.563	0.778	0.664	0.846	1.035	0.865	0.850	0.891	1.255	0.886	1.219	0.866	1.088

JUNO

	Tamaño de la matriz (m) y de bloque (b).													
(m)	1024		2048		4096		4608		5120		6144		8192	
(b)	64	128	128	256	256	512	256	512	512	1024	512	1024	512	1024
P1	-0.929	-0.301	-0.444	-0.234	-0.038	0.165	0.004	0.125	0.114	0.422	0.073	0.390	0.049	0.273
P2	-1.219	-0.455	-0.515	-0.169	0.153	0.269	0.215	0.300	0.207	0.413	0.219	0.398	0.194	0.340
P2'	-0.783	-0.090	-0.296	-0.140	0.126	0.220	0.195	0.228	0.199	0.327	0.177	0.354	0.218	0.311
P3	-1.092	-0.205	-0.354	-0.234	0.732	0.932	0.867	0.961	0.891	1.181	0.845	1.208	0.862	1.213

ODROID

Tabla 5.2 representa la disminución de potencia en watios respecto a la ejecución normal de BOTLEV. Como se puede observar, salvo el primer grupo de matrices que identificábamos anteriormente, el resultado es bueno para todas las políticas, obteniendo disminuciones de potencia cercanas a 0.25 watios para las políticas P2 y P2', y reducciones de hasta 1 watio para la política P3 sobre la plataforma JUNO.

Como se observa en las Figuras 5.6 y 5.7, la política que mayor rendimiento energético alcanza, incluso superando a una ejecución normal con BOTLEV es la política P3. Esta política, caracterizada por disminuir frecuencia sobre el cluster de núcleos big consigue, a costa de obtener un menor rendimiento, una mejora elevada de rendimiento energético frente al resto de políticas y a la ejecución normal. La Tabla 5.3 indica la mejora en términos gflops/watio de cada política y para cada configuración de tamaño tanto en la plataforma JUNO como en la plataforma ODROID. Como se puede observar, la mejora es muy significativa para matrices de tamaño mayor a 2048, alcanzando una mejora aproximada de 1 GFLOP/watio en la plataforma JUNO, y una mejora entorno a 0.4 GFLOP/watio en la plataforma ODROID. Adicionalmente a esta mejora, la política P3 también consigue reducir la potencia instantánea media en todas las configuraciones, siendo esta política también válida para plataformas donde la potencia esté limitada (similar a las políticas anteriores).

La razón de que la política P3 posea unos mejores resultados frente al resto de políticas se puede observar en la Figura 5.8, la cual muestra los datos obtenidos para una configuración concreta del problema (en este caso, para una matriz de 8192×8192 elementos divida en

Tabla 5.3: Mejora de rendimiento energético absoluta (en GFLOPS/Watio) para la factorización de Cholesky utilizando distintas políticas P1 a P3 respecto a una ejecución estándar del mismo problema utilizando el planificador BOTLEV sobre OmpSs, sobre la plataforma JUNO.

	Tamaño de la matriz (m) y de bloque (b).													
(m)	1024		2048		4096		4608		5120		6144		8192	
(b)	64	128	128	256	256	512	256	512	512	1024	512	1024	512	1024
P1	-1.405	-1.963	-0.968	-0.519	-0.318	-0.146	-0.183	-0.124	-0.119	-0.469	0.101	-0.331	-0.011	-0.252
P2	-1.635	-1.995	-1.139	-0.391	-0.289	-0.060	-0.255	-0.011	-0.036	-0.418	0.249	0.184	0.069	0.012
P2'	-1.438	-1.766	-0.675	-0.064	-0.092	-0.084	-0.056	-0.006	-0.012	-0.221	0.131	0.012	0.018	-0.058
P3	-1.594	-1.971	-1.001	-0.141	0.460	1.496	0.652	1.045	0.860	1.391	1.046	1.643	1.114	1.263

