Herramienta de monitorización de rendimiento y consumo energético basado en Odroid SmartPower2

TRABAJO DE FIN DE GRADO

Andrés Plaza Hernando y Germán Franco Dorca

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería Informática

Facultad de Informática

Universidad Complutense de Madrid

2019

Herramienta de monitorización de rendimiento y consumo energético basado en Odroid SmartPower2

Memoria de Trabajo de Fin de Grado

Andrés Plaza Hernando y Germán Franco Dorca

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería Informática

Facultad de Informática

Universidad Complutense de Madrid

2019

Agradecimientos

Quisiéramos agradecer a Juan Carlos todo el apoyo y ayuda que nos ha prestado durante el transcurso del trabajo, y sobretodo la motivación extra que nos ha dado en la fase final del proyecto. Ha sido sin duda gracias a él que se ha podido realizar y finalizar este TFG. También agradecer a la Universidad Complutense de Madrid todo el trabajo que nos ha permitido tener los conocimientos para realizar este Trabajo de Fin de Grado.

Resumen

Odroid SmartPower2 es una fuente de alimentación que permite monitorizar externamente el consumo de potencia y otros aspectos relacionados del dispositivo que alimenta. El firmware proporcionado por el fabricante del Odroid SmartPower2 permite extraer información de monitorización por WiFi, telnet o puerto serie.

En este proyecto se ha desarrollado el soporte necesario para realizar mediciones con este dispositivo desde PMCTrack. Esta herramienta de software libre para Linux permite acceder cómodamente a los contadores hardware de monitorización del rendimiento en procesadores actuales de Intel, AMD y ARM. Estos contadores soportan la captura de métricas de rendimiento como el número de instrucciones por ciclo o la tasa de fallos de cache para cualquier aplicación en ejecución. Gracias a la abstracción de módulos de monitorización (*plugins*) de PMCtrack, es posible exponer al usuario -junto con las métricas de rendimiento- otro tipo de información relevante de monitorización acerca del hardware, como el consumo energético o el espacio usado por una aplicación en una caché compartida. El desarrollo del soporte necesario para la medición con Odroid SmartPower2 desde PMCTrack ha permitido extraer información tanto de rendimiento como de consumo.

Como caso de uso del soporte de medida de consumo desarrollado con PMCTrack, en este trabajo también se ha realizado una caracterización del rendimiento y la eficiencia energética de los benchmarks de la *suite* SPEC CPU en los distintos tipos de cores de un procesador big.LITTLE de ARM, arquitectura multicore asimétrica que actualmente se encuentra presente en un amplio espectro de dispositivos móviles.

Palabras clave — Procesadores multicore asimétricos, Odroid SmartPower2, Odroid XU4, PMCTrack, Eficiencia Energética, Kernel Linux.

Abstract

Odroid SmartPower2 is a power supply board that can externally monitor energy consumption and other aspects from the device on which it feeds. The firmware provided by the manufacturer of Odroid SmartPower2 can makes possible outputting monitoring data through WiFi, telnet or serial port.

We have developed for this project the required support to carry out measurements using this device from PMCTrack. This free software tool allows to easily access the hardware monitoring counters of current Intel, AMD, and ARM processors. These counters provide the means for gathering performance metrics such as the number of instructions per cycle or the rate of cache misses of any running application. Thanks to the abstraction of PMCTrack's monitoring modules (plugins), it is possible to obtain -along performance metrics- other relevant monitoring information concerning hardware, like energy consumption or the amount of space used by an application in a shared cache. Our development of the required support to carry out measurements using Odroid SmartPower2 from PMCTrack has enabled the ability to extract both performance and energy consumption information at once.

As a use case of the developed support for PMCTrack, it's been done a characterization of the performance and the energy efficiency of the SPEC CPU suite benchmarks for each core type of a big.LITTLE ARM processor, an asymmetric multicore architecture that is present in a wide range of mobile devices.

Keywords — Asymmetric Multicore Proccessors, Odroid XU4, Odroid SmartPower2, PMCTrack, Energy Efficiency, Linux Kernel.

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

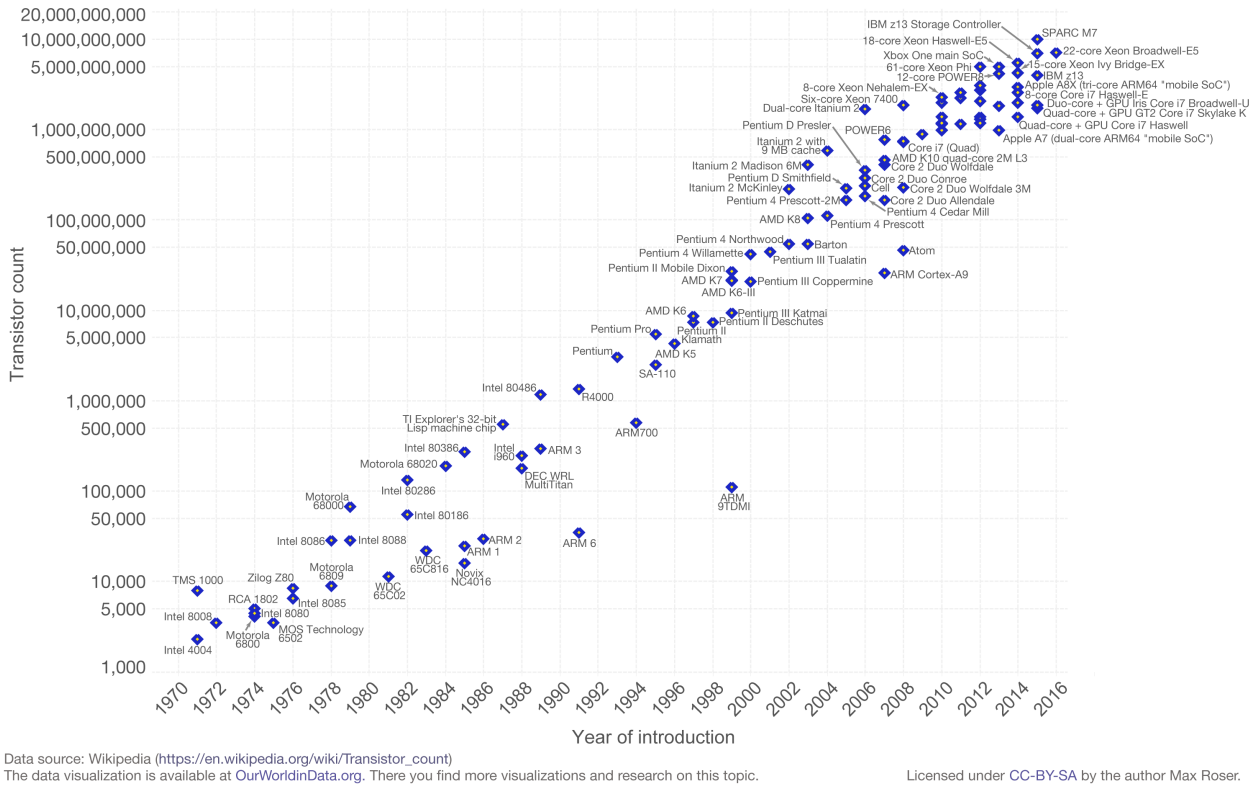
Introducción

La eficiencia energética se ha convertido no sólo en un gran dilema de la época actual, sino también en un reto para los fabricantes de hardware. La eficiencia energética de un sistema o un dispositivo frente a otro, es la diferencia de sus consumos al encontrarse en el mismo escenario de ejecución. Es decir un sistema es más eficiente energéticamente que otro cuando, por ejemplo al ejecutar la misma aplicación, el primero consume menos energía que el segundo.

En la actualidad un sistema eficiente consigue tres metas: proteger el medio ambiente, ser más económico (no en coste de compra, pero sí en su mantenimiento) y en el caso de dispositivos móviles, mayor autonomía de los mismos. Esto último se puede apreciar no solo en móviles y tablets, sino también en dispositivos como los coches eléctricos.

La tecnología ha seguido una evolución exponencial en las últimas décadas debido a la búsqueda de sistemas cada vez más potentes. Para conseguirlo los fabricantes siempre habían optado por incrementar la frecuencia de los procesadores y disminuir su tamaño. Además desde su propuesta original en 1965, este incremento se ha hecho en base a la ley de Moore (figura A.1), en la que se pronosticaba que el número de transistores de un microprocesador se duplicaría cada dos años.

Esta evolución se ha encontrado con diversos problemas a lo largo de su historia. Uno de los más importantes, el incremento cada vez mayor del consumo energético de los procesadores y problemas con la disipación térmica de los mismos. Esta tendencia sugería que el incremento de frecuencia de los procesadores con el paso del tiempo se encontraría con una barrera difícil de superar. Esto ha obligado a los fabricantes a buscar nuevas técnicas de desarrollo que permitieran mejorar el rendimiento de los procesadores sin incrementar la frecuencia. De aquí surgió la aproximación que se usa a día hoy, la explotación del paralelismo entre procesadores, para seguir aprovechando la continuidad de la ley de Moore aumentando el número de transistores, sin incrementar la frecuencia de los procesadores. El paralelismo entre distintos procesadores consiste en incluir más elementos de procesamiento, también llamados *cores*, en un mismo microchip. Esta mejora permite explotar el paralelismo a nivel de hilo (*Thread Level Parallelism*). A estos procesadores que integran más de un núcleo de procesamiento (*core*) se denominan procesadores multicore.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Figura 1.1: Diagrama de la ley de Moore [1].

El siguiente paso a los procesadores multicore convencionales o simétricos (SMP) fueron los procesadores multicore asimétricos o AMP (*Asymmetric Multicore Processors*) [2]. A diferencia de los primeros, en los que todos los *cores* eran iguales, los sistemas AMP combinan *cores* de alto rendimiento con *cores* de bajo consumo energético y arquitectónicamente más sencillos. A pesar de ser diferentes, todos los *cores* de un AMP poseen un repertorio de instrucciones común para no complicar el desarrollo de software. La ventaja de los AMP sobre los SMP es que es posible mejorar sustancialmente la eficiencia energética del sistema, sacrificando un poco de rendimiento en proporción [2].

Los *cores* de alto rendimiento están diseñados para ejecutar cargas de trabajo más rápido mediante una arquitectura más compleja a costa de un mayor consumo de energía y producción de calor. Al contrario, los *cores* de bajo consumo tienen una arquitectura más simple y un rendimiento menor, por lo que su consumo energético es más reducido. Por esta razón, los *cores* de bajo consumo son ideales para ejecutar cargas de trabajo en las que hay mayor número de paradas del *pipeline*, como en aquellas más intensivas de memoria, ya que consumen menos mientras esperan; a diferencia de otras cargas de trabajo que hacen mayor uso del procesador, y por tanto optimizan mejor los recursos y el consumo extra de energía. El hardware por sí solo no es consciente de estos beneficios, por lo que necesita de la ayuda del sistema operativo para hacer el reparto de tareas. A esta distinción se le denomina *especialización de ILP* y se han propuesto algoritmos de planificación que son capaces de mejorar el rendimiento explotando estos beneficios [3].

Los grupos *cores* con las mismas características se denominan *clusters*. Normalmente suele haber dos *clusters*. Un ejemplo de la organización de los *cores* de un AMP y la memoria caché compartida se puede ver en la figura A.2. Se trata del modelo de *heterogeneous multi-processing* en el que todos los *cores* pueden ser utilizados de forma simultánea. Es el modelo más potente ya que permite la ejecución de cargas de trabajo en distintos *cores* dependiendo de sus necesidades en cada momento. También cabe mencionar que hay otros dos modelos de procesamiento: *clustered switching*, que es el más simple, en el que sólo uno de los *clusters* está activo según las necesidades globales del sistema; y el modelo *in-Kernel switching*, en el que los *cores* se agrupan en parejas de cores *big* y *LITTLE* y sólo uno de los dos está activo según las necesidades de la carga de trabajo que ejecutan. Estos últimos son más restrictivos que el modelo *heterogeneous multi-processing*, lo que permite menos optimizaciones a nivel de software.

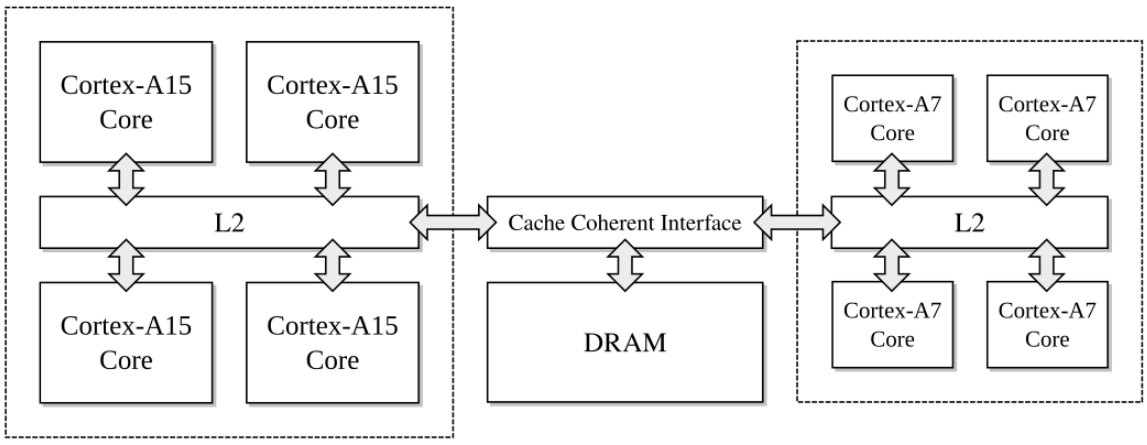


Figura 1.2: Ejemplo de la organización de los cores y la caché en un AMP.

ARM implementa un sistema AMP con su arquitectura big.LITTLE. Actualmente hay en el mercado una gran variedad de procesadores con esta arquitectura, como es el caso del procesador de la placa de desarrollo ARM Juno. Esta placa cuenta con un procesador dividido en dos *clusters*: el *cluster big* contiene dos *cores* ARM Cortex-A57 y el *cluster LITTLE* contiene cuatro *cores* de bajo consumo ARM Cortex-A53. Además esta placa integra hardware específico que le permite obtener medidas de consumo energético directamente desde el sistema operativo.

Frente al elevado coste de ARM Juno (alrededor de unos 8000$), cabe destacar la existencia de otros dispositivos más económicos que también incorporan la arquitectura AMP. Es el caso de Odroid XU4 (50$ en la web del fabricante), que sin embargo carece de hardware integrado para medir su consumo energético. Por este motivo nos hemos propuesto suplir esta falta, y desarrollar un método para medir el consumo de manera externa utilizando el monitor de consumo Odroid SmartPower2 del mismo fabricante. Las características de ambos componentes se detallan en el capítulo 3.

1.1 Objetivos

El principal objetivo de este Trabajo de Fin de Grado es dar acceso al dispositivo Odroid XU4 a las medidas de su consumo energético en tiempo real para que los algoritmos de planificación de procesos (existentes o futuros) puedan hacer uso de esta información.

Posteriormente, como caso de uso de la implementación, se han hecho pruebas experimentales para una caracterización del rendimiento y eficiencia energética de los *benchmarks* SPEC CPU 2000 y 2006.

Para ello necesitamos una herramienta que nos permita obtener las mediciones desde una fuente externa (Odroid SmartPower) al ejecutar distintas aplicaciones en cada uno de los tipos de core de nuestra plataforma. Optamos por la herramienta de monitorización PMCTrack porque nos permite integrar los valores de consumo junto con otras medidas de rendimiento que ya provee mediante contadores hardware del procesador, ya que así es más fácil compararlas al estar todo integrado en la misma herramienta. Además, PMCTrack expone las medidas de consumo tanto al usuario como al sistema operativo.