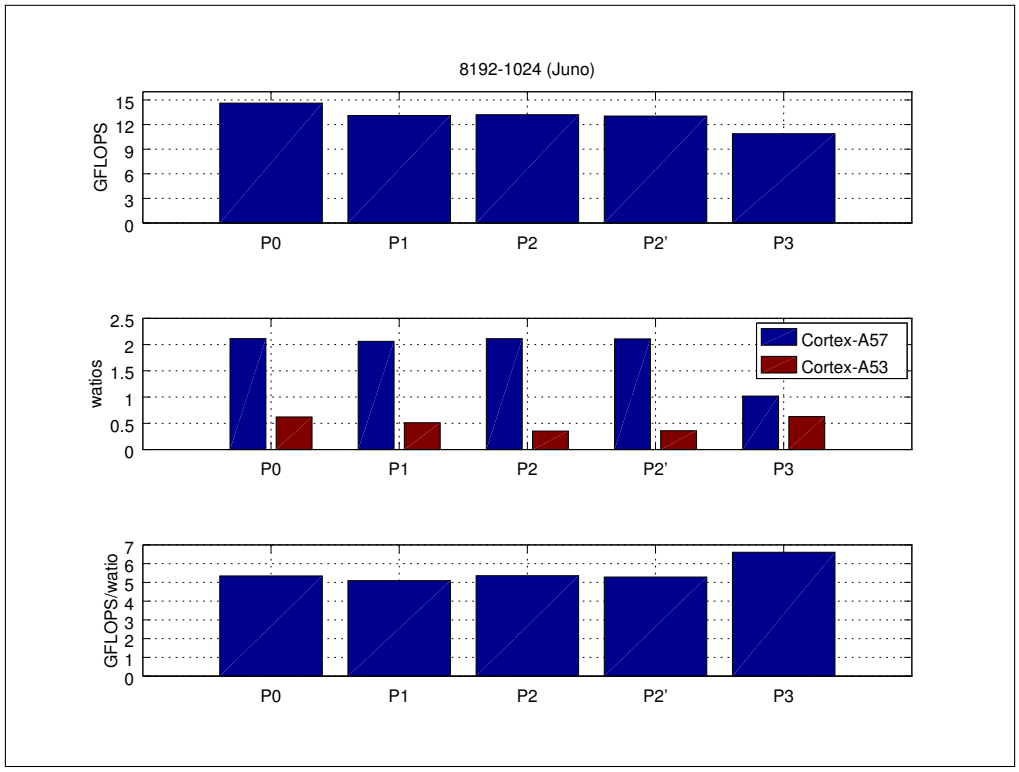
JUNO

	Tamaño de la matriz (m) y de bloque (b).													
(m)	1024		2048		4096		4608		5120		6144		8192	
(b)	64	128	128	256	256	512	256	512	512	1024	512	1024	512	1024
P1	-4.835	-3.835	-2.293	-1.106	0.042	-0.187	-0.121	-0.072	-0.053	-0.097	-0.057	-0.195	-0.028	-0.120
P2	-5.124	-4.866	-3.004	-1.087	-0.038	-0.163	-0.112	-0.165	-0.174	-0.158	-0.105	-0.258	-0.182	-0.251
P2'	-4.636	-2.490	-2.100	-1.036	-0.005	-0.230	-0.164	-0.188	-0.121	-0.037	-0.149	-0.193	-0.248	-0.274
P3	-5.014	-4.223	-2.699	-1.472	0.408	0.309	0.339	0.426	0.408	0.432	0.313	0.476	0.362	0.500

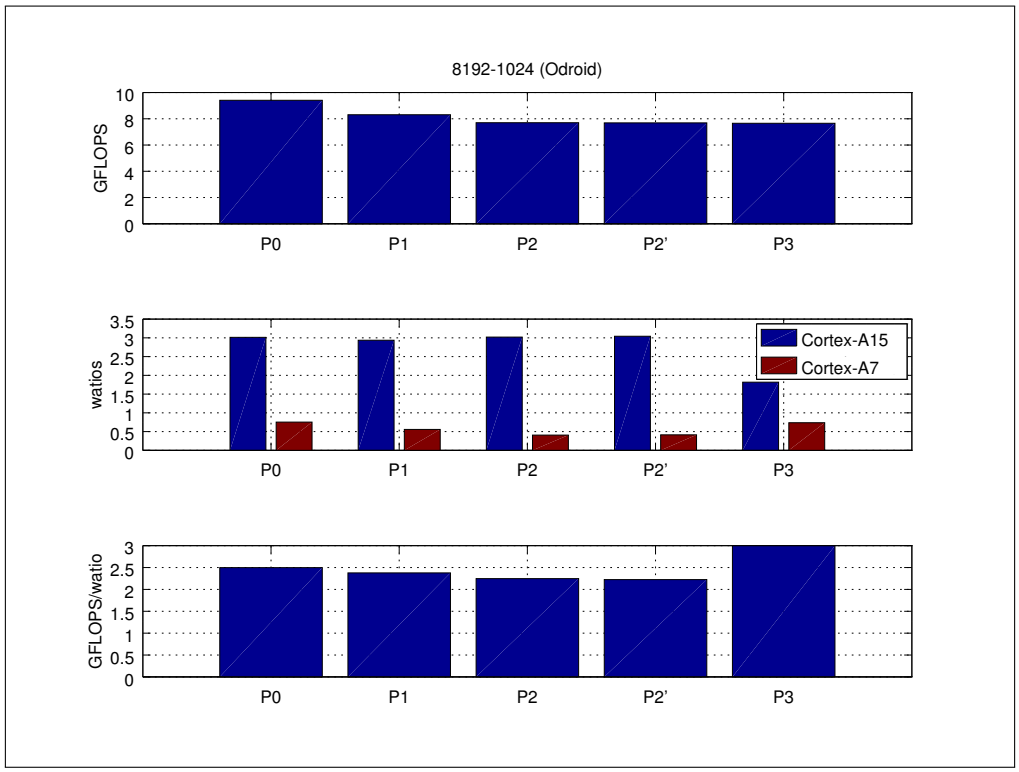
ODROID

bloques de dimensión 1024). Como se puede observar en la figura, la disminución de la potencia media en las políticas P1, P2 y P2' sobre el cluster LITTLE es ínfima frente a la pérdida de rendimiento, seguramente causada por el alto grado de eficiencia energética que posee un procesador LITTLE, provocando que no exista mejora en el rendimiento energético. Sin embargo, la política P3 provoca una gran reducción de la potencia en el cluster big, mayor en proporción que la pérdida de rendimiento, consiguiendo así una mejora en la eficiencia energética muy elevada.

Si se observa con detalle la gráfica, una consecuencia de la política P3 es el aumento de la potencia en el cluster LITTLE. La causa de este aumento es que al disminuir la frecuencia del cluster big, las tareas en ejecución tardan más en finalizar, por lo que existe un mayor número de tareas listas para ser ejecutadas, y mientras que en una ejecución normal existen momentos en los que el cluster de núcleos LITTLE se encuentra ocioso ya que no existen tareas listas, en esta política el cluster sí tiene tareas para ejecutar en todo momento. Sin embargo, a pesar del aumento de potencia en el cluster de núcleos LITTLE, la eficiencia energética sigue siendo mayor frente a una ejecución normal.



(a) Juno



(b) Odroid

Figura 5.8: Detalle de las políticas P1-P3 para la configuración `m=8192` y `b=1024`

Tabla 5.4: Porcentaje de tiempo de ejecución en el que el cluster se encuentra desactivado en función del momento elegido para desactivar.

	Tamaño de la matriz (m) y de bloque (b).													
(m)	1024		2048		4096		4608		5120		6144		8192	
(b)	64	128	128	256	256	512	256	512	512	1024	512	1024	512	1024
50 %	69.36	45.83	39.34	55.83	43.44	50.83	39.39	38.48	40.91	42.07	39.15	35.24	40.32	48.33
40 %	68.20	31.25	29.29	48.75	29.41	33.33	30.61	32.73	32.73	30.54	32.83	30.42	30.15	37.92
30 %	63.42	34.58	20.89	48.33	21.14	32.50	20.88	24.85	25.00	28.06	23.08	25.43	21.81	33.75
20 %	20.40	17.92	11.64	34.17	11.40	31.67	11.97	17.27	18.41	21.04	14.56	19.12	13.11	26.67
10 %	23.10	15.00	4.90	27.92	5.27	20.00	4.87	11.52	9.55	15.13	7.69	10.3	5.64	12.92

Política 4 - Juno

5.3.2. Análisis de resultados para las políticas P4-P6

Políticas P4 y P5

Mientras las políticas anteriores dividían el espacio de frecuencias en partes iguales respecto al tamaño máximo de la cola, las políticas P4-P6 no disponen de un punto claro en la ejecución donde desactivar el cluster correspondiente. Al igual que en las otras políticas, se ha decidido que este punto esté reflejado por la relación entre el tamaño actual de la cola de tareas listas frente al tamaño máximo alcanzado hasta el momento. Los experimentos realizados contemplan desde ejecuciones que desactivan el cluster cuando el número de tareas listas para ejecución es un 50 % del tamaño máximo alcanzado, decreciendo hasta un punto de corte del 10 % respecto al tamaño máximo en intervalos de un 10 % del tamaño máximo. Nótese que este porcentaje no se encuentra relacionado con el tiempo de ejecución, sino con el número de tareas listas en cada momento. A modo informativo, la Tabla 5.4 muestra para cada configuración del tamaño de la matriz y bloque (columnas), y para cada experimento lanzado (filas), el porcentaje del tiempo de ejecución en el que el cluster se encuentra desactivado para la política P4 y la plataforma Juno.

La Figura 5.9 muestra el comportamiento de las políticas P4 y P5 para la plataforma Juno, combinando para cada configuración de tamaño de matriz y bloque los distintos tamaños de cola experimentados. Adicionalmente, se incorporan las medidas realizadas sobre el planificador Botlev para poder comparar los resultados (columna izquierda en cada conjunto de datos).

La primera observación que se puede realizar sobre los datos obtenidos es el impacto muy significativo que posee variar el punto de desactivado del cluster sobre el rendimiento final de la aplicación, tanto desactivando el cluster LITTLE (política P4) como desactivando el cluster big (política P5). La Tabla 5.5 muestra el rendimiento alcanzado por cada configura-

Tabla 5.5: Rendimiento obtenido para las políticas P4 y P5 en ambas plataformas en relación a una ejecución normal con BOTLEV.

	JUNO					ODROID				
	50 %	40 %	30 %	20 %	10 %	50 %	40 %	30 %	20 %	10 %
P4	52.18 %	59.56 %	67.33 %	78.29 %	85.87 %	70.477 %	75.93 %	82.19 %	87.55 %	91.43 %
P5	58.2 %	62.99 %	69.85 %	76.05 %	84.77 %	61.96 %	67.85 %	74.59 %	82.03 %	90.27 %

ción del problema, política y plataforma según el momento elegido para desactivar el cluster correspondiente. Nótese que mientras que en la plataforma JUNO es la política P4 la que obtiene peor rendimiento frente a la política P5, en la plataforma ODROID sucede al contrario para las tres primeras columnas de la tabla. Una de las causas que provoca este hecho se encuentra en el tamaño de cada cluster en cada una de las plataformas. En JUNO, el cluster big se encuentra formado únicamente por dos núcleos, mientras que el cluster LITTLE se encuentra formado por cuatro. Al desactivar el cluster LITTLE, únicamente se dispone de dos núcleos donde ejecutar las tareas, por lo que el grado de paralelismo explotado es menor que si se desactiva el cluster big. Como conclusión adicional, se puede observar como, a pesar de que el rendimiento de un core big (Cortex-A57 en este caso) es más elevado que el rendimiento de un core LITTLE (Cortex-A53), realizar la ejecución sobre cuatro núcleos LITTLE obtiene un rendimiento mayor que realizarla sobre únicamente dos núcleos big.