En resumen, el objetivo TFG es exponer las medidas de consumo tomadas desde Odroid SmartPower2 al sistema operativo a través de PMCTrack, y de forma adicional, hacer mediciones experimentales en Odroid XU4 para posteriormente analizar los resultados.

1.2 Plan de trabajo

Al comenzar el curso 2018-2019 se realizó una reunión inicial en la que se discutieron las líneas generales del Trabajo de Fin de Grado, cuáles iban a ser los objetivos del mismo, enumerados en la sección anterior y cuáles serían los pasos a seguir para poder elaborar el trabajo de forma progresiva y ordenada. También se establecieron varios métodos de comunicación así como repositorios de trabajo donde poder almacenar y consultar documentación y se creó un proyecto en la aplicación Trello [4] para poder hacer un seguimiento de los avances realizados individual y conjuntamente.

Finalmente con la ayuda del tutor se hizo una división inicial de las tareas del proyecto para empezar el trabajo y realizar la primera parte de forma paralela entre los distintos integrantes del equipo de trabajo. Las tareas de la fase final del trabajo se realizaron de manera conjunta debido a su mayor complejidad.

Para las partes de desarrollo en cada una de las placas, tuvimos que repartirnos el tiempo que las tenía cada uno para poder trabajar sobre ellas en cada una de las partes. Una vez el software era estable (no había alta probabilidad de que se apagara), dejamos el sistema completo montado y accesible por red para poder continuar con los experimentos sin tener que desplazarnos.

La planificación establecida contempló las siguientes tareas:

1. Investigación sobre la herramienta PMCTrack ya que es el núcleo del proyecto.
2. Análisis detallado del firmware de la placa Odroid SmartPower2.

3. Modificaciones del firmware de la placa Odroid SmarPower2 para adaptarlo a las necesidades del proyecto.
4. Desarrollo de un driver para Odroid SmartPower2 para utilizarlo con la herramienta PMCTrack.
5. Ejecución de los *benchmarks* de SPEC CPU 2000 y 2006 con PMCTrack.
6. Tratar los datos obtenidos para obtener gráficas y métricas.
7. Análisis de los resultados utilizando dichas métricas para caracterizar el rendimiento y eficiencia energética de los *benchmarks*.

1.3 Estructura de la memoria

La memoria del proyecto se organiza en los siguientes capítulos:

- El **capítulo 2** presenta la herramienta PMCTrack, su funcionamiento, y sus características. Se le ha dedicado un capítulo entero ya que más tarde será necesario comprender ciertos aspectos de la misma.
- El **capítulo 3** explica el funcionamiento a grandes rasgos de las herramientas utilizadas para la obtención de las métricas y cómo trabajan juntas, así como la implementación del software necesario para hacerlo.
- El **capítulo 4** describe los métodos y técnicas utilizados para la extracción de las métricas de consumo al ejecutar los *benchmarks*.
- El **capítulo 5** muestra los resultados obtenidos de los experimentos realizados sobre el entorno y el análisis realizado en base a estos datos.
- El **capítulo 6** expone las conclusiones obtenidas de este Trabajo de Fin de Grado. También se habla de posibles aplicaciones y avances a realizar como trabajo futuro.
- Por último se incluyen la introducción y las conclusiones del Trabajo de Fin de Grado, ambas traducidas al Inglés. Después las contribuciones de cada alumno y finalmente la bibliografía.

Capítulo 2

Herramienta PMCTrack

En este capítulo se presenta la herramienta de monitorización PMCTrack. En la sección 2.1 se describe cada uno de los componentes software que conforman esta compleja herramienta y su funcionamiento. En la sección 2.2 se detalla el mecanismo de extensiones de PMCTrack, que ha sido esencial para llevar a cabo este TFG.

2.1 Introducción a PMCTrack

PMCTrack es una herramienta de software libre para Linux diseñada para la monitorización de. rendimiento de las aplicaciones mediante contadores hardware (Performance Monitoring Counters o PMCs) [5], [6]. Los contadores hardware son un conjunto de registros y lógica de control asociada que incorporan los procesadores y que permiten llevar la cuenta en tiempo real de ciertos eventos hardware de bajo nivel, como por ejemplo el número de instrucciones retiradas, los accesos y fallos a memoria caché, el número de ciclos transcurridos o los fallos de TLB. Con PMCTrack es posible exponer estas métricas al usuario y al sistema operativo de forma independiente de la arquitectura hardware subyacente. Estas métricas se pueden utilizar para medir de forma sencilla el rendimiento global del sistema en tiempo real, e incluso es posible monitorizar el comportamiento de cada aplicación de forma individual.

La arquitectura de PMCTrack está dividida en tres partes como ilustra la figura 2.1: (1) una serie de APIs e interfaces que residen dentro del propio kernel Linux, (2) un módulo del kernel y (3) herramientas del espacio de usuario.

PMCTrack opera en modo kernel debido a que los contadores hardware no son accesibles desde modo usuario, aunque también se puede utilizar desde modo usuario como veremos más adelante. Para su instalación es necesario compilar el kernel tras aplicar un pequeño parche. Este incluye la API del kernel, que es la que permite la comunicación del sistema operativo con el módulo del kernel de PMCTrack.

La funcionalidad principal de PMCTrack se aloja en un único módulo cargable del kernel. Este módulo contiene una capa de abstracción que accede a los contadores hardware presentes en el procesador con conocimiento específico de cada arquitectura. La parte común de estos

módulos es el núcleo de PMCTrack y es independiente de la arquitectura, lo que permite la comunicación con el exterior. Además desde este módulo se exponen una serie de ficheros en el pseudo sistema de ficheros `/proc` para comunicarse con el modo usuario. Leyendo y escribiendo sobre estos ficheros podemos interactuar con la implementación subyacente, como mostraremos en ejemplos más adelante. La funcionalidad del módulo del kernel de PMCTrack puede extenderse mediante extensiones, que se presentan en la siguiente sección de este capítulo.

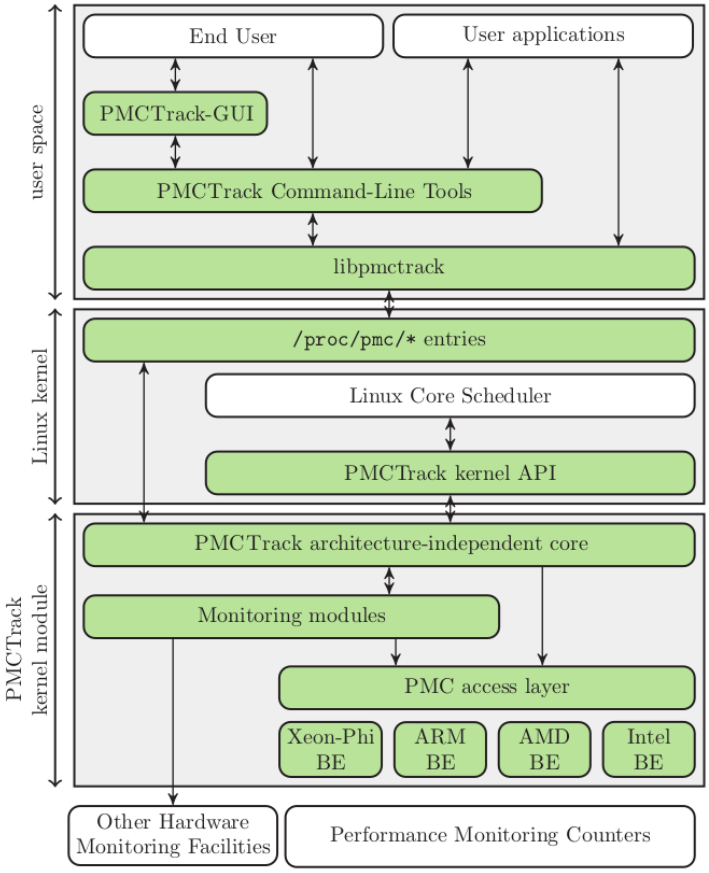


Figura 2.1: Estructura de PMCTrack. Figura extraída de [6].

A nivel de usuario disponemos de `libpmctrack`, una biblioteca desarrollada en C que nos permite acceder a la información de los contadores de forma programática. Mediante funciones para iniciar y parar la monitorización de los contadores podemos monitorizar cualquier fragmento de código de un programa escrito en C o C++. Es posible indicar las métricas que se quieren monitorizar (siempre que estén disponibles para la plataforma) sin tener que lidiar con las particularidades del hardware de cada sistema. Esta librería se comunica con el núcleo de PMCTrack mediante los ficheros virtuales en el directorio `/proc` que mencionamos anteriormente.

El usuario también dispone de una herramienta de línea de comandos de uso sencillo que permite monitorizar aplicaciones. Esta herramienta hace uso de la biblioteca `libpmctrack`. Con el comando `pmctrack` podemos configurar mediante parámetros por ejemplo los datos que queremos obtener, la frecuencia de muestreo o los *cores* en los que se va a ejecutar una

aplicación para su monitorización mediante el uso de parámetros.

Veamos el siguiente ejemplo de uso de esta herramienta:

```
$ pmctrack -T 0.5 -c instr,llc_misses ./gcc06
[Event-to-counter mappings]
pmc1=instr
pmc2=llc_misses
[Event counts]
nsample    pid      event          pmc1          pmc2
      1  20312       tick      55195073         64086
      2  20312       tick      86681788         46936
      3  20312       tick     105806928         98742
      4  20312       tick      56357883        109429
      5  20312       tick      56316043        109792
      6  20312       tick      56360919        109618
      7  20312       tick      57153090        110997
      ...
```

Lo primero que muestra la salida del programa en la sección `[Event-to-counter mappings]` es la información de las métricas monitorizadas y los contadores que se han utilizado para cada métrica. A continuación, en la sección `[Event counts]` se muestran los valores de cada contador en cada columna según se van obteniendo las muestras. Cada fila contiene los valores de una sola muestra.

Con la opción `-T` indicamos la frecuencia de muestreo, en este caso medio segundo. Por defecto PMCTrack tiene una frecuencia de un segundo (*time-based sampling*), pero también se puede configurar para mostrar datos cada vez que un evento hardware alcanza determinado valor, lo que se denomina *event-based sampling*. Con la opción `-c` indicamos que queremos monitorizar el número de instrucciones y los fallos de caché mediante el uso de mnemotécnicos para indicar los eventos. Por último indicamos el programa que se va a monitorizar y sus argumentos si los tuviera.

Asimismo cabe destacar que PMCTrack cuenta también con una herramienta con interfaz gráfica llamada PMCTrack-GUI. Mediante esta interfaz podemos hacer las mismas configuraciones que con la herramienta de línea de comandos de forma más visual. Además es capaz de construir gráficas de los datos obtenidos en tiempo real y mostrarlas por pantalla.

En este TFG haremos uso de la herramienta de línea de comandos `pmctrack` para acceder a los datos de consumo del dispositivo Odroid SmartPower2 y analizarlos posteriormente. No obstante, con las modificaciones que hemos realizado en la herramienta, la información de consumo también es accesible desde PMCTrack-GUI.

2.2 Módulos de monitorización

PMCTrack cuenta con un sistema de extensiones o *plugins* denominados **módulos de monitorización**, que permiten expandir su funcionalidad mediante contadores virtuales. Los

contadores virtuales son una abstracción de los contadores hardware que podemos crear utilizando la API de monitorización de PMCTrack, y nos permiten exponer al usuario y al sistema operativo información de monitorización que no está accesible mediante la interfaz de los contadores hardware del procesador o PMU (Performance Monitoring Unit). El consumo de potencia y energía es un claro ejemplo de información de monitorización que no se expone mediante la PMU. PMCTrack implementa contadores virtuales para ofrecer esta información al usuario.

```
typedef struct {
    char info[MAX_CHARS_EST_MOD];
    struct list_head    links;
    int     (*probe_module)(void);
    int     (*enable_module)(void);
    void    (*disable_module)(void);
    int     (*on_read_config)(char* s, unsigned int maxchars);
    int     (*on_write_config)(const char *s, unsigned int len);
    int     (*on_fork)(unsigned long clone_flags, pmon_prof_t*);
    void    (*on_exec)(pmon_prof_t*);
    int     (*on_new_sample)(pmon_prof_t* prof, int cpu, pmc_sample_t* sample,
                             int flags, void* data);
    void    (*on_migrate)(pmon_prof_t* p, int prev_cpu, int new_cpu);
    void    (*on_exit)(pmon_prof_t* p);
    void    (*on_free_task)(pmon_prof_t* p);
    void    (*on_switch_in)(pmon_prof_t* p);
    void    (*on_switch_out)(pmon_prof_t* p);
    int     (*get_current_metric_value)(pmon_prof_t* prof, int key, uint64_t* value);
    void    (*module_counter_usage)(monitoring_module_counter_usage_t* usage);
    int     (*on_syswide_start_monitor)(int cpu, unsigned int virtual_mask);
    void    (*on_syswide_stop_monitor)(int cpu, unsigned int virtual_mask);
    void    (*on_syswide_refresh_monitor)(int cpu, unsigned int virtual_mask);
    void    (*on_syswide_dump_virtual_counters)(int cpu, unsigned int virtual_mask,
                                        pmc_sample_t* sample);
} monitoring_module_t;
```

La API de monitorización consta de una serie de funciones que pueden implementar los módulos de monitorización. Para implementar la API debemos instanciar un *struct* de C, cuyos campos son cada una de las funciones de la API. Sólo es obligatoria la implementación de algunas funciones, como `enable_module` o `disable_module`. La mayoría son opcionales. Los campos no especificados en la instancia son inicializados a `NULL`.

Estas funciones nos notifican los diferentes eventos que ocurren con respecto a la aplicación que se está monitorizando. Por ejemplo, cuando se crea o termina un proceso, cuando un proceso entra o sale de la CPU, etc. Otras funciones sirven para la correcta inicialización y terminación del módulo, lo que nos permite crear y destruir estructuras y definir los contadores virtuales que expone el módulo. Además, PMCTrack también soporta un modo de monitorización global, es decir, que no se monitoriza ninguna aplicación en concreto si no el conjunto de todo

el software que se ejecuta en un core, y la API dispone de funciones para ello.

Los módulos de monitorización pueden contener código específico de cada arquitectura y se pueden declarar dentro del código de inicialización del módulo del kernel de la arquitectura a la que pertenecen. PMCTrack destaca por la facilidad de crear estos módulos, pues sólamente hay que declararlos, y el resto del código puede ir encapsulado en sus propios ficheros.

En el siguiente capítulo se describe nuestra implementación de uno estos módulos de monitorización para extraer los datos de consumo energético del dispositivo Odroid SmartPower2.

Capítulo 3

Plataforma experimental

En este capítulo se presentan las características de las herramientas y dispositivos utilizados para la elaboración del Trabajo de Fin de Grado. En la sección 3.1 se describe la placa de desarrollo Odroid XU4 sobre la que se han ejecutado los *benchmarks*, y en la que está instalada la herramienta PMCTrack. Siguiendo con la sección dos que está dedicada al dispositivo Odroid SmartPower2, que es el que nos ha permitido realizar las medidas de consumo. Finalmente la sección 3.2 describe los cambios realizados al software de los dispositivos para adaptarlos a nuestras necesidades.

3.1 Odroid XU4

La placa de desarrollo Odroid XU4 [7] dispone de un SoC (*System on Chip*) Samsung Exynos 5422, equipado con un procesador multicore asimétrico big.LITTLE de 8 *cores*, de los cuales cuatro son *big* (Cortex A15 2.0GHz) y cuatro son *LITTLE* (Cortex A7 1.5GHz) como se muestra la tabla 3.1. También se ha incluido un diagrama de bloques para ilustrar la organización y distribución interna de los componentes de la placa (figura 3.1).