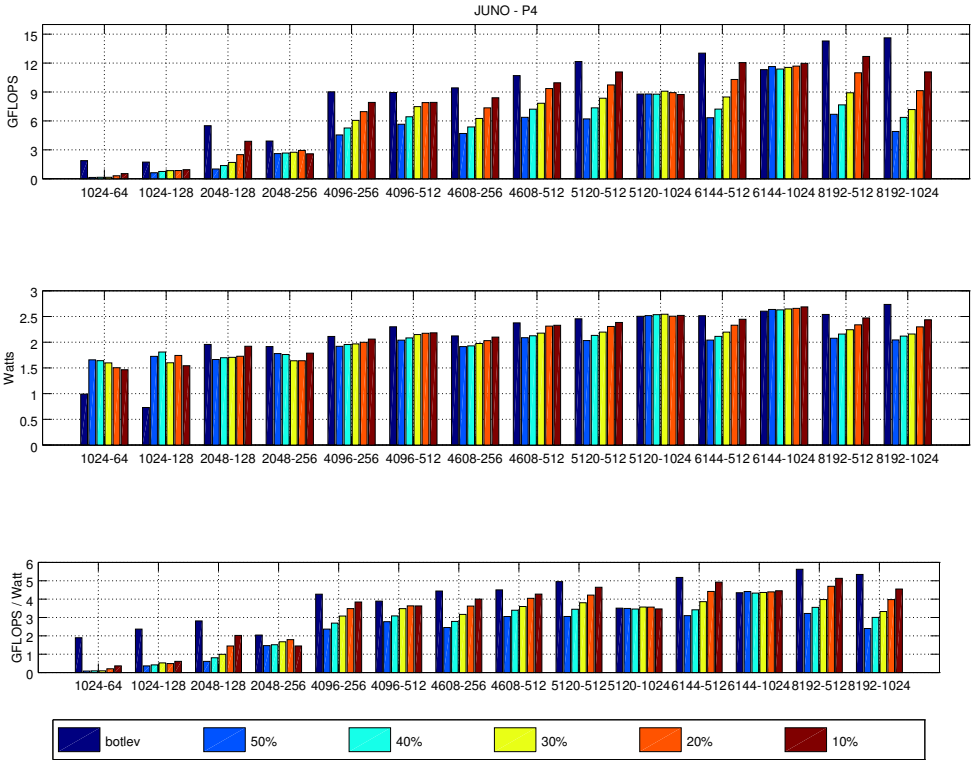
Como se puede apreciar en la figura, tanto la política P4 como la política P5 no consiguen obtener la mejora energética buscada. Mientras que la política P4 obtiene resultados muy inferiores a la ejecución con BOTLEV, la política P5 alcanza resultados cercanos. La Tabla 5.6 muestra con detalle la diferencia de rendimiento energético con la ejecución de referencia con BOTLEV para la política P5 con cada configuración de tamaño.

A la vista de los resultados, y aunque la política P5 no consiga obtener mejoras en el rendimiento energético, señalar que consigue resultados similares a la ejecución convencional reduciendo la potencia instantánea consumida, aunque a costa de reducir el rendimiento computacional. Sin embargo, esta política puede tener su utilidad en problemas donde la ejecución esté limitada por la potencia consumida y no tanto por el rendimiento conseguido, pudiendo alcanzar un compromiso entre rendimiento y potencia sin afectar al rendimiento energético final escogiendo el valor adecuado para determinar cuándo desactivar el cluster.

Política P6

La figura 5.10 compara los resultados obtenidos mediante la política P5 y P6. Como se comentaba en la sección 5.2.2, desactivar un core a nivel del kernel tiene un coste asociado al tener que migrar los procesos a un core activo, hecho que se refleja en que el rendimiento

Política P4:



Política P5:

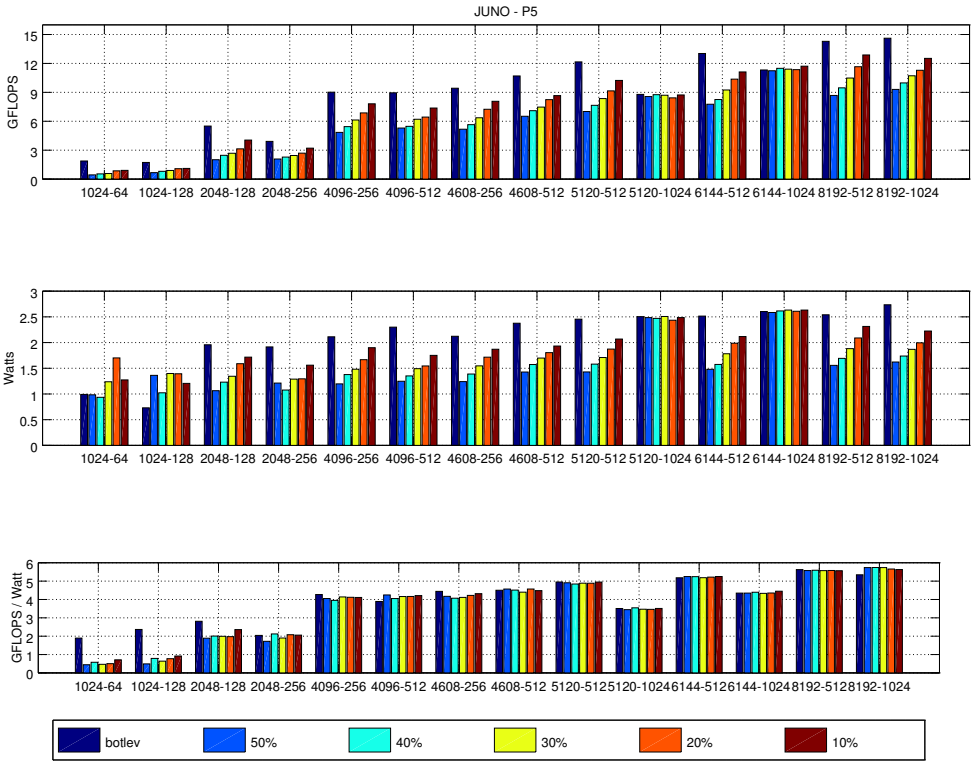


Figura 5.9: Resultados para los experimentos P4 y P5 sobre la plataforma JUNO.

Tabla 5.6: Mejora relativa a la ejecución sobre BOTLEV de la eficiencia energética para la política P5 en ambas plataformas.

	Tamaño de la matriz (m) y de bloque (b).													
(m)	1024		2048		4096		4608		5120		6144		8192	
(b)	64	128	128	256	256	512	256	512	512	1024	512	1024	512	1024
10 %	-3.666	4.066	-2.797	-1.841	-0.523	-0.270	-0.515	-0.370	-0.324	0.045	-0.328	-0.076	-0.440	-0.222
20 %	-3.978	1.981	-2.589	-1.773	-0.303	-0.162	-0.478	-0.234	-0.247	-0.001	-0.269	0.002	-0.283	-0.133
30 %	-4.033	-0.991	-2.248	-1.422	-0.154	-0.120	-0.267	-0.134	-0.122	0.001	-0.152	-0.035	-0.286	-0.025
40 %	-3.693	1.560	-2.451	-1.077	0.055	-0.057	-0.154	-0.008	-0.022	-0.017	-0.057	-0.014	-0.193	0.001
50 %	-4.037	-1.470	-1.052	-0.881	0.156	0.038	0.065	0.013	0.052	-0.005	0.002	-0.033	-0.114	0.080

ODROID

	Tamaño de la matriz (m) y de bloque (b).													
(m)	1024		2048		4096		4608		5120		6144		8192	
(b)	64	128	128	256	256	512	256	512	512	1024	512	1024	512	1024
10 %	-1.451	-1.876	-0.921	-0.322	-0.218	0.356	-0.270	0.065	-0.043	-0.063	0.067	-0.000	-0.051	0.393
20 %	-1.317	-1.578	-0.805	0.079	-0.317	0.159	-0.373	0.009	-0.110	0.037	0.062	0.046	-0.033	0.397
30 %	-1.428	-1.727	-0.818	-0.143	-0.130	0.269	-0.329	-0.104	-0.064	-0.039	0.005	-0.015	-0.055	0.393
40 %	-1.392	-1.593	-0.837	0.037	-0.150	0.273	-0.220	0.066	-0.066	-0.047	0.037	0.005	-0.047	0.315
50 %	-1.184	-1.447	-0.453	0.015	-0.159	0.320	-0.126	-0.022	-0.004	0.003	0.070	0.1 02	-0.061	0.286

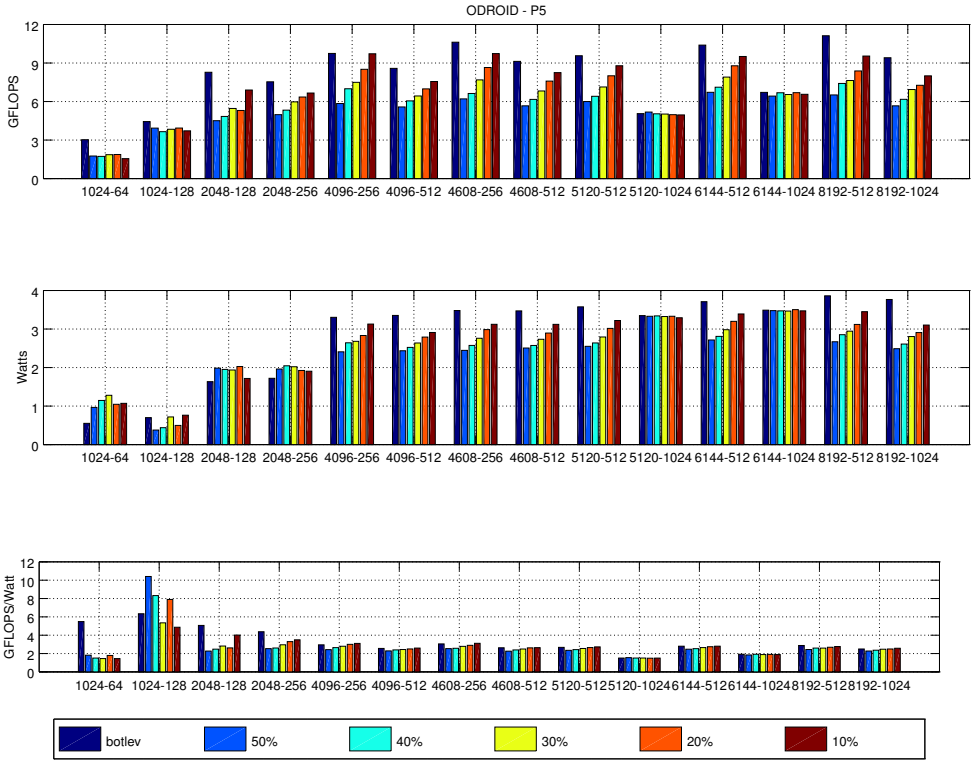
JUNO

Tabla 5.7: Mejora de la eficiencia energética para la política P6.

	Tamaño de la matriz (m) y de bloque (b).													
(m)	1024		2048		4096		4608		5120		6144		8192	
(b)	64	128	128	256	256	512	256	512	512	1024	512	1024	512	1024
10 %	-4.984	-5.028	-3.400	-2.671	-0.160	-0.085	-0.021	0.009	0.015	-0.019	-0.008	0.023	0.239	0.325
20 %	-4.956	-4.707	-3.278	-2.745	0.004	-0.053	0.034	-0.031	0.008	0.006	0.008	-0.024	0.350	0.350
30 %	-4.718	-4.830	-3.490	-2.578	0.124	0.070	0.162	0.036	0.047	0.009	0.052	-0.047	0.127	0.347
40 %	-4.922	-4.442	-3.210	-2.465	0.146	0.023	0.285	0.071	0.086	0.034	0.043	-0.006	0.124	0.408
50 %	-4.893	-3.923	-3.233	-2.291	0.060	0.018	0.144	0.060	0.085	-0.012	0.039	-0.035	0.051	0.367

obtenido en los experimentos para la política P6 sea menor al obtenido para la política P5. Sin embargo, como en la plataforma ODROID desactivar el cluster big supone un consumo cercano a cero, las medidas para la potencia instantánea son muy inferiores a las obtenidas provocando que los núcleos estuvieran *idle* pero sin llegar a desactivarlos. Combinando estos dos resultados se obtiene el rendimiento energético mostrado en la tabla 5.7. Como se puede apreciar, la mejora en el consumo de potencia es superior a la pérdida de rendimiento, consiguiendo ganancias en el rendimiento energético, haciendo a esta política una solución válida para obtener una mejora de rendimiento energético obteniendo reducciones considerables de la potencia disipada durante la ejecución.