En la figura 3.2 se muestran imágenes de la parte superior e inferior de la placa. Esta plataforma cuenta con un ventilador incorporado para la refrigeración del SoC que no se muestra en la figura. Como veremos más adelante este ventilador introduce ciertas restricciones y condiciones al ejecutar los experimentos.

La placa Odroid XU4 que se nos proporcionó para la realización del TFG tenía instalado en la tarjeta MicroSD el sistema operativo Ubuntu 18.04 con el parche de PMCTrack del kernel Linux. Para interactuar con el sistema hemos usado dos interfaces: a través del puerto serie utilizando Minicom y por SSH. El puerto serie nos permite interactuar con el sistema

Modelos de core	**Último nivel de caché**	**Memoria principal**
4 x Cortex A15 @ 2.0GHz 4 x Cortex A7 @ 1.5GHz	2MB (L2) 512KB (L2)	2GB DDR3 @ 933MHz

Tabla 3.1: Especificaciones del procesador de Odroid XU4.

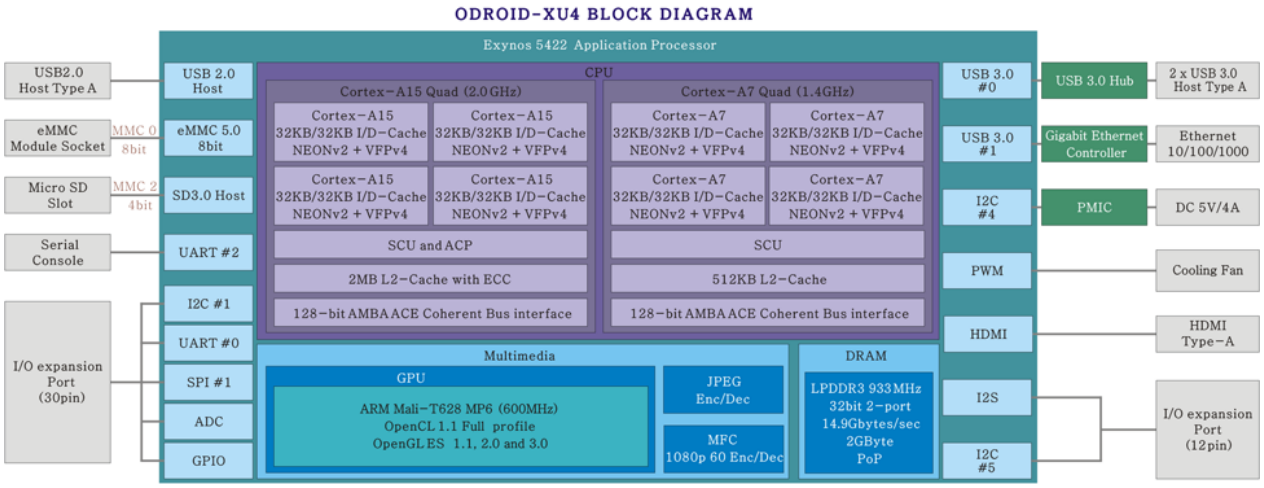


Figura 3.1: Diagrama de bloques Odroid XU4.

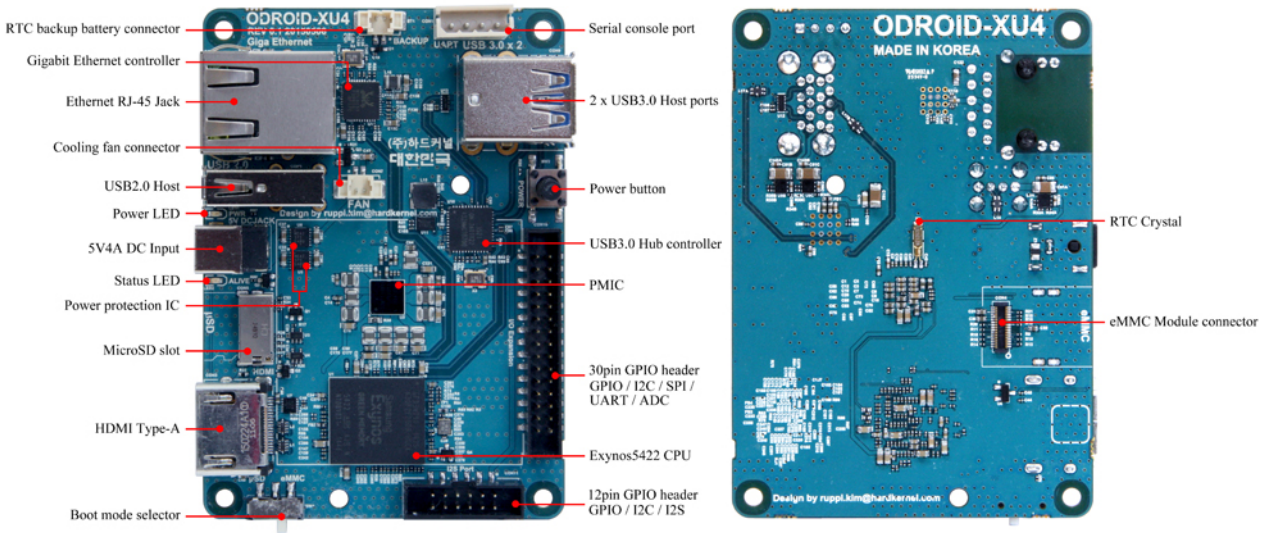


Figura 3.2: Dispositivo Odroid XU4.

mientras arranca. Esto es útil para depurar errores en un entorno de programación del kernel como es nuestro caso. Más tarde, cuando el entorno se encontraba correctamente configurado, nos conectamos remotamente por red usando SSH para ejecutar los *benchmarks* para mayor comodidad y para acceder a funcionalidad extra, como por ejemplo poder montar remotamente al sistema de ficheros mediante SFTP y trabajar con él desde el entorno gráfico de un equipo externo de desarrollo.

3.2 Odroid SmartPower2

El dispositivo Odroid SmartPower2 es un monitor de consumo energético externo. Conectando este dispositivo por la toma de corriente es capaz de proporcionar alimentación mientras mide el consumo de energía, la intensidad de corriente, la tensión y la potencia a lo largo del tiempo. Las mediciones se exponen al usuario a través de varios medios:

- Un **servidor web** accesible por WiFi. Accediendo a la IP del dispositivo con un navegador se nos da acceso a una pequeña aplicación web que muestra las mediciones de consumo.
- Un pequeño **monitor LCD** que muestra los valores actuales.
- Y por último, a través de un **puerto serie microUSB** que es el que utilizaremos en el módulo de monitorización desarrollado para hacer las mediciones de consumo.

La figura 3.3 ilustra las partes del dispositivo.

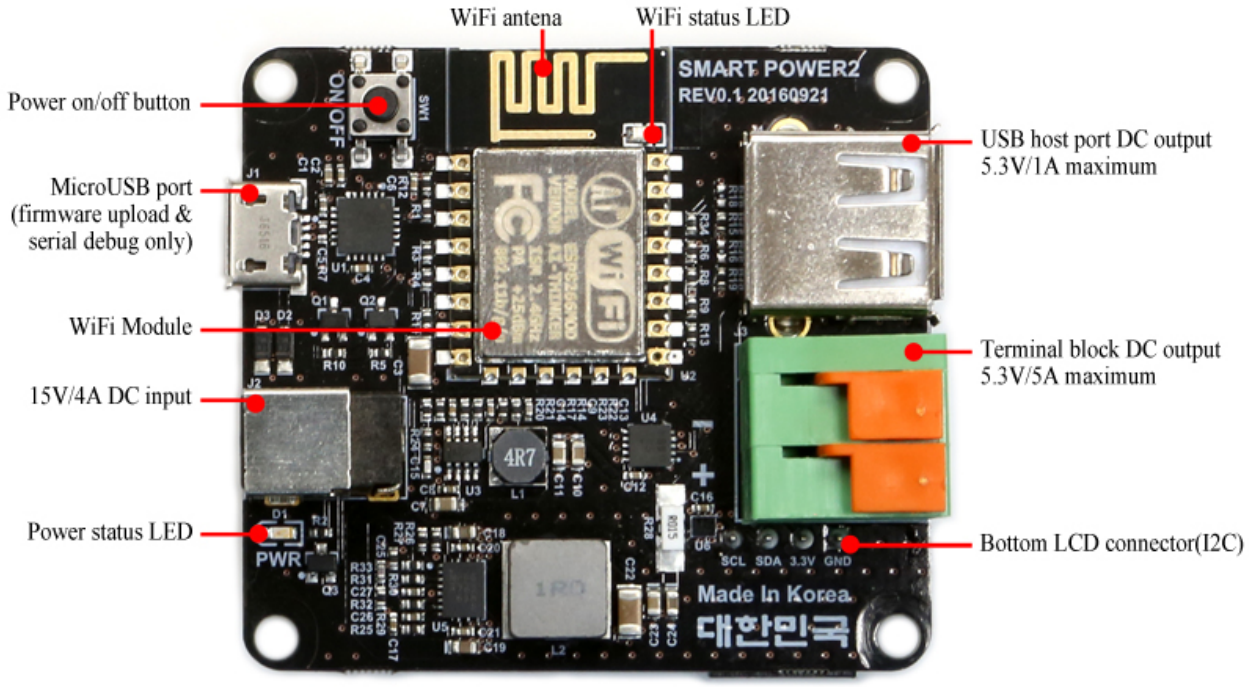


Figura 3.3: Odroid SmartPower2.

Tabla 3.2: Especificaciones técnicas de Odroid SmartPower2

Tensión de salida del bloque terminal	DC 4-5,3V en intervalos de 100mV
Corriente de salida máxima del bloque terminal	5A
Tensión de salida host USB	DC 4-5,3V en intervalos de 100mV
Corriente de salida máxima host USB	1A
Tensión de entrada toma de corriente	DC 9-15V

Para gestionar y dar salida a los datos de consumo por los diferentes medios, Odroid SmartPower2 consta de un firmware interno que explicaremos con más detalle en la siguiente sección.

3.3 Implementación

3.3.1 Firmware Odroid SmartPower2

El firmware del dispositivo Odroid SmartPower2, cuyo código fuente proporciona el fabricante, es el encargado de enviar los datos por el puerto serie del que leeremos. La salida de los datos está gobernada por un temporizador que se activa en intervalos de tiempo fijados en tiempo de compilación, y envía los datos obtenidos hasta el momento por todas las vías mencionadas previamente de forma simultánea. El código fuente original del firmware tiene una frecuencia de muestreo de $1s$. Si queremos utilizar estos datos para medir el consumo de las aplicaciones necesitaremos una frecuencia mayor, con lo que ha sido necesario modificar, recompilar y reinstalar el firmware en el dispositivo para que la frecuencia fuera de 100ms. El motivo es que no había muestras suficientes para la granularidad con la que se han hecho las medidas de consumo en los experimentos descritos en los siguientes capítulos.

En la wiki del fabricante [8] podemos encontrar los pasos para compilar el firmware desde un equipo de desarrollo conectado físicamente al dispositivo. El firmware se carga a través del mismo puerto microUSB por el que se leen los datos de consumo. Podemos obtener el código fuente del firmware y las herramientas de compilación con los siguientes comandos:

```
$ sudo apt install git python-pip
$ git clone https://github.com/hardkernel/smartpower2.git
$ sudo pip install -U platformio
```

Después de realizar los cambios deseados en el firmware, basta con ejecutar estos dos comandos para compilar y cargar el firmware en el dispositivo respectivamente:

```
$ sudo platformio run
$ sudo platformio run --target upload
```

3.3.2 Implementación del módulo de monitorización PMCTrack

Al comienzo del desarrollo de este TFG la herramienta PMCTrack disponía de soporte para la obtención de medidas de consumo energético en algunas plataformas x86 (gracias al soporte hardware Intel RAPL) y ARM, como las placas de desarrollo ARM Versatile Express TC2

o ARM Juno, equipadas con registros específicos de consumo energético accesibles desde el sistema operativo. Adicionalmente, como parte de un trabajo de fin de grado previo [9] se desarrolló una extensión de PMCTrack para obtener medidas de consumo energético usando el dispositivo Odroid SmartPower. A pesar de la similitud en el nombre comercial de este dispositivo [10] –que ya no se comercializa– y el usado en este TFG (Odroid Smart Power2) el segundo no es una evolución del primero. De hecho existen múltiples diferencias entre ambos monitores de consumo externo. Por ejemplo, antes no contaba con soporte USB por puerto serie y no ofrecía el resto de funcionalidades. Tan sólo emitía lo datos estableciendo una conexión USB a nivel de driver de dispositivo de bajo nivel, que es mucho más complejo. También operan a distinta tensión tanto de entrada como de salida. Debido a estas diferencias, el driver de Odroid SmartPower no es compatible con Odroid SmartPower2 y algunos miembros de la comunidad de PMCTrack habían expresado su interés en el soporte para Odroid SmartPower2.

En esta sección se explica nuestra implementación del driver que da soporte a Odroid SmartPower2 para PMCTrack mediante la implementación de la API de un módulo de monitorización.

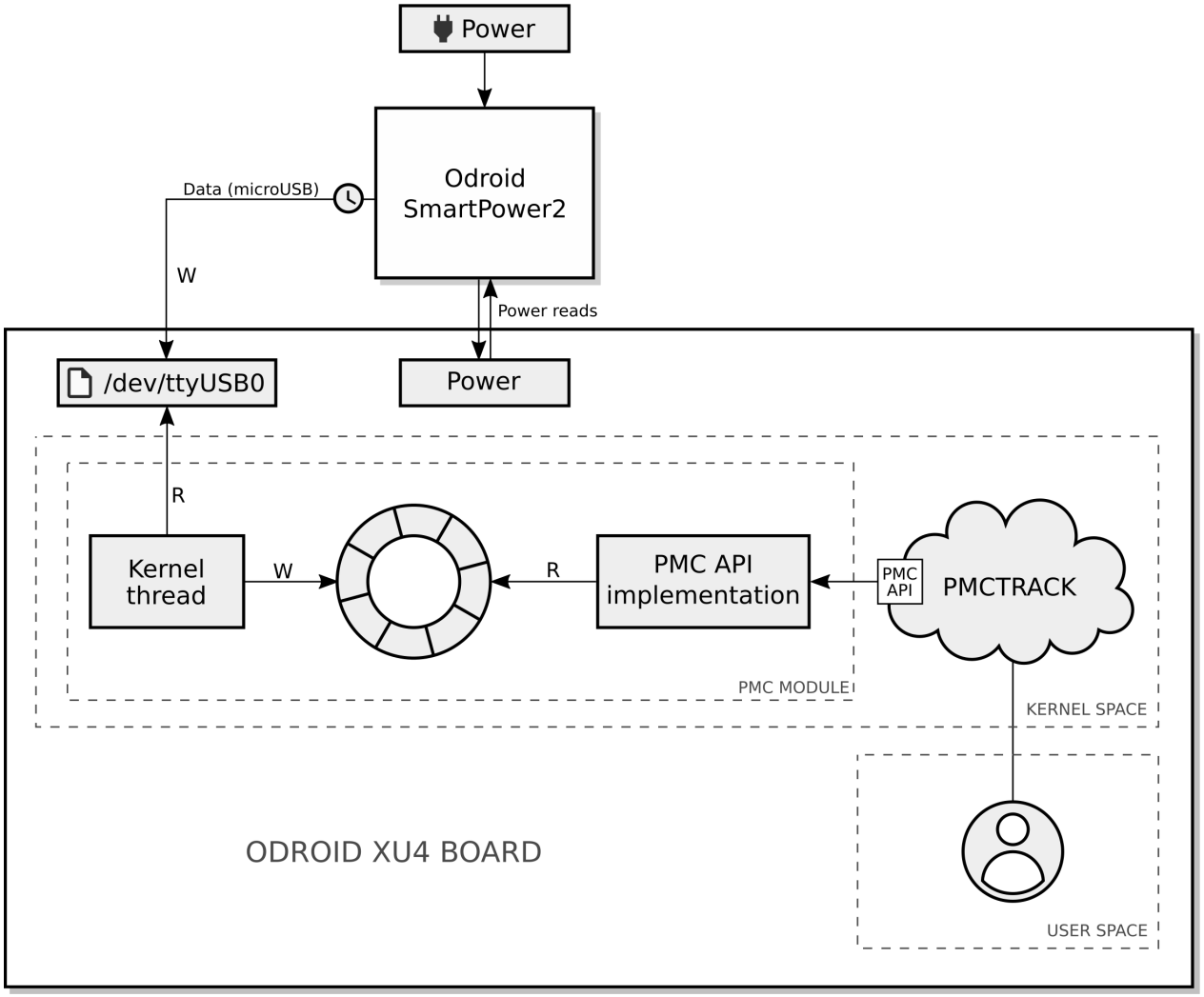


Figura 3.4: Estructura y organización del sistema.