Política P5:



Política P6:

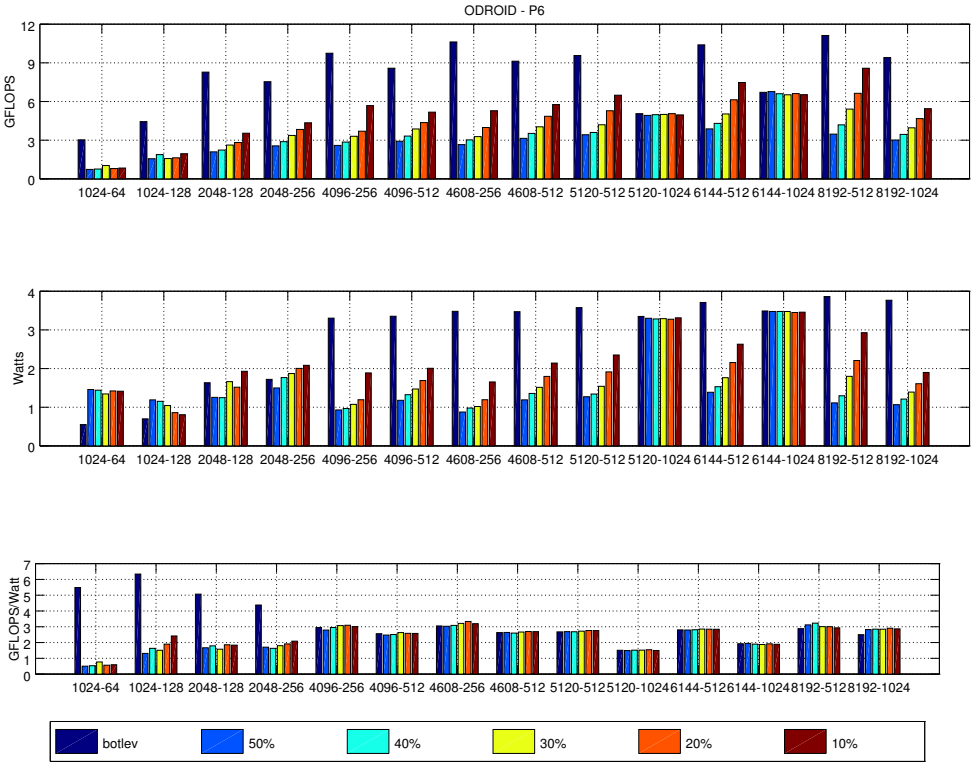


Figura 5.10: Resultados de los experimentos realizados para las políticas P5 y P6 sobre Odroid.

Capítulo 6

Conclusiones

6.1. Conclusiones y trabajo futuro

En este trabajo se han desarrollado un conjunto de técnicas sobre arquitecturas asimétricas, y en especial para los procesadores big.LITTLE diseñados por ARM, que buscan obtener mejoras tanto en el rendimiento de las aplicaciones como en la eficiencia energética de la plataforma.

En base a un paradigma basado en la extracción de paralelismo a nivel de tareas en tiempo de ejecución, y a través de una versión de la biblioteca BLIS adaptada a arquitecturas asimétricas, y basándose en un planificador de tareas convencional, se ha demostrado como es posible obtener un mejor rendimiento sin necesidad de adaptar el código del problema a la arquitectura, o introducir cambios específicos en el planificador. Esta propuesta permite reutilizar los *runtimes* convencionales (y todas las técnicas desarrolladas durante los últimos años) sobre las nuevas arquitecturas asimétricas con el único coste de la implementación de versiones asimétricas de cada rutina de la librería. Los experimentos realizados han demostrado que esta solución es totalmente comparable con un planificador consciente de la asimetría de la plataforma, incluso llegando a obtener mejores resultados en el rendimiento de la aplicación. Esta técnica ha sido presentada en *"The Sixth International Workshop on Accelerators and Hybrid Exascale Systems (AsHES)"*, a través del artículo [12] mostrado en la bibliografía.

Buscando la mejora de la eficiencia energética, se han desarrollado una serie de políticas integradas sobre un planificador consciente de la asimetría, basadas tanto en técnicas de escalado de frecuencia (DVFS) como en técnicas de planificación, tomando las decisiones de forma dinámica en función del número de tareas listas para ser ejecutadas en cada momento. El primer conjunto de experimentos realizados muestra como, independientemente de las dos

plataformas sobre las que se han ejecutado los experimentos, aplicar técnicas de escalado de frecuencia sobre el cluster de núcleos big obtiene una mejora en el rendimiento energético significativa frente a un planificador convencional consciente de la asimetría sin ninguna política de ahorro energético integrada.

De igual manera, el segundo conjunto de experimentos revela como en aquellos procesadores que permiten desactivar un cluster completo a nivel *hardware*, apagarlo en ciertos momentos de la ejecución cuando el nivel de tareas listas es lo suficientemente bajo permite obtener una mejora de rendimiento energético reduciendo la potencia instantánea media consumida drásticamente. También se ha puesto de manifiesto como, en los procesadores que no soportan un apagado del cluster a nivel físico, una política de planificación de tareas que no asigne tareas a ese cluster obtiene los mismos resultados de eficiencia energética que el planificador normal consciente de la asimetría.

Como trabajo futuro, se considera realizar los experimentos sobre un conjunto mayor de problemas de interés científico, así como sobre otras arquitecturas donde el número de núcleos big y LITTLE se encuentre más desequilibrado. Adicionalmente, se considera desarrollar políticas que tomen decisiones de manera dinámica en base al problema ejecutado y su DAG correspondiente, así como a resultados parciales tomados durante la ejecución de las distintas tareas previas del problema.

Conclusions

Conclusions and future work

Throughout this work, we have presented a set of newly developed techniques over asymmetric architectures, focusing on ARM designed big.LITTLE systems-on-chip, which are aimed at obtaining performance and energy efficiency improvements in parallel applications.

We have demonstrated that, for a task-level parallelism model, an approach that delegates the burden of dealing with asymmetry to the library (in our case, using an asymmetry-aware BLIS implementation), does not require any reformulation of an existing task scheduler, and can deliver high performance. This proposal paves the road towards reusing conventional runtime schedulers for SMPs (and all the associated improvement techniques developed through the past few years), as the runtime only has a symmetric view of the hardware, at the cost of developing asymmetry-aware underlying libraries. Our experiments reveal that this solution is competitive and even improves the results obtained with an asymmetry-aware scheduler for DLA operations. This work has been presented in *"The Sixth International Workshop on Accelerators and Hybrid Exascale Systems (AsHES)"*, with the paper [12] shown in the bibliography.

Trying to improve energy efficiency, a set of new policies has been integrated into the asymmetry-aware scheduler, based both on dynamic voltage frequency scaling (DVFS) techniques and scheduling techniques, making decisions dynamically depending on the number of ready tasks to be executed every time. The first set of experiments shows how, independently of the platform used, applying techniques of frequency scaling over big cores leads to better results than an asymmetry-aware scheduler.

Similarly, the second set of experiments shows how, in processors which allow deactivating a full cluster at *hardware* level, switching it off at certain times during the execution when the number of ready tasks is low enough allows to obtain an energy performance gain by drastically reducing the average instant power consumed. Also, these experiments reveal that, in architectures that do not support a *hardware* switching off, a policy which does not assign tasks to a certain cluster has the same energy performance results that an asymmetric-aware scheduler has, decreasing the average power consumed.

As future work, we are considering to carry out the experiments over a bigger set of problems of scientific interest, and to run them over other platforms where the number of big and LITTLE cores is less balanced. Additionally, we consider to develop new techniques that make decisions dynamically depending on the problem we are trying to solve and its associated DAG, as well as partial results taken during previous task executions.

Bibliografía

[1] M. Abalenkovs, A. Abdelfattah, J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, and A. YarKhan. Parallel programming models for dense linear algebra on heterogeneous systems. Vol. 2, No. 4:pp. 67–86, 2015.

[2] E. Agullo, O. Beaumont, L. Eyraud-Dubois, J. Herrmann, S. Kumar, L. Marchal, and S. Thibault. Bridging the Gap between Performance and Bounds of Cholesky Factorization on Heterogeneous Platforms. In *Heterogeneity in Computing Workshop 2015*, Hyderabad, India, May 2015.

[3] E. Agullo, O. Beaumont, L. Eyraud-Dubois, and S. Kumar. Are Static Schedules so Bad ? A Case Study on Cholesky Factorization. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS 2016)*, Chicago, IL, United States, May 2016. IEEE.

[4] P. Alonso, R. M. Badia, J. Labarta, M. Barreda, M. F. Dolz, R. Mayo, E. S. Quintana-Ortí, and R. Reyes. Tools for power-energy modelling and analysis of parallel scientific applications. In *41st Int. Conf. on Parallel Processing – ICPP*, pages 420–429, 2012.

[5] AMD. AMD Core Math Library. `http://developer.amd.com/tools/cpu/acml/pages/default.aspx`, 2012.

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[7] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience*, 21(18):2438–2456, 2009.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[9] S. Catalán, F. D. Igual, R. Mayo, R. Rodríguez-Sánchez, and E. S. Quintana-Ortí. Architecture-aware configuration and scheduling of matrix multiplication on asymmetric multicore processors. *Cluster Computing*, pages 1–15, 2016.

[10] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero. Criticality-aware dynamic task scheduling for heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 329–338, New York, NY, USA, 2015. ACM.

[11] Cilk project home page. `http://supertech.csail.mit.edu/cilk/`. Last visit: July 2015.

[12] L. Costero, F. D. Igual, S. Catalán, K. Olcoz, R. Rodríguez-Sánchez, and E. Quintana-Ortí. Refactoring conventional task schedulers to exploit asymmetric ARM big.LITTLE architectures in dense linear algebra. *AsHES, The Sixth International Workshop on Accelerators and Hybrid Exascale Systems*, 2016.

[13] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[14] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[15] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[16] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[17] Extrae. `https://www.bsc.es/computer-sciences/extrae`. Last visit: August 2016.

[18] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[19] S. H. Fuller and L. I. Millett (Editors). *The future of computing performance: game over or next level?* National Research Council of the National Academies, 2011.

[20] T. Gautier, X. Besseron, and L. Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, pages 15–23, New York, NY, USA, 2007. ACM.

[21] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. XKaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Proc. IEEE 27th Int. Symp. on Parallel and Distributed Processing*, IPDPS'13, pages 1299–1308, 2013.