El driver consta de dos componentes software. El primero realiza la lectura de los datos proporcionados por Odroid SmartPower2 empleando un hilo del kernel (*kernel thread*), y el segundo actúa en respuesta a llamadas de funciones de la API de PMCTrack para devolver la información almacenada. La figura 3.4 ilustra la interacción entre las distintas partes.

Cuando conectamos Odroid SmartPower2 a Odroid XU4 por USB el sistema operativo crea automáticamente un fichero de dispositivo en el directorio `/dev`. Leyendo de ese fichero podemos obtener los datos en formato string enviados desde Odroid SmartPower2 a través del puerto serie USB. Un *kernel thread* es el encargado de leer de dicho fichero y procesar los valores, almacenando cada muestras obtenida en un buffer circular en orden cronológico junto con una marca de tiempo. El buffer tiene un tamaño limitado, las muestras más recientes van sobreescribiendo las posiciones en las que se almacenan las más antiguas.

Cada vez que PMCTrack solicita datos hace una llamada a distintas funciones de la API del módulo de monitorización. Dichas llamadas vienen acompañadas de otra marca de tiempo que indica cuándo se procedió a la última lectura de los datos de consumo. Con esa información se leen del buffer las muestras posteriores a esa marca de tiempo, y se devuelve una sola muestra con la media de cada uno de los valores.

Todas las operaciones de lectura y escritura en el buffer y otras estructuras están protegidas mediante cerrojos (*read-write spinlocks*), dado que se accede a ellas de forma concurrente desde varios flujos de ejecución del kernel.

Para ampliar PMCTrack con nuestro módulo de monitorización hay que definir el módulo desde el código de inicialización de PMCTrack. Esto exige recompilar PMCTrack y modificar los *scripts* de compilación del módulo del kernel de PMCTrack de nuestra plataforma (ficheros *Kbuild* y *Makefile*). A estos ficheros debemos añadir los nuevos ficheros del módulo de monitorización y definir un nuevo *flag* del compilador de C para que el módulo de monitorización sólo esté disponible cuando la compilación se realice para nuestra plataforma. PMCTrack viene equipado con una serie de utilidades de línea de comandos que nos facilitan la compilación del código y la instalación. Para que los comandos correspondientes estén disponibles en nuestro `PATH`, es preciso ejecutar el siguiente comando:

```
$ . shrc
```

A continuación, para compilar el código simplemente debemos ejecutar:

```
$ pmctrack-manager build
```

Si la compilación finaliza correctamente disponemos de una versión de PMCTrack con la funcionalidad ampliada de nuestro módulo de monitorización. El siguiente paso es instalar el nuevo módulo del kernel de PMCTrack en el sistema operativo con este comando, indicando cuál es nuestra plataforma:

```
$ pmctrack-manager load-module
Several compatible modules found.
1) arm
2) odroid-xu
Please pick one [1-2]
```

```
2
Loading kernel module mchw_odroid_xu ...
==========================================
```

Una vez instalado el módulo del kernel, podemos listar los módulos de monitorización de PMCTrack disponibles en nuestra plataforma leyendo de un fichero del sistema `/proc`:

```
$ cat /proc/pmc/mm_manager
[*] 0 - This is just a proof of concept
[ ] 1 - IPC sampling SF estimation module
[ ] 2 - Odroid Smart Power
[ ] 3 - Odroid Smart Power 2
```

Para activarlo escribimos `activate N` sobre el mismo fichero, donde N es el índice del módulo según aparecía en el paso anterior. En este caso es el 3:

```
$ echo activate 3 > /proc/pmc/mm_manager
$ cat /proc/pmc/mm_manager
[ ] 0 - This is just a proof of concept
[ ] 1 - IPC sampling SF estimation module
[ ] 2 - Odroid Smart Power
[*] 3 - Odroid Smart Power 2
```

Una vez activado el módulo podremos ver una lista de los contadores virtuales proporcionados por el módulo de monitorización activo.

```
$ pmc-events -V
[Virtual counters]
power_mw
current_ma
energy_uj
```

Ahora podemos ejecutar el comando `pmctrack` para que nos muestre los valores de los nuevos contadores virtuales. Para ello utilizamos el siguiente comando, que devuelve el número de instrucciones retiradas (contador hardware) por segundo (-T 1) así como los datos de nuestros contadores virtuales como el consumo energético en μJ, la intensidad de corriente media (mA) y la potencia media (mW) consumidas por Odroid XU4 mientras se ejecuta un programa de SPEC CPU2006 en la CPU 0 (-b 0), uno de los *cores* LITTLE Cortex A7 en nuestra plataforma.

```
$ pmctrack -T 1 -b 0 -c instr -V power_mw,current_ma,energy_uj ./mcf06
[Event-to-counter mappings]
pmc1=instr
virt0=power_mw
virt1=current_ma
virt2=energy_uj
[Event counts]
nsample    pid     event          pmc1     virt0     virt1        virt2
      1  27244      tick     603042053      4883      1205      4395100
```

```
      2  27244      tick     147946132      4815      1188      4334100
      3  27244      tick     145162288      4823      1190      4341000
      4  27244      tick     140157465      4827      1191      4344600
      5  27244      tick     138359364      4819      1189      4819900
      6  27244      tick     119464114      5163      1274      4647000
      7  27244      tick     132362422      4835      1193      4351600
...
```

Capítulo 4

Modelo de medida de consumo

Este capítulo describe el procedimiento para obtener el consumo de potencia neto (considerado en los experimentos) a partir del consumo total del sistema proporcionado por Odroid SmartPower2. Asimismo, se presentan las métricas utilizadas en nuestro entorno experimental para evaluar el rendimiento y eficiencia energética derivada de la ejecución de una aplicación en distintos tipos de *core* de un sistema multicore asimétrico.

4.1 Aproximación del consumo de potencia

El objetivo es caracterizar la eficiencia energética de una aplicación al ejecutarse en distintos tipos de core. Por ello, para intentar alcanzar la mayor precisión posible, es preciso intentar obtener exclusivamente el consumo de energía de la aplicación a partir del consumo total del sistema eliminando otros factores. Para aproximar el consumo neto de una aplicación consideraremos únicamente el consumo de la memoria y del *core* en el que se ejecuta.

Para ello introducimos las siguientes fórmulas[1]:

$$C_{app(small)} = C_{system} - C_{small}$$

$$C_{app(big)} = C_{system} - C_{big}$$

Donde:

- C_{big} representa el consumo de potencia del sistema en reposo (*idle*) cuando los *cores big* operan a la máxima frecuencia ($2GHz$) y los *cores LITTLE* trabajan a la mínima frecuencia configurable ($200MHz$).
- C_{small} representa el consumo de potencia del sistema en reposo cuando los *cores small* operan a la máxima frecuencia (*1.5GHz*) y los *cores big* trabajan a la mínima frecuencia configurable (*200MHz*).
- C_{system} representa el consumo de potencia total del sistema.
- $C_{app(big)}$ y $C_{app(small)}$ representan el consumo neto de ejecutar una aplicación en uno de los *cores big* o *LITTLE* respectivamente.

[1]Todos los valores de consumo están en milivatios (mW).

En nuestro análisis experimental usamos PMCTrack para obtener los citados valores de consumo en cada una de las configuraciones de frecuencia. En particular, para obtener C_{big} y C_{small} medimos el consumo de potencia en mW mientras se ejecuta el programa `sleep`, que libera la CPU durante el tiempo en segundos especificado como parámetro.

```
$ pmctrack -T 1 -c instr -V power_mw sleep 200
[Event-to-counter mappings]
pmc1=instr
virt0=power_mw
[Event counts]
nsample    pid      event          pmc1         virt0
      1  16974       tick        300155          3279
      2  16974       tick             0          3268
      3  16974       tick             0          3266
      4  16974       tick             0          3259
      5  16974       tick             0          3511
      6  16974       tick             0          3530
...
```

Los resultados fluctúan un poco, con lo que nos quedaremos con el mínimo de los valores de todas las muestras.

En trabajos anteriores se ha observado que un factor importante a tener en cuenta a la hora de calcular el consumo de las cargas de trabajo es el ventilador [9]. Por defecto está configurado para encenderse de forma automática cuando aumenta la temperatura o el uso de los procesadores. Esto afecta a las medidas, dando picos de consumo en principio impredecibles como se puede observar en la figura 4.1. Por eso hemos optado por poner el ventilador en modo manual para que esté siempre encendido y así poder añadirlo al consumo en reposo del sistema de forma consistente.

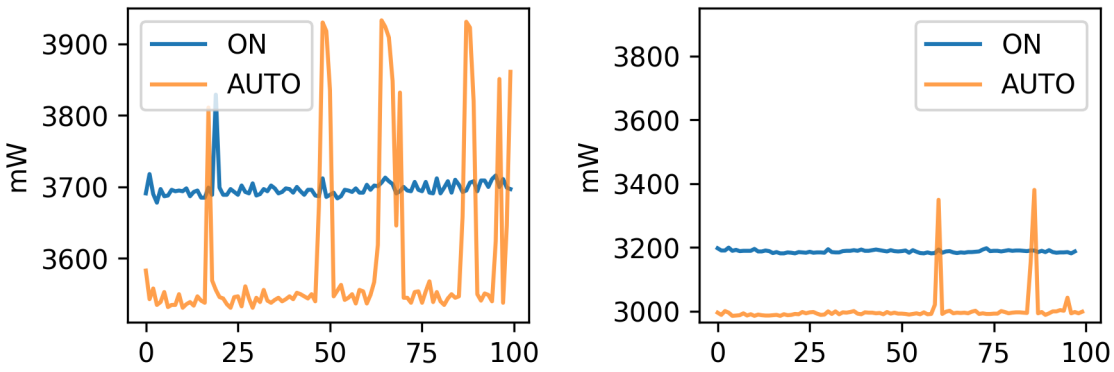


Figura 4.1: Comparativa de consumo con el ventilador encendido y en automático en un *core* big (izquierda) y en un *core* little (derecha).

Con los siguientes comandos podemos cambiar la política de *automático* a *manual* y poner la velocidad al máximo respectivamente:

```
$ echo 0 | sudo tee /sys/devices/platform/pwm-fan/hwmon/hwmon0/automatic
$ echo 255 | sudo tee /sys/devices/platform/pwm-fan/hwmon/hwmon0/pwm1
```

Para revertir la política al estado anterior y que se regule de forma automática utilizamos el mismo comando cambiando el valor de *automatic* de 0 a 1.

```
$ echo 1 | sudo tee /sys/devices/platform/pwm-fan/hwmon/hwmon0/automatic
```

En nuestra plataforma experimental, los valores obtenidos de C_{big} y C_{small} son 3678 mW y 3181 mW respectivamente y son los que utilizaremos en los próximos cálculos.

4.2 Métricas utilizadas

Para analizar los datos obtenidos utilizaremos las siguientes métricas propuestas en trabajos anteriores [11]–[14].

El factor de ganancia o SF (*Speedup Factor*) es el ratio de instrucciones por segundo ejecutadas en un *core big* frente a las instrucciones ejecutadas por segundo en un *core small* en un AMP. Esta métrica indica el grado de mejora al ejecutar una secuencia de instrucciones en un *core big* frente a un *core small*.

$$SF = \frac{IPS_{big}}{IPS_{small}} \tag{4.1}$$

El factor de eficiencia energética o EEF (*Energy efficiency factor*) relaciona el factor de ganancia con la energía neta consumida por instrucción en un *core big* (NET_EPI_{Big}). Como veremos en el siguiente capítulo, el EEF denota el balance entre el rendimiento ganado y la energía adicional que gastamos al ejecutar una carga de trabajo a un *core big*.

$$EEF = \frac{SF}{NET\_EPI_{big}} \tag{4.2}$$

Estas dos métricas han demostrado ser útiles para optimizar distintos objetivos en la tarea de planificación de procesos. Mientras que el SF sirve para optimizar la justicia y el rendimiento global del sistema, el EEF sirve para optimizar la eficiencia energética. En el siguiente capítulo veremos de qué manera podemos calcular estos coeficientes.

Capítulo 5

Análisis experimental

En este capítulo se analizan las medidas de consumo energético y rendimiento obtenidas al ejecutar los *benchmarks* de SPEC CPU 2000 y 2006 en cada uno de los tipos de *core* de la placa Odroid XU4. Con los datos obtenidos se han generado distintos tipos de gráficas para realizar la caracterización de las aplicaciones y poder discernir qué aplicaciones obtienen mayor eficiencia energética en cada uno de los *cores*. Finalmente se realiza un modelo de predicción para el SF y EEF en función de otras métricas. De este modo, dependiendo de la política de planificación del sistema, ésta puede decidir en qué *core* es mejor ejecutar cada proceso en términos de eficiencia energética.

5.1 Caracterización de aplicaciones mediante SF y EEF

Para poder realizar la caracterización se han recopilado los valores de distintos eventos hardware y de consumo energético para cada uno de los *benchmarks* de SPEC CPU en los distintos tipos de *core* utilizando PMCTrack. Estos datos han sido ordenados y procesados. A partir de ellos hemos calculado otras métricas de alto nivel.

Gracias a PMCTrack hemos podido recopilar de forma conjunta tanto los datos de los contadores hardware como los valores de consumo energético. Utilizando el modo EBS (*Event-Based Sampling*) de PMCTrack, obtenemos una muestra cada 700M de instrucciones hasta un máximo de 250 muestras. El número final de muestras puede ser menor si el programa termina antes.

Para llevar a cabo las mediciones, se ha escogido un *core* representativo de cada *cluster* del procesador, y se han ejecutado de forma secuencial todos los *benchmarks* en cada uno de ellos. En todo momento sólo se ejecutaba en el sistema uno de los *benchmarks* y en un solo *core*.

En nuestra plataforma experimental se ha utilizado un medidor de consumo externo que sólo es capaz de proporcionar la energía consumida globalmente por todo el sistema. Al tratarse de un entorno controlado en el que sólo se ejecuta una aplicación de forma simultánea, asumimos que toda la energía extra consumida (con respecto a la consumida por el sistema en reposo) es debido a la ejecución de la aplicación que se está ejecutando en ese momento. De esta

forma podemos calcular el EPI aproximado y en consecuencia el EEF.

La obtención de los diferentes eventos hardware se ha dividido en distintas fases debido a la limitación del número de contadores hardware de los que dispone el procesador de Odroid XU4, por lo que se ha repetido cuatro veces la ejecución de los *benchmarks* en cada *core* monitorizando eventos distintos cada vez. Esto no es un problema a la hora de comparar los datos, debido a que todas las muestras tienen la misma ventana de instrucciones (700 millones) y cada programa se comporta de la misma manera en cada una de las ventanas entre varias ejecuciones.

A partir de los eventos de bajo nivel obtenidos se han calculado las métricas de alto nivel requeridas para el análisis de la siguiente sección, y otras métricas que nos permitirán calcular el EEF y el SF para compararlos con dichas métricas. A continuación se enumeran las métricas obtenidas:

- **IPC**: Instrucciones por ciclo.
- **EPI**: Energía consumida por instrucción medida en micro julios (μJ).
- **LLCRPKI**: Accesos a memoria caché de último nivel por cada 1000 instrucciones.
- **LLCMPKI**: Fallos de memoria caché de último nivel por cada 1000 instrucciones.
- **LLCRPKC**: Accesos a memoria caché de último nivel por cada 1000 ciclos.
- **LLCMPKC**: Fallos de memoria caché de último nivel por cada 1000 ciclos.
- **L1CRPKI**: Accesos a la memoria caché de primer nivel por cada 1000 instrucciones.
- **L1CMPKI**: Fallos de memoria caché de primer nivel por cada 1000 instrucciones.
- **L1CRPKC**: Accesos a la memoria caché de primer nivel por cada 1000 ciclos.
- **L1CMPKC**: Fallos de memoria caché de primer nivel por cada 1000 ciclos.
- **DTLBMPKI**: Fallos de TLB de datos cada 1000 instrucciones.
- **ITLBMPKI**: Fallos de TLB de instrucciones cada 1000 instrucciones.
- **DTLBMPKC**: Fallos de TLB de datos cada 1000 ciclos.
- **ITLBMPKC**: Fallos de TLB de instrucciones cada 1000 ciclos.

Tras calcular estas métricas, calculamos el SF y el EEF medio por aplicación utilizando las ecuaciones descritas en la sección 4.2. En la figura 5.1 se muestra una comparativa del EEF y el SF medios para cada uno de los *benchmarks* de SPEC CPU considerados en nuestro análisis. Se han ordenado los *benchmarks* por SF de forma creciente.

En la figura se observa que el rendimiento relativo no está correlacionado con el factor de eficiencia energética. Si bien el EEF es proporcional al SF, dicha proporción no es constante entre todas las posibles configuraciones de software que es posible ejecutar, dado que la energía que utiliza un procesador para su ejecución depende de múltiples factores. En trabajos anteriores se habían observado tendencias similares a las obtenidas [9].

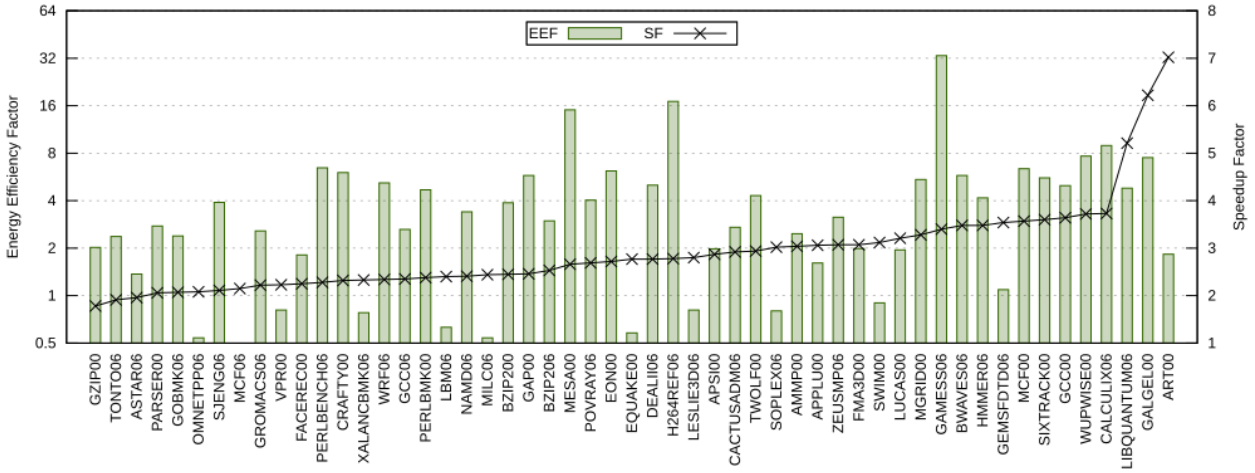


Figura 5.1: EEF y SF medio para cada aplicación de SPEC CPU 2000 y 2006.

También se ha comparado el consumo energético en los distintos *cores* con el SF en la figura 5.2, en la que se muestra la energía media por instrucción de cada aplicación al ser ejecutada en un *core big* o en un *core LITTLE* y el SF de cada uno de los *benchmarks*. De nuevo podemos observar que ni el consumo energético de los *cores* ni la diferencia entre ellos está correlacionada con el SF.

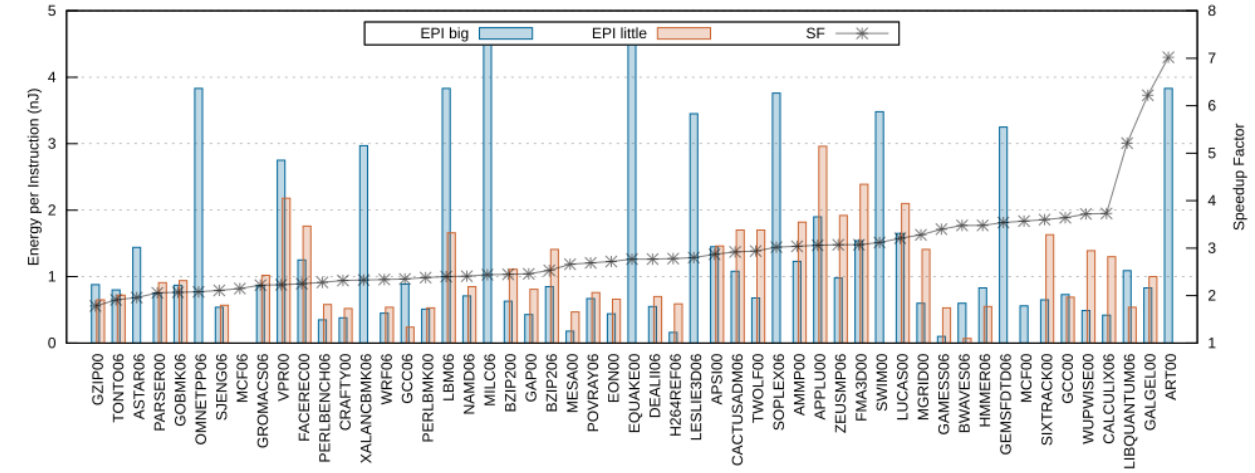


Figura 5.2: EPI_{big}, EPI_{little} y SF medio para cada aplicación de SPEC CPU 2000 y 2006.

En base a lo observado, podemos llegar a la conclusión de que optimizar la eficiencia energética no es lo mismo que optimizar el rendimiento.

Por otra parte, en trabajos anteriores se ha observado que los valores medios de SF y EEF no siempre son representativos para todas las aplicaciones [14]. Estas métricas pueden variar en el transcurso de la ejecución de una aplicación, y por tanto el valor medio puede ser engañoso. En general, es posible encontrar fases con diferentes características en las que una aplicación se comporta de una determinada manera. En la siguiente sección utilizaremos algunos de

estos casos para analizar la correlación del EEF con otras métricas de alto nivel calculadas a partir de contadores hardware.

5.2 Evolución del SF y EEF en el tiempo

Como se ha visto en la sección anterior, el valor del SF y el EEF puede variar a lo largo del tiempo, y para poder caracterizar una aplicación es preciso conocer sus valores. Sin embargo, no es posible obtener medidas de consumo energético desglosadas por aplicación en una carga de trabajo multiprogramada. Sin estas medidas no es posible calcular el EEF de forma directa, por lo que en esta sección se comparará el EEF observado en el transcurso de la ejecución de los programas con otras métricas de alto nivel calculadas a partir de las medidas extraídas de los contadores hardware. El objetivo es encontrar una correlación entre el EEF y otras métricas, para calcular de forma aproximada el EEF utilizando los valores de los contadores hardware de los *cores*.

En primer lugar vamos a analizar el efecto que tienen los fallos de caché de último nivel (*Last Level Cache Misses*) por cada mil ciclos sobre el EEF a lo largo de la ejecución de las distintas aplicaciones.

En la figura 5.3 se diferencian con claridad las distintas fases por las que pasa la aplicación `astar` durante su ejecución. A grandes rasgos se puede observar que cuando el número de fallos de caché de último nivel aumenta la eficiencia energética disminuye, y viceversa.

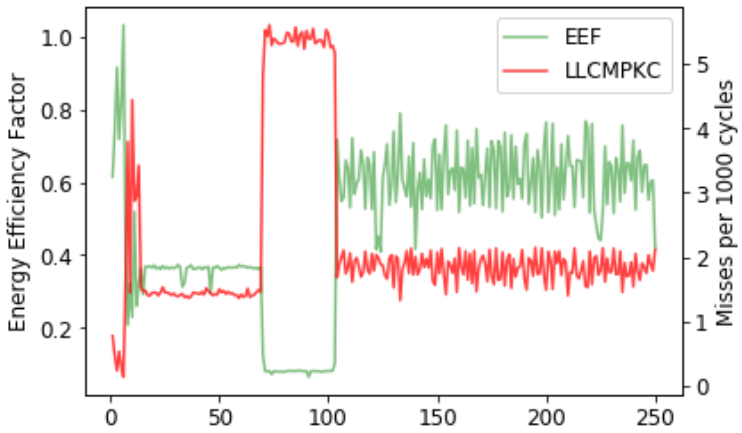


Figura 5.3: EEF y fallos de caché de último nivel observados para el programa `astar` en un *core big* a lo largo del tiempo.

Otro caso que sigue el mismo patrón es el programa `calculix`, representado en la figura 5.4. En este caso se distinguen dos fases de ejecución, una con más accesos a memoria caché de último nivel y otra con menos accesos. En la fase con más accesos el EEF también disminuye.

A grandes rasgos se puede deducir que el incremento de lecturas y fallos de caché de último nivel afecta en gran medida al valor del EEF. Es decir, la eficiencia energética de ejecutar estas fases de ejecución con elevado uso de la memoria caché de último nivel en un *core big* es menor que si se ejecutara en un *core little*.

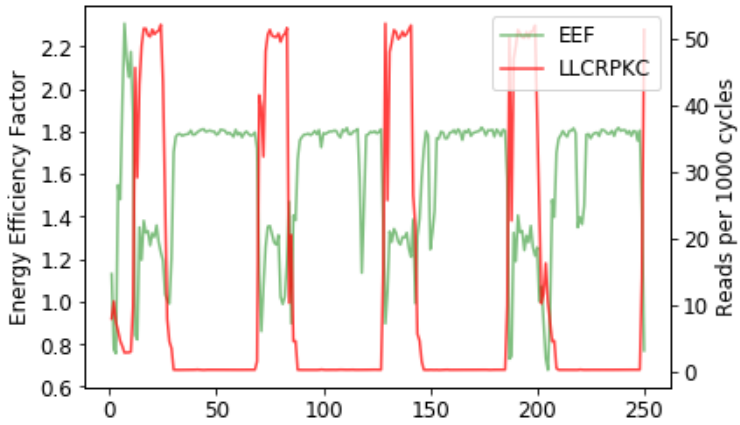


Figura 5.4: EEF y lecturas de caché de último nivel observados para el programa `calculix` en un *core big* a lo largo del tiempo.

Sin embargo, en otros casos es más difícil reconocer los patrones de comportamiento de los que hablábamos, ya que los programas no siempre presentan fases estables de ejecución. Este es el caso del programa `apsi`, ilustrado en la figura 5.5. En este caso resulta imposible analizar de forma visual el resultado.

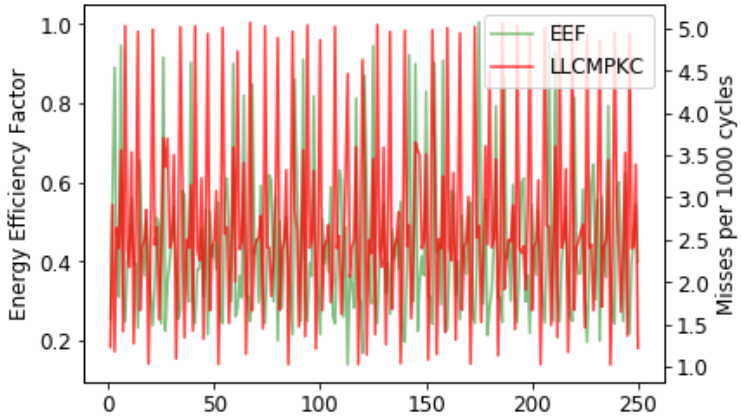


Figura 5.5: EEF y fallos de caché de último nivel observados para el programa `apsi` en un *core big* a lo largo del tiempo.

Cabe destacar que no todos los programas responden igual ante el aumento o disminución de los fallos de caché de último nivel en términos de eficiencia energética. Por ejemplo, el caso del programa `tonto`, ilustrado en la figura 5.6. En este caso los fallos de caché y el EEF varían de forma directa en lugar de inversa. Por tanto, no podemos aproximar el EEF utilizando únicamente el número de fallos y accesos a memoria caché. Es necesario tener en cuenta otros factores.

Otra métrica que podría estar relacionada con el valor del EEF es el número de fallos de caché de primer nivel (L1C). En la figura 5.7 aparece la evolución durante la ejecución del programa `astar` de esta métrica comparada con el EEF.

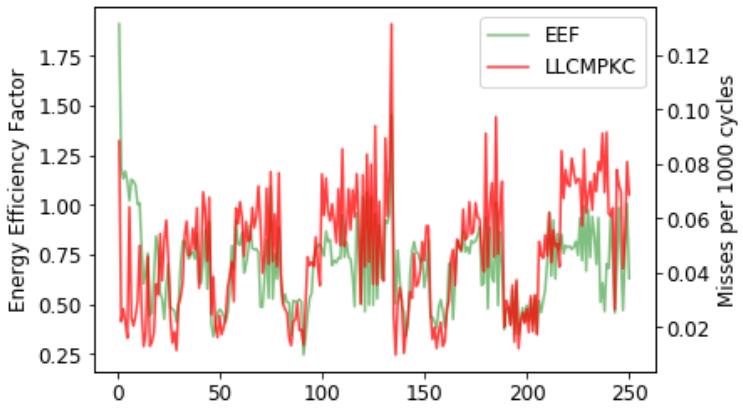


Figura 5.6: EEF y fallos de caché de último nivel observados para el programa `tonto` en un *core big* a lo largo del tiempo.

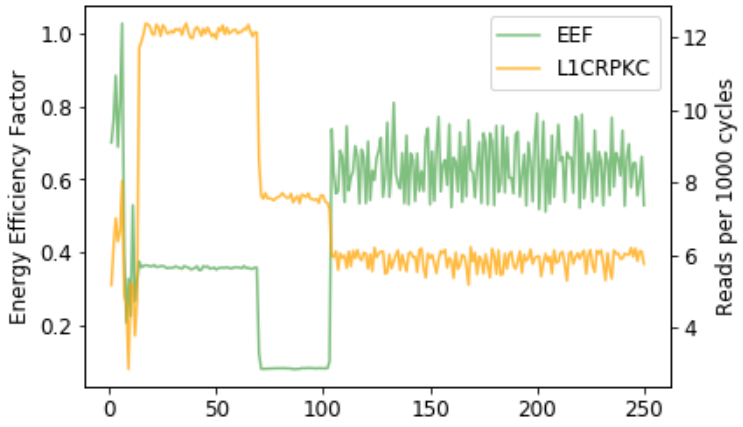


Figura 5.7: EEF y accesos de caché de primer nivel observados para el programa `astar` en un *core big* a lo largo del tiempo.

El comportamiento que tiene el EEF es similar a los casos de los fallos de LLC. Cuando hay un aumento significativo de los fallos de caché de primer nivel como ocurre al comienzo de la figura 5.7, el EEF disminuye drásticamente. Asimismo, tras la muestra 100 se produce una reducción de los fallos de L1C, y se produce un aumento considerable en el EEF. Sin embargo, es importante destacar que al llegar al tramo de la muestra 60 se produce una disminución de las accesos a la L1C, que también provoca una disminución del EEF. Esto es debido a que este no es el único factor que influye en el EEF, por lo que es razonable que no siempre se ajuste perfectamente a una única métrica.

Por último vamos a analizar el efecto sobre el EEF, de los fallos de TLB de datos (*Data Translation Look-aside Buffer*). En la figura 5.8 se muestra el número de fallos de TLB comparado con el EEF durante la ejecución del programa `astar`.

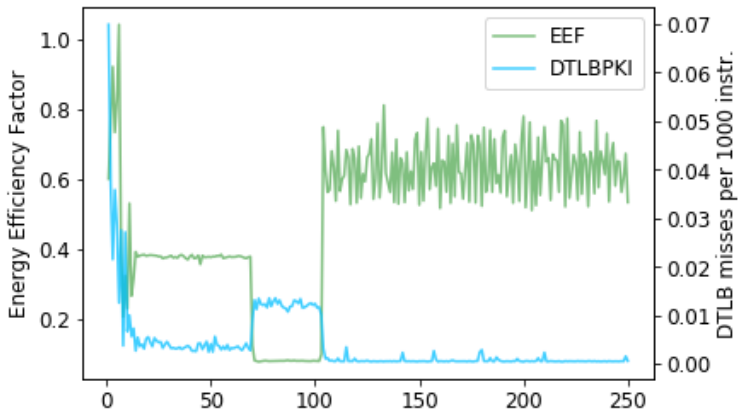


Figura 5.8: EEF y fallos de DTLB observados para el programa `astar` en un *core big* a lo largo del tiempo.

Se observa nuevamente un patrón idéntico al resultante de la comparación del EEF con los fallos de caché de último nivel. Cuando los fallos de DTLB se incrementan se produce un descenso del EEF. Por el contrario, cuando los fallos de DTLB decrecen o son nulos, el EEF aumenta.

Tras observar los distintos patrones de comportamiento al monitorizar varias métricas durante la ejecución de un conjunto de aplicaciones, podemos concluir a grandes rasgos lo siguiente: Los accesos y fallos de L1C y LLC, así como los fallos de DTLB, influyen en gran medida sobre el valor del EEF. Es decir, en general, un programa es más eficiente energéticamente si se ejecuta en un *core LITTLE* mientras hace un uso intensivo de la memoria. Sin embargo, no es posible determinar el valor del EEF únicamente a partir de una de estas métricas. Es necesario hacer un análisis más preciso que contemple varias de ellas.

5.3 Modelos de predicción

En la sección anterior mostramos la relación del SF y el EEF con otras métricas es compleja, y no siempre guardan una relación directa con una de ellas. Otros trabajos [12], [14] han

demostrado que el SF y el EEF pueden aproximarse de forma relativamente precisa utilizando modelos de predicción en tiempo de ejecución.

Para construir el modelo de predicción seguiremos paso a paso la metodología descrita en [12], tanto para el EEF como para el SF. En primer lugar se ejecutan los *benchmarks* en los *cores* big y LITTLE para obtener los valores del IPS para ambos tipos de core, así como la EPI_{big} necesarios para calcular el SF y EEF. A continuación se calculan las métricas, ya que las necesitamos en el análisis descrito en la sección anterior. Una vez obtenidos los datos a lo largo del tiempo para cada uno de los *benchmarks*, identificamos fases de ejecución de larga duración en los mismos para despreciar variaciones repetitivas en el cálculo de las estimaciones. Estas fases de ejecución se caracterizan por ser estables y no tener muchas variaciones. Los datos obtenidos sirven de entrada al programa de *machine learning* WEKA [15], que nos permitirá construir los modelos de estimación.

WEKA dispone de múltiples técnicas de *machine learning* para construir modelos de estimación. Para construir nuestras estimaciones hemos utilizado la técnica de regresión aditiva [16], tal y como se ha hecho en [12]. El resultado de una estimación hecha por regresión aditiva es una función que recibe como parámetros un valor de salida y una serie de parámetros de entrada. En nuestro caso las entradas son los valores de las métricas y la salida es el SF o el EEF que intentamos predecir. El objetivo es intentar aproximar el valor de la salida a partir de los valores de entrada minimizando el error. Esto lo consigue con un sumatorio con un elemento por cada variable de entrada. Cada elemento del sumatorio es una función simple que depende del valor de entrada. En este caso la función es un condicional cuyo resultado se decide entre dos valores fijos (positivos o negativos) en función de si el valor de entrada supera o no cierto umbral. Lo que diferencia a una estimación de otra es el valor de los umbrales y los valores que se suman en función de los umbrales.

Vamos a ilustrar esto con un ejemplo. Esto es una porción de la salida de WEKA. En este caso nos indica el umbral del IPC considerado. Si el valor del IPC supera el umbral $(0,24)$ se sumaría un *delta* de $-0,11$ a la salida de la estimación, en caso contrario se le sumaría un *delta* de $0,32$.

```
ipc <= 0.2492435 : 0.32524065391290025
ipc > 0.2492435 : -0.11675305525078472
```

Será necesario predecir el SF y EEF desde cada *core* utilizando los valores recabados por sus contadores hardware. Los pesos de las variables de entrada para la función de estimación no serán los mismos, dado que los valores las métricas son diferentes en cada tipo de *core* al tratarse de un AMP.

Para construir los modelos de estimación se han proporcionado todas las métricas disponibles como entrada. Sin embargo, los modelos generados no siempre van en función de todas las métricas. WEKA descarta automáticamente los valores de entrada cuyo peso en el cálculo de la estimación no es lo suficientemente significativo. En la tabla 5.1 se muestran las métricas utilizadas para las estimaciones de EEF y SF desde cada tipo de *core*.

El resultado de las estimaciones hechas por WEKA se puede observar en las figuras 5.9 y 5.10. En ellas se muestra la comparación de los valores observados de EEF y SF con las

Métricas de rendimiento	**EEF_{big}**	**EEF_{little}**	**SF_{big}**	**SF_{little}**
Instrucciones por ciclo (IPC)	✓	✓		✓
Accesos LLC por cada 1000 instr.				
Fallos LLC por cada 1000 instr.		✓		✓
Accesos LLC por cada 1000 ciclos	✓	✓	✓	
Fallos LLC por cada 1000 ciclos	✓		✓	✓
Fallos DTLB por cada 1000 instr.			✓	
Fallos ITLB por cada 1000 instr.			✓	✓
Fallos DTLB por cada 1000 ciclos	✓	✓		
Fallos ITLB por cada 1000 ciclos	✓	✓	✓	✓
Accesos L1C por cada 1000 instr.	✓	✓	✓	✓
Fallos L1C por cada 1000 instr.			✓	
Accesos L1C por cada 1000 ciclos			✓	
Fallos L1C por cada 1000 ciclos		✓		✓

Tabla 5.1: Contadores hardware utilizados para las estimaciones de cada métrica.

estimaciones hechas para la placa Odroid XU4. Cada punto pertenece a uno de los *benchmarks* realizados. Los puntos más cercanos a la línea son los que más se ajustan a la predicción.

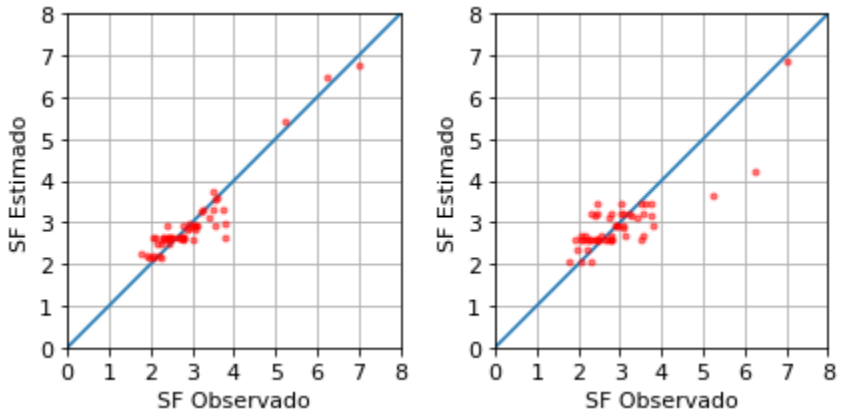


Figura 5.9: Predicción del SF desde un core big (izquierda) y un core LITTLE (derecha) mediante regresión aditiva.

Los coeficientes de correlación para la estimación del SF son 0,97 y 0,92 para los cores big y LITTLE respectivamente. Los coeficientes de correlación en los core big y LITTLE para la estimación del EEF son respectivamente 0,998 y 0,997. Estos índices de correlación son bastante aceptables, con lo que las estimaciones podrían ser efectivamente útiles para calcular el EEF y el SF en sistemas reales de forma relativamente precisa, partiendo de las métricas calculadas a partir de los valores de los contadores hardware.

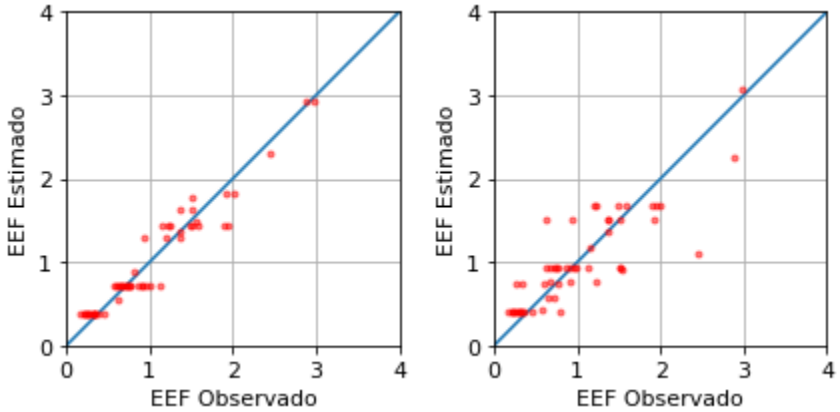


Figura 5.10: Predicción del EEF desde un core big (izquierda) y un core LITTLE (derecha) mediante regresión aditiva.

Capítulo 6

Conclusiones

6.1 Conclusiones

Trabajos previos han demostrado que los procesadores multicore asimétricos (*Asymmetric Multicore Processors* ó AMPs) contribuyen a mejorar la eficiencia energética de forma sustancial, siempre y cuando el planificador del sistema operativo tenga en cuenta las características de las distintas aplicaciones de una carga de trabajo a la hora de asignarlas a distintos tipos de core [12, p. [@cj2018]]. Este tipo de procesadores están cada vez más presentes en los dispositivos móviles comerciales, por lo que actualmente resultan muy relevantes. El procesador big.LITTLE de ARM [referencia] es quizás el procesador multicore asimétrico más extendido a nivel comercial.

Para facilitar el diseño y el análisis experimental de algoritmos de planificación eficientes sobre la placa Odroid XU4 -equipada con un procesador multicore asimétrico big.LITTLE- en este Trabajo de Fin de Grado hemos desarrollado el soporte necesario en la herramienta PMCTrack [6] para obtener medidas de consumo energético en Linux usando el monitor de consumo Odroid SmartPower2 [8]. Gracias a este dispositivo, podemos acceder a datos de consumo energético en tiempo real mientras distintas aplicaciones se ejecutan en diferentes cores de la plataforma. Estos datos son accesibles tanto en modo usuario, a través de las herramientas gráficas y de línea de comandos de PMCTrack, como desde cualquier componente del sistema operativo, incluido el planificador de procesos. Esto ha sido posible gracias a la capacidades de extensión de la herramienta PMCTrack, que permiten ampliar su funcionalidad mediante *plugins* denominados *módulos de monitorización*. La API accesible desde un módulo de monitorización, uno de los cuales fue necesario implementar como parte este TFG, nos permite saber, por ejemplo, cuándo se crea un nuevo hilo, cuándo termina o cuándo se bloquea, de modo que podemos monitorizar su consumo energético junto a otras métricas monitorizadas por los contadores hardware del procesador. La información se expone al usuario y al sistema operativo en forma de *contadores virtuales* de PMCTrack, que permiten representar información de monitorización hardware no accesible mediante la interfaz asociada a los contadores de monitorización del rendimiento del procesador (*Performance Monitoring Counters* - PMCs). El código fuente del módulo de monitorización desarrollado en este TFG ha sido incorporado al repositorio oficial de PMCTrack [17]. Esto permite satisfacer la petición

de la comunidad de usuarios de PMCTrack, que había mostrado interés por la inclusión en la herramienta del soporte necesario para interactuar con el dispositivo Odroid Smart Power 2 [18].

Estudios previos han demostrado que las métricas EEF (*Energy-Efficiency Factor*) [14] y SF (*Speedup Factor*) [12] asociadas a cada aplicación (y definidas en el capítulo 4 de esta memoria) son clave a la hora de hacer optimizaciones en la planificación de procesos para sistemas AMP. Lamentablemente, la mayorîa de plataformas de propósito general no cuentan con el hardware necesario para determinar el valor de estas métricas por medición directa (p.ej., con contadores hardware o registros de consumo de energía) para aplicaciones individuales en una carga de trabajo multiprogramada. En particular, para calcular el EEF es necesario saber lo que está consumiendo cada *core* mientras se ejecuta una aplicación, y cuál es la contribución de una aplicación de la carga al consumo de energía de otros componentes clave del sistema, como la memoria o la jerarquía caché. Por este motivo, y como caso de uso del soporte para Odroid SmartPower2 desarrollado en el TFG, hemos analizado el comportamiento de los *benchmarks* SPEC CPU 2000 y 2006, comparando su rendimiento con la eficiencia energética al ejecutarlos en los distintos tipos de *core* de un AMP (placa de desarrollo Odroid XU4) usando Odroid SmartPower2 mediante PMCTrack. Utilizando técnicas de *machine learning*, hemos podido concluir que se pueden realizar estimaciones del *SF* y el *EEF* de una aplicación a partir de otras métricas recabadas en tiempo real, sin tener acceso a medidas de consumo de energía desglosadas por aplicación. Estas métricas clave, como el número de fallos de caché por cada mil instrucciones o el número de instrucciones retiradas por ciclo, se pueden obtener utilizando los contadores hardware de monitorización del rendimiento que integra cada *core* del procesador.

Gracias a esto, podemos decidir en qué tipo de *core* de un AMP es mejor ejecutar cada tipo de aplicación para optimizar el uso que se hace de la energîa consumida en proporción al rendimiento ganado. Tan sólo tenemos que observar cómo se comportan las aplicaciones en los diferentes *cores* leyendo de los contadores hardware de los que disponemos.

6.2 Trabajo futuro

Como ya se ha mencionado a lo largo de esta memoria, las medidas de consumo energético pueden ser útiles para mejorar la eficiencia energética en sistemas multicore asimétricos. De hecho, ya se han realizado propuestas de algoritmos de planificación de procesos que optimizan la eficiencia energética de los AMP, pero sin embargo perjudicaban la justicia u otros parámetros [14].

Proponemos como trabajo futuro diseñar un algoritmo de planificación para Linux que sea capaz de priorizar distintos objetivos de optimización en base a las características de la carga de trabajo y otros factores. Dicho algoritmo podrîa ser configurado por el usuario o por el sistema operativo de forma automática según las condiciones globales del sistema. Por ejemplo, en el caso de los dispositivos móviles, el planificador del sistema operativo podrîa dar más peso a la optimización de la eficiencia energética cuando quede poca baterîa, o en general, cuando no se ejecute ninguna tarea en primer plano. Por el contrario, podrîa elegir priorizar la justicia o el rendimiento global si hay muchos hilos en ejecución. También podrîa ser capaz de

estimar en tiempo de ejecución la pérdida de rendimiento si priorizara el consumo energético en función a los programas que se están ejecutando y acúar en función de si la pérdida es alta o baja en ese determinado momento.

El método más común que se utiliza actualmente en Linux para ahorrar energîa en los procesadores de forma dinámica es ajustar la frecuencia. Esto puede conllevar una pérdida de rendimiento significativa. Con un método como el que proponemos se podrîa conseguir un modo de ahorro de energîa adicional sin degradar demasiado el rendimiento.

Apéndice A

Introduction

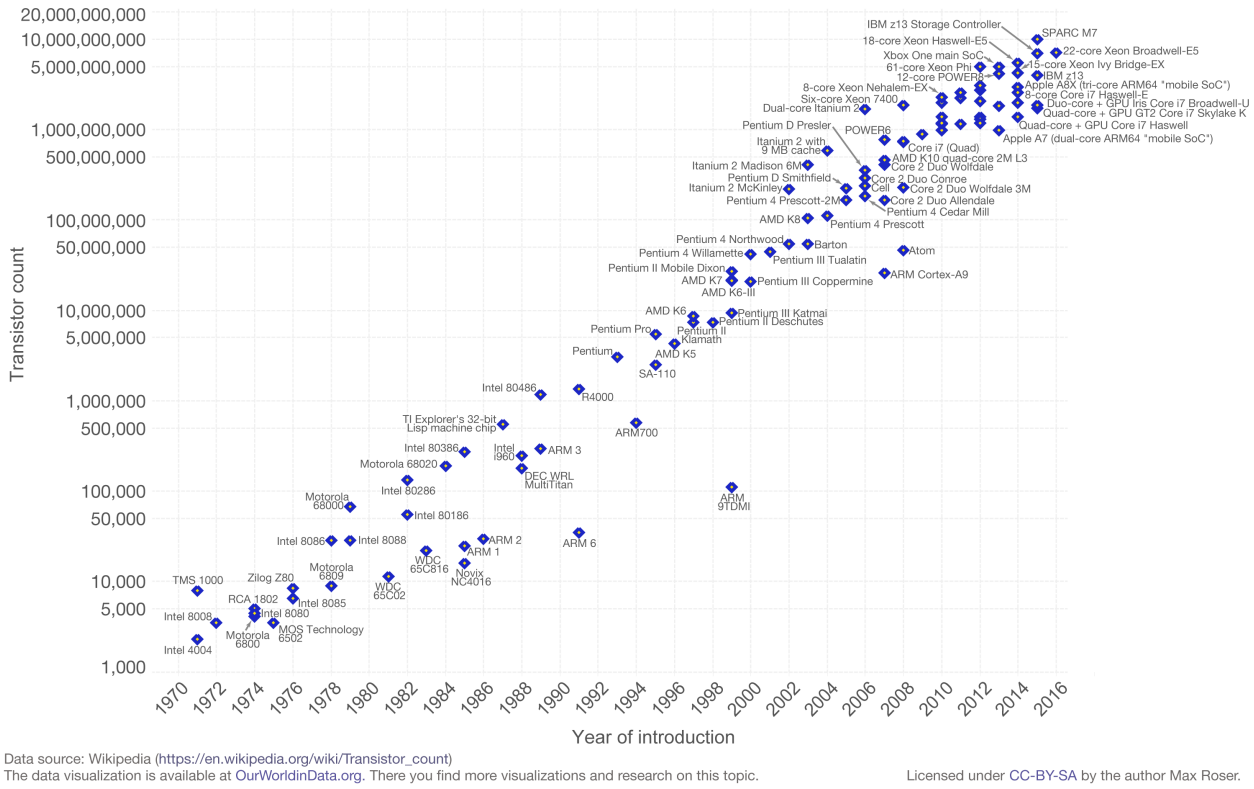
Not only has energy efficiency become a big dilemma of the current times, but also a challenge to hardware producers. The energy efficiency of a system or a device compared to another is the difference between their consumption while being under the same execution scenario. This means a system is more energetically efficient than another when, for example, the first one consumes less energy than the second one while executing the same application.

Nowadays, an efficient system meets three goals: protecting the environment, being cheaper (not on the prices of purchase, but on its maintenance), and having higher autonomy, when referring to mobile devices. This last goal can be appreciated not only on phones or tablets but also on devices such as electric cars.

Technology has experienced an exponential evolution during the last decades due to the search for increasingly powerful systems. To accomplish this, the producers always opted to increase processors' frequency and reduce its size. Also, from its original proposal in 1965, this increase has followed Moore's Law (figure A.1), according to which it was predicted that the number on transistors of a microprocessor would be doubled every two years.

This progress has encountered multiple problems throughout its history. One of the most important has been the steady rise of processors power consumption, as well as the issues with thermal dissipation. This trend suggested that the increase of the processors frequency over time would encounter a barrier difficult to break down. This forced producers to search for new development techniques that improved the processors performance without increasing their frequency. From this idea, what is today a widespread approach was born: the exploitation of parallelism between processors by taking advantage of the continuity of Moore's law thus increasing the number of transistors without raising frequency. Parallelism between different processors consists in including more processing elements, also known as *cores*, on the same microchip. This enhancement allows the parallelism to be exploited on a thread level (*Thread Level Parallelism*). These processors that have more than one process unit are known as multicore processors.

The next step beyond conventional or symmetric multicore processors (SMP) were the asymmetric multicore processors or AMP [2]. Unlike the formers, in which all the *cores* where



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Figura A.1: Moore's law Diagram [1].

identical, AMP systems combine high performance *cores* with low energy consumption *cores* that have a simpler architecture. Despite their differences, all the *cores* of an AMP use the same instruction repertoire so that software development stays as simple as possible. The advantage of an AMP over a SMP relies on the fact that it enables to improve significantly the energy efficiency of a system by only sacrificing a little bit of performance [2].

High performance *cores* are designed to execute workloads faster by having a more complex architecture, at the expense of a higher energy consumption and a higher heat production. On the other hand, low consumption *cores* have a simpler architecture and a lower performance, so this type of core consumes less energy. For this reason, low consumption *cores* are ideal for executing workloads in which there is a larger number of pipeline stops, like those more memory-intensive, as they consume less while they are waiting, unlike other workloads that make a larger use of the processor and therefore they optimize resources and extra consumption better. The hardware on its own is not aware of these benefits, so it needs help from the operating system to make the assignment of the different tasks. This distinction is named ILP specialisation, and several scheduling algorithms have been proposed to improve performance by exploiting these benefits [3].

Groups of cores with the same characteristics are called clusters. There are usually two *clusters*. One example of the organization of the *cores* on an AMP and the shared cache memory can be seen in the figure A.2. The model shown is a *heterogeneous multi-processing* model, in which all the *cores* can be used simultaneously. This is the most powerful model, as it allows the

execution of workloads on the different *cores* considering the necessities of the system on each moment. It is also worth mentioning that there are other two processing models: *clustered switching*, which is the most simple, in which only one of the clusters is active, according to global necessities of the system; and the *in-Kernel switching* model, in which the cores are paired *big* and *LITTLE* and only one is active according to the necessities of the workload they are executing. This last model is more restrictive than the *heterogeneous multi-processing* model, which allows fewer software level optimizations.

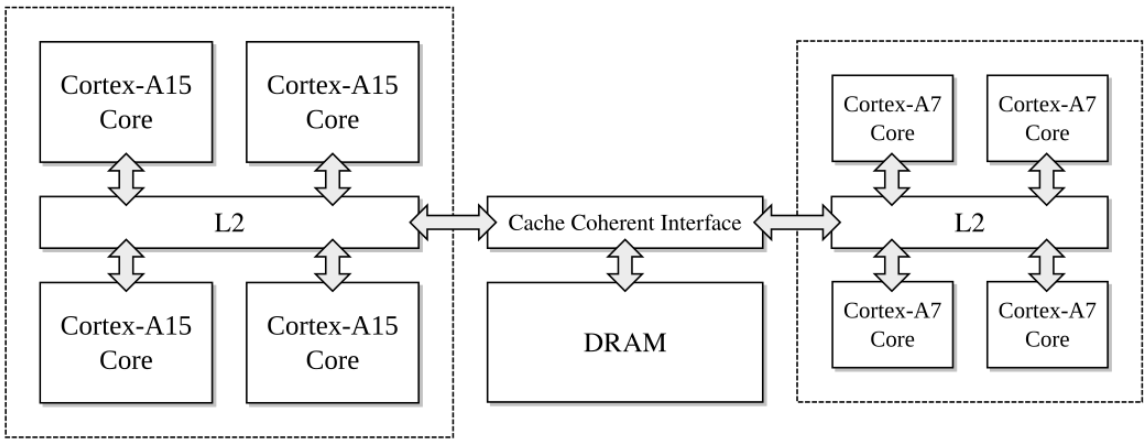


Figura A.2: Cache and cores organization example.

ARM implements an AMP system with its big.LITTLE architecture. Nowadays, a great variety of processors with this architecture is available on the market, like the case of the processor in the ARM Juno development board. This board has a processor divided in two clusters: the big cluster that has two ARM Cortex-A57 cores, and the LITTLE cluster that contains four low consumption ARM Cortex-A53 cores. Furthermore, this board contains specific hardware that allows the operating system to directly obtain power consumption metrics.

As opposed to the high cost of ARM Juno (around 6000$), it should be appreciated the existence of cheaper devices that also include an AMP architecture. This is the case of Odroid XU4 (50$ on the producers website), that however lacks the integrated software to measure its power consumption. For this reason, we are proposing to supply this hardware lack and develop a method to measure it externally by using the power monitor device Odroid SmartPower2 of the same manufacturer. The characteristics of both components are detailed on chapter 3.

A.1 Goals

The main goal of this end-of-degree project is to give the Odroid XU4 device access to the measurements of power consumption in real time, so that (existing or future) processes scheduling algorithms can use this information.

Later, as an use case of our implementation, experimental tests have been performed for a characterization of the performance and the energy efficiency of the SPEC CPU 2000 and 2006 benchmarks.

To accomplish this we have required a tool to obtain the measurements from an external source (Odroid SmartPower2) when running the different applications on each of the different types of core of our platform. We decided to use the monitoring tool PMCTrack, as it enables to integrate consumption values along with other performance measurements that processors hardware counters already provide. This would simplify the comparison of metrics, as it is all on the same tool. Moreover, PMCTrack shows the consumption metrics to the user and the operating system.

In summary, the goal of this project is to expose the consumption measurements, measured from the Odroid SmartPower2, to the operating system through PMCTrack, and additionally, record experimental consumption measurements on the Odroid XU4 to later analyze them.

A.2 Workplan

An initial meeting took place at the begining of the course 2018-2019, in which the general aspects of this project were discussed, its main goals (listed in the previous section), and what were the steps to follow, so that the project could be elaborated in a progressive and scheduled manner. Likewise, we stablished several methods of communication, some working repositories for storing and reviewing documentation were opened and a Trello project was created [4] so that the progresses made individually and as a group could be tracked.

Finally, with the tutor's help, we made an initial distribution of the project's tasks so that the different members of the team could start working on the first phase on parallel. The tasks of the final phase of the project were done together due to its higher complexity.

For the development phases on each of the boards, we had to split the time each member had to work on the boards. Once the software was stable (the probability of shutting down was not high), we left the whole system set and accessible though the network, to continue with the experiments without having to commute.

The tasks considered in the agreed planning were as follows:

1. Research of the PMCTrack tool, as it is the core of this project.
2. Detailed analysis of the Firmware of the Odroid SmartPower2 board.
3. Modification of the Odroid SmartPower2 firmware to adapt it to the necessities of this project.
4. Development of a driver for the Odroid SmartPower2 to use it with the PMCTrack tool.
5. Execution of the SPEC CPU 2000 and 2006 benchmarks with PMCTrack.
6. Processing of the obtained data to generate metrics and graphics.
7. Analysis of the results using the mentioned metrics to characterize performance and energy efficiency of the benchmarks.

A.3 Document structure

This document is organised on the following chapters:

- **Chapter 2** presents the PMCTrack tool, its behaviour, and its characteristics. A whole chapter has been dedicated to this since later it will be critical to understand several aspects of this tool.
- **Chapter 3** explains briefly the behaviour of the tools used for the extraction of the metrics, how they work together, and the implementation of the software needed.
- **Chapter 4** describes the methods and techniques used for the extraction of the consumption metrics when the benchmarks are run.
- **Chapter 5** shows the result obtained from the experiments done on the environment and the analysis that was carried out on these results.
- **Chapter 6** exposes the conclusions of this project. It also mentions potential applications and advances that could be developed in the future.
- Finally, the introduction and conclusions of this project are included, both translated into English. They are followed by the contributions of each participant and, finally, the bibliography.

Apéndice B

Conclusions

B.1 Conclusions

Previous projects have proved that Asymmetric Multicore Processors or AMPs contribute to improve energy efficiency substatially, as long as the operative system's scheduler takes into account the different characteristics of the different applications of a workload when assigning them to the different types of core [12, p. [@cj2018]]. This type of processors is more and more present on commercial mobile devices, which makes them nowadays very relevant. The ARM big.LITTLE processor is probably the most extended asymmetric multicore processor on the market.

In order to ease the design and the experimental analysis of efficient scheduling algorithms on the Odroid XU4 board - equipped with a big.LITTLE asymmetric multicore processor- we have developed on this end-of-degree project the neccesary support for the PMCTrack tool [6] to obtain the energy consumption on Linux using the power monitor Odroid SmartPower2 [8]. Thanks to this device, we can access energy consumption measurements in real time while different applications are running on different cores of the platform. This data is accessible from user mode, through PMCTrack's graphic interface and its command line tool, as well as from any component of the operating system, including the process scheduler. The escalation capacities of the PMCTrack tool have made this possible, as it allows to extend its functionalities through *plugins* known as *monitoring modules*. The API, which is accessible from the monitoring modules, one of which was necessary to develop as part of this project, allows us to know, for example, when a new thread is created, when it ends, or when it is blocked, so that we can monitor their power consumption along with other metrics being monitorized by the processor hardware counters. This information is exposed to the user and system through PMCTrack's *virtual counters* allowing the representation of the hardware-monitored information that it is not accessible through the interface associated to the Performance Monitoring Counters - PMCs. The source code of the monitoring module developed in this project has been incorporated to the official PMCTrack repository [17]. This fulfills the demand expressed by the community of PMCTrack users, that had shown interest for the inclusion of the required support for the interaction with the Odroid SmartPower2 device into PMCTrack [18].

Previous studies have provided evidence that the EEF (*Energy-Efficiency Factor*) [14] and SF (*Speedup Factor*) [12] metrics associated to each application (defined on chapter 4 of this document) are key to optimize the scheduling of processes on AMP systems. Unfortunately, most general purpose platforms do not have the required hardware to determine the value of these metrics by direct measurements (for example, using hardware counters or energy consumption registries) of individual applications on a multi-programmed workload. In particular, calculating EEF requires to know each core's consumption while an application is running, and which is the contribution of an application to the workload of energy consumption of other key components of the system, like the memory and the cache hierarchy. For this reason, and as a use case for the developed support for Odroid SmartPower2, we have analysed the behaviour of the SPEC CPU 2000 and 2006 benchmarks, comparing their performance with their energy efficiency when running on different types of core of an AMP (Odroid XU4 development board) by using Odroid SmartPower2 through PMCTrack. With the help of machine learning techniques, we have concluded that SF and EEF of an application can be estimated by using other metrics measured in real time, without having access to energy consumption measures for each application. These key metrics, like the number of cache misses per thousand instructions or the number of instructions retired per cicle, can be obtained using the performance monitoring hardware counters that are integrated into each core of the processor.

Thus, we can decide in which type of core of an AMP is better to execute each type of application to optimize the energy usage in respect with gained performance. We just need to observe how applications behave on the different cores reading from the hardware counters that we have.

B.2 Future Work

As it has been mentioned throughout this report, energy consumption measurements can be useful to enhance the energy efficiency on asymmetric multicore systems. In fact, scheduling algorithms have already been proposed to optimize energy efficiency in AMPs, but in contrast they could damage justice or other parameters [14].

We propose as future work the design of a scheduling algorithm for Linux that is able to prioritize different optimization objectives based on the characteristics of the workload and other factors. Said algorithm could be manually configured by the user, or automatically by the operating system according to the global conditions of the system. For example, in the case of mobile devices, the system's scheduler could choose to grant higher priority to the optimization of energy efficiency when it were low on battery, or in general, when there were no tasks on the foreground. On the other hand, it could choose to prioritize justice or global performance if there were many threads running. It could also be able to estimate during execution time the loss in performance if it prioritized energy consumption based on the programs that were running and would act based on whether the loss was high or low in that given moment.

The most common method currently used on Linux to save energy on the processors dynamically is to adjust the frequency. This can imply a significant loss in performance. With a

method as the one we are proposing, we could achieve an additional energy-saving method without degrading too much the performance.

Apéndice C

Contribuciones de cada participante

En este apéndice enumeramos las aportaciones al proyecto de cada miembro del equipo de trabajo. Algunas tareas del proyecto se han dividido en subtareas (realizadas por los distintos miembros) y pueden aparecer en ambas partes, y otras se han realizado de forma totalmente conjunta con lo que no se incluyen aquí, como es el caso del análisis experimental descrito en el capítulo 5.

C.1 Contribuciones de Andrés Plaza Hernando

Las contribuciones de Andrés se han enfocado más hacia la familiarización con el hardware que se iba a utilizar, y a la modificación del firmware de una de las placas. Esto se debió a que al dividir las tareas, el tutor recomendó a Germán para la parte del desarrollo del módulo ya que estaba más familiarizado con el entorno inicialmente. Una vez completada por ambas partes la primera fase en paralelo, las siguientes fases del trabajo se hicieron en su gran mayoría de manera conjunta. A continuación se listan las tareas concretas realizadas por Andrés.

C.1.1 Placa Odroid XU4

En primer lugar Andrés comenzó disponiendo de la placa. Se dedicó a estudiar su funcionamiento, y cómo acceder a ella fácil y rápidamente. Aunque había varias formas de conectarse a la misma, se optó por la conexión por puerto serie, para empezar. Al no haber utilizado nunca una placa de desarrollo, se tardó más de lo esperado en familiarizarse con la placa. Más tarde se utilizó la conexión por ssh a la placa ya que no solo era mas rapida, y más robusta, además, tenía la comodidad de poder realizar conexiones a distancia, lo que nos evitaba largos desplazamientos.

C.1.2 Firmware Odroid SmartPower2

Andrés Plaza tuvo que estudiar el código del firmware de la Odroid SmartPower 2 sobretodo centrado en la frecuencia de muestreo. Él dispuso primero placa SmartPower para poder

realizar pruebas directamente sobre el hardware. Su objetivo era entender exactamente cómo funcionaba el hardware con el que se iba a trabajar, por lo que tuvo que realizar una investigación muy detallada de la placa.

Una vez finalizada la investigación sobre la SmartPower 2 tuvo modificar el firmware de la SmartPower 2 para ajustar la frecuencia de muestreo de 1000 ms a la mitad (500ms). Para poder realizar pruebas ya que la entrada se recibe por un puerto serie,utilizó la herramienta *Minicom* que permite las lecturas por puerto serie y ajustar ciertos parámetros para que la lectura sea posible. Más tarde dado que se encontraron problemas con la escasez de datos generados por la placa tuvo que volver a modificar la frecuencia de muestreo a 100 ms. Esta escasez se debía al buffer que consumía los datos de la SmartPower 2 para proporcionarlos a PMCTrack, sufría inanición lo que nos generaba una gran cantidad de valores nulos.

C.1.3 Generación de gráficas con Pandas

Una vez estaban realizadas y extraídas las métricas correspondientes a los distintos benchmark. Después de que Germán calculase y dibujase las gráficas correspondientes a las SF y EEF medios, Andrés se encargó de realizar los cálculos necesarios para poder medir la evolución en tiempo real del EEF, el SF los fallos de cache de último nivel y los accesos a cache de último nivel. Tras esto, con la herramienta pandas, escribió un programa que leía los archivos de cada benchmark y dibujaba las distintas gráficas de cada aplicación. Con la ayuda de Germán se perfeccionó el programa ya que tenía ciertos fallos. De forma conjunta se mejoró el programa y se decidió en qué métricas centrar el análisis del capítulo 5.

C.1.4 Generación de gráficas de métricas adicionales

Más tarde, referido a la anterior sección, Andrés tuvo que calcular nuevas métricas para generar más gráficas del EEF, fallos de caché de primer nivel y accesos a la TLB. Esto se debe a que tras una reunión con el director del proyecto y con Germán se decidió que eran necesarias para el capítulo 5, para generar más evidencias de los efectos en el EEF de otras operaciones de accesos a memoria. Ya que el estudio busca encontrar qué factores afectan a la eficiencia y solo estábamos considerando un espectro muy reducido.

C.1.5 Traducciones al inglés

Andrés ha sido el encargado de realizar gran parte de las traducciones requeridas en inglés de la memoria. El motivo de escoger realizar la mayoría ha sido que se dedica a hacer traducciones de artículos en su actual trabajo, y aunque su nivel de inglés es muy alto, quería poder practicar con lenguaje técnico, y además contar con un largo texto que traducir.

C.1.6 Generación de gráficas con WEKA

En la parte final del análisis cuando todos los datos estaban ya procesados y ordenados, Andrés se ha dedicado al estudio de la herramienta de *machine learning* **WEKA**. Con ayuda del tutor el alumno aprendió a utilizar esta herramienta y generar predicciones de SF y EEF en base a diferentes métricas. Con esta herramienta se pueden generar ficheros de datos de estas

predicciones. Que más tarde se pueden utilizar, usando la herramienta **Pandas**, para generar las gráficas de predicción que aparecen en en la sección 5.3. En primer lugar se realizaron cuatro predicciones. El valor de SF y del EEF en cada tipo de *core*, utilizando como datos de aprendizaje los fallos y accesos a la LLC por ciclo y por instrucción. Más tarde con el fin de incluir más variables a tener en cuenta se utilizaron también los fallos y accesos a las LL1 y a la TLB. Los resultado obtenidos así como los logs de los experimentos realizados por la aplicación, fueron guardados en la ruta del proyecto.

C.2 Contribuciones de Germán Franco Dorca

Las contribuciones de Germán están más enfocadas al desarrollo de las herramientas y la preparación y configuración de gran parte de los entornos de desarrollo, así como la documentación requerida para ello. Asumió esta parte del trabajo porque estaba más familiarizado con muchas de ellas. Asimismo, era una forma eficiente y equitativa de dividir la carga de trabajo de forma que ambas partes pudieran avanzar en paralelo, aunque fue fundamental la colaboración y apoyo mutuo en muchas fases. A continuación se listan y describen las contribuciones concretas de Germán.

C.2.1 Documentación PMCTrack

Germán Franco se encargó de estudiar el código fuente de PMCTrack y la API de los módulos de monitorización mediante el análisis de los módulos existentes. Se trataba de un requisito previo necesario para el correcto desarrollo de las tareas que involucran PMCTrack. Puesto que se había acordado que Germán se encargaría de la mayoría de lo relacionado con PMCTrack, resultaba más eficiente que asumiera este trabajo de documentación. Tras ello, estos conocimientos se pusieron en común para que Andrés también pudiera realizar algunas tareas relacionadas con PMCTrack, como la ejecución de los *benchmarks*.

C.2.2 Módulo de monitorización de PMCTrack

Al comienzo del proyecto, en la repartición de tareas se acordó que Andrés se encargaría de la adaptación del firmware de Odroid SmartPower2, mientras que Germán desarrollaba el módulo de monitorización para la placa Odroid XU4. Para probar las modificaciones en el *firmware* eran necesarios los dos dispositivos. Para poder trabajar de forma paralela en el desarrollo del módulo de monitorización, se procedió a instalar PMCTrack en una máquina virtual con Debian en un equipo, siendo para ello necesario recompilar el kernel Linux con el parche de PMCTrack. Primero se desarrollaron las partes básicas con datos de prueba, y una vez estuvo lista la base del módulo y se tuvo acceso a las placas de desarrollo, se añadió la funcionalidad restante integrando los datos provenientes por el puerto serie.

C.2.3 Toma de las medidas de consumo en reposo

Tras implementar el módulo de monitorización y disponer de acceso a todo el entorno de trabajo plenamente funcional, se procedió a realizar las mediciones de consumo energético

del sistema en reposo, configurando los *cores* para tomar las muestras con las distintas configuraciones de frecuencias. La toma de las medidas en sí fue sencilla porque sólo había que ejecutar un comando que ya conocíamos por las etapas previas. Sin embargo, para tomar estas medidas fue preciso configurar el ventilador de la placa para que estuviera siempre encendido. Para ello fue necesaria una breve fase de investigación. También se hicieron mediciones con el ventilador en automático para demostrar la existencia de picos de consumo cuando el ventilador se activaba. Para ello se compararon ambos resultados en una gráfica.

C.2.4 Procesamiento de los datos y generación de gráficas con gnuplot

Una vez que obtuvimos los datos de rendimiento y eficiencia energética para cada *benchmark* y *core*, fue necesario ordenarlos y procesarlos para poder operar con ellos y construir las gráficas. Inicialmente se crearon dos scripts en Python (para los resultados de consumo en reposo y de los *benchmarks*) que leían la salida en bruto de PMCTrack almacenada previamente en ficheros de texto, operaban con los datos y generaban ficheros *.dat*.

A partir de los datos en esos ficheros se construyeron las gráficas de los valores medios de EEF y SF. Para ello se utilizó la herramienta *gnuplot*, sobre la cual fue necesario documentarse previamente.

C.2.5 Colaboración en la generación de gráficas con Pandas

Para la realización del análisis se hicieron mejoras en el código de la generación de las gráficas de las métricas a lo largo del tiempo de los *benchmarks* que había creado Andrés, para que el código fuera más escalable y poder visualizar distintas gráficas de forma rápida según las necesidades del análisis. También se hicieron mejoras visuales para que los datos pudieran visualizarse de manera correcta y más intuitiva.

C.2.6 Compilación y configuración de la plantilla de la memoria

Para escribir esta memoria se ha utilizado una plantilla para *pandoc* que nos permite escribir en Markdown y LaTeX al mismo tiempo, y genera el PDF final automáticamente. La plantilla requería una configuración previa mediante parámetros de configuración y la modificación de algunas partes estáticas para adaptarla a las necesidades específicas de este proyecto; así como la corrección de algunos errores, a los que a la larga se ha dedicado un tiempo considerable, pues requieren cierto trabajo de documentación debido al no tener conocimientos previos de LaTeX. A pesar de todo, la plantilla y las instrucciones de uso nos han facilitado mucho el trabajo de redactar esta memoria y han contribuido considerablemente a mejorar su calidad estética.

C.2.7 Traducciones a inglés

Las traducciones se dividieron entre los miembros para repartir el trabajo y cada uno revisó la parte del otro. Germán se encargó de la traducción del resumen y la revisión de la traducción

de la introducción, las conclusiones y los apéndices.

C.2.8 Publicación de las contribuciones a PMCTrack

Dado que algunos miembros de la comunidad de PMCTrack habían expresado su interés en el soporte para Odroid SmartPower2, se ha incorporado el módulo de monitorización desarrollado al código de PMCTrack en el repositorio oficial de GitHub [17].

Bibliografía

[1] M. Roser, «Ley de Moore, Wikipedia». https://en.wikipedia.org/wiki/Moore%27s_law.

[2] R. Kumar y others, «Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction», en *Proc. of MICRO 36*, 2003.

[3] J. C. Sáez, M. Prieto, A. Pousa, y A. Fedorova, «Explotación de Técnicas de Especialización de Cores para Planificación Eficiente en Procesadores Multicore Asimétricos».

[4] «Trello». https://trello.com/.

[5] PMCTrack, «project official website». http://pmctrack.dacya.ucm.es/.

[6] J. C. Sáez y otros, «PMCTrack: Delivering Performance Monitoring Counter Support to the OS Scheduler», *The Computer Journal*, vol. 60, n.º 1, pp. 60-85, 2017.

[7] Hardkernel, «Odroid XU4 wiki». https://wiki.odroid.com/odroid-xu4/odroid-xu4.

[8] Hardkernel, «Odroid SmartPower2 wiki». https://wiki.odroid.com/accessory/power_supply_battery/smartpower2.

[9] A. G. García y Álvaro Sanz del Río, «Soporte de sistema operativo para ahorro de energía en plataformas móviles con procesadores multicore asimétricos», 2016.

[10] Hardkernel, «Odroid SmartPower2 wiki». https://www.odroid.com/dokuwiki/doku.php?id=en:odroidsmartpower.

[11] J. C. Saez, A. Pousa, A. E. de Giusti, y M. Prieto-Matias, «On the Interplay Between Throughput, Fairness and Energy Efficiency on Asymmetric Multicore Processors», *The Computer Journal*, vol. 61, n.º 1, pp. 74-94, 2018.

[12] J. C. Saez, A. Fedorova, D. Koufaty, y M. Prieto, «Leveraging Core Specialization via OS Scheduling to Improve Performance on Asymmetric Multicore Systems», *ACM Trans. Comput. Syst.*, vol. 30, n.º 2, pp. 6:1-6:38, abr. 2012.

[13] J. C. Saez, A. Pousa, F. Castro, D. Chaver, y M. Prieto-Matias, «Towards completely fair scheduling on asymmetric single-ISA multicore processors», *Journal of Parallel and Distributed Computing*, vol. 102, pp. 115-131, 2017.

[14] J. C. Saez, A. Pousa, A. E. de Giusti, y M. Prieto-Matias, «On the Interplay Between Throughput, Fairness and Energy Efficiency on Asymmetric Multicore Processors», *The Computer Journal*, vol. 61, n.º 1, pp. 74-94, abr. 2017.

[15] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, y I. H. Witten, «The WEKA data mining software: an update», *SIGKDD Explor. Newsl.*, vol. 11, n.º 1, pp. 10-18, 2009.

[16] J. H. Friedman, «Stochastic Gradient Boosting. 1999», *http://statweb.stanford.edu/~jhf/ftp/stobst.pdf*.

[17] PMCTrack, «Source code repository at Github». https://github.com/jcsaezal/pmctrack, 2015.

[18] «Odroid SmartPower2 request for PMCTrack». https://github.com/jcsaezal/pmctrack/issues/3.