[22] G. H. Golub and C. F. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.

[23] K. Goto and R. A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):12:1–12:25, May 2008.

[24] A. Haidar, C. Cao, A. Yarkhan, P. Luszczek, S. Tomov, K. Kabir, and J. Dongarra. Unified development for mixed multi-gpu and multi-coprocessor environments using a lightweight runtime environment. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 491–500, May 2014.

[25] IBM. Engineering and Scientific Subroutine Library. `http://www.ibm.com/systems/software/essl/`, 2012.

[26] Linaro IKS: The big.LITTLE in-kernel switcher. `https://wiki.linaro.org/projects/big.LITTLE.MP/Big.Little.Switcher/Docs/in-kernel-code`. Last visit: August 2016.

[27] Intel Corp. Intel math kernel library (MKL) 11.0. `http://software.intel.com/en-us/intel-mkl`, 2014.

[28] Kaapi project home page. `https://gforge.inria.fr/projects/kaapi`. Last visit: July 2015.

[29] P. Kogge and J. Shalf. Exascale computing trends: Adjusting to the "new normal" for computer architecture. *Computing in science and engineering*, 15(6):16–26, Nov 2013.

[30] LAPACK project home page. `http://www.netlib.org/lapack`.

[31] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, Sept. 1979.

[32] C. E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM.

[33] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Ortí. Analytical modeling is enough for high performance BLIS. *ACM Trans. Math. Soft.*, 43, Aug 2016. Available at `http://www.cs.utexas.edu/users/flame`.

[34] E. Anderson et al. *LAPACK Users' guide*. SIAM, 3rd edition, 1999.

[35] Mercurium. `https://pm.bsc.es/mcxx`. Last visit: August 2016.

[36] S. Mittal. A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Comput. Surv.*, 48(3):45:1–45:38, Feb. 2016.

[37] Nanos++. `https://pm.bsc.es/nanox`. Last visit: August 2016.

[38] OmpSs project home page. `http://pm.bsc.es/ompss`. Last visit: Aug. 2016.

[39] OpenBLAS. `http://xianyi.github.com/OpenBLAS/`. Last visit: July 2015.

[40] M. Oskin and J. Torrellas. Laying a new foundation for it: Computer architecture for 2025 and beyond. *Workshop on Advancing computer architecture research (ACAR-II)*, Sept 2010.

[41] Paraver: the flexible analysis tool. `http://www.bsc.es/computer-sciences/performance-tools/paraver`. Last visit: August 2016.

[42] QUARK project home page. `http://icl.cs.utk.edu/quark`. Last visit: August 2016.

[43] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, and E. Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software*, 36(3):14:1–14:26, 2009.

[44] N. Rajovic, A. Rico, F. Mantovani, D. Ruiz, J. Vilarrubi, C. Gomez, D. Nieto, H. Servat, X. Martorell, J. Labarta, C. Adeniyi-Jones, S. Derradji, H. Gloaguen, P. Lanucara, N. Sanna, J.-F. Méhaut, K. Pouget, B. Videau, E. Boyer, M. Allalen, A. Auweter,

D. Brayford, D. Tafani, V. Weinberg, D. Brömmel, R. Halver, J. Meinke, R. Beivide, M. Benito, E. Vallejo, M. Valero, and A. Ramirez. The Mont-Blanc prototype: An Alternative Approach for HPC Systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Salt Lake City, United States, Nov. 2016.

[45] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? *Communications of ACM*, 58(5):77–86, Apr 2015.

[46] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, pages 1–25, Berlin, Heidelberg, 2011. Springer-Verlag.

[47] T. M. Smith, R. van de Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee. Anatomy of high-performance many-threaded matrix multiplication. In *Proc. IEEE 28th Int. Symp. on Parallel and Distributed Processing*, IPDPS'14, pages 1049–1059, 2014.

[48] StarPU project home page. `http://runtime.bordeaux.inria.fr/StarPU/`. Last visit: July 2015.

[49] Superglue project home page. `http://www.it.uu.se/research/scientific_computing/software/superglue`. Last visit: August 2016.

[50] M. Tillenius. Superglue: A shared memory framework using data versioning for dependency-aware task-based parallelization. *SIAM Journal on Scientific Computing*, 37(6):C617–C642, 2015.

[51] M. Tillenius, E. Larsson, R. M. Badia, and X. Martorell. Resource-aware task scheduling. *ACM Trans. Embed. Comput. Syst.*, 14(1):5:1–5:25, Jan. 2015.

[52] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. IEEE Symp. on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010.

[53] M. Valero, M. Moreto, M. Casas, E. Ayguade, and J. Labarta. Runtime-aware architectures: A first approach. *International Journal of Supercomputing Frontiers and Innovations*, 1(1), 2014.

[54] F. G. Van Zee, T. M. Smith, B. Marker, T. M. Low, R. A. van de Geijn, F. D. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel, J. Gunnels, and L. Killough. The BLIS framework: Experiments in portability. *ACM Trans. Math. Soft.*, (2):12:1–12:19, Jun 2016. Available at `http://www.cs.utexas.edu/users/flame`.

[55] F. G. Van Zee and R. A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33, 2015.

[56] A. YarKhan, J. Kurzak, and J. Dongarra. Quark users' guide: Queueing and runtime for kernels. Technical report, Innovative Computing Laboratory, University of Tennessee, 2011.

[57] F. G. V. Zee. `libflame`*: The Complete Reference*. `www.lulu.com`, 2009.

–¿Qué te parece desto, Sancho? – Dijo Don Quijote –
Bien podrán los encantadores quitarme la ventura,
pero el esfuerzo y el ánimo, será imposible.

Segunda parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes