Herramienta de gestión de contadores hardware para Android

TRABAJO FIN DE GRADO

Luis Javier Cabrera Sagbay
Youness El Guennouni

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Junio 2016

Herramienta de gestión de contadores hardware para Android

Memoria de Trabajo Fin de Grado

Luis Javier Cabrera Sagbay
Youness El Guennouni

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Junio 2016

Documento maquetado con TEXIS v.1.0.

Este documento está preparado para ser imprimido a doble cara.

Autorización de difusión y utilización

Los abajo firmantes, alumnos y tutor del Trabajo Fin de Grado (TFG) en el Grado en Ingeniería Informática de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el Trabajo Fin de Grado (TF) cuyos datos se detallan a continuación. Así mismo autorizan a la Universidad Complutense de Madrid a que sea depositado en acceso abierto en el repositorio institucional con el objeto de incrementar la difusión, uso e impacto del TFG en Internet y garantizar su preservación y acceso a largo plazo.

TÍTULO del TFG: "Herramienta de gestión de contadores hardware para Android "

Curso académico: 2015/2016

Nombre de los Alumnos: Luis Javier Cabrera Sagbay y Youness El Guennouni

Tutor del TFG y departamento al que pertenece: Juan Carlos Sáez Alcaide en el Departamento de Arquitectura de Computadores y Automática.

Juan Carlos Sáez Alcaide

Luis Javier Cabrera Sagbay

Youness El Guennouni

Madrid, a 15 de junio de 2016.

"Tell me and I forget.
Teach me and I remember.
Involve me and I learn."
– Benjamin Franklin

Agradecimientos

Queremos agradecer en primer lugar a nuestro director de proyecto Juan Carlos Sáez Alcaide, por compartir sus conocimientos con nosotros, por su infinita paciencia, por su motivación y por su cercanía. Siempre ha estado disponible para todas las dudas que hemos tenido durante la realización de este proyecto proporcionándonos todos los medios para asegurar la realización de un proyecto de calidad.

Queremos agradecer a todos nuestros profesores, ellos nos enseñaron a valorar los estudios y a superarnos cada día.

Queremos agradecer a nuestros padres y familiares porque ellos estuvieron en los días más difíciles de nuestras vidas dando todo el esfuerzo para culminar esta etapa. Con el esfuerzo de ellos y nuestro esfuerzo ahora podemos ser unos grandes profesionales y seremos un gran orgullo para ellos.

Por último, queremos agradecer a nuestros amigos que han estado con nosotros en las buenas y en las malas y que nunca dejaron de apoyarnos.

Muchas gracias a todos los que confiaron en nosotros.

Resumen

PMCTrack es una herramienta de código abierto para Linux que permite monitorizar el rendimiento de las aplicaciones haciendo uso de los contadores hardware del procesador. Esta herramienta soporta la captura de métricas como el número de instrucciones por ciclo o la tasa de fallos de cache.

El objetivo de este proyecto es portar PMCTrack al sistema operativo Android sobre plataformas que integran procesadores de ARM. Esto conlleva la realización de las siguientes tareas: (1) modificación de la variante del kernel Linux propia de Android para incluir las extensiones requeridas por el módulo del kernel de PMCTrack, (2) adaptación de las herramientas de modo usuario de PMCTrack, y (3) desarrollo de una aplicación Android que permita visualizar en tiempo real las medidas de los contadores recabadas para las distintas aplicaciones que están siendo monitorizadas.

Para poner a prueba la adaptación de la herramienta PMCTrack al sistema operativo Android y mostrar la utilidad de nuestras aportaciones, se han llevado a cabo diversos casos de estudio empleando la placa de desarrollo Odroid XU4.

Palabras clave: Contadores hardware, Kernel Android, Aplicaciones multihilo, Monitorización de la memoria cache.

Abstract

PMCTrack is an open-source tool for Linux that enables the gathering of performance data from applications using hardware performance counters. This tool makes it possible for the user to monitor high-level performance metrics, such as the number of instructions per cycle or the cache miss rate.

The main goal of this project is to create a fully functional version of PMCTrack for the Android operating system running on platforms equipped with ARM processors. This comes down to the following three tasks: (1) modification of the variant of the Linux kernel for Android, to include the necessary extensions required by the PMCTrack's kernel module to function, (2) porting the user-level tools in PMCTrack to Android, and (3) development of an Android app that extends PMCTrack with capabilities of real-time visualization of the various high-level metrics monitored for an application using hardware counters.

To demonstrate the effectiveness of our contributions and to test the functionality of the Android port for PMCTrack created in this project, we analyze several case studies by using the Odroid XU4 development board.

Keywords: Performance monitoring counters, Android kernel, Multithreaded applications, Cache Monitoring.

Índice

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

La mayor parte de los procesadores actuales cuentan con una serie de contadores hardware para monitorización del rendimiento o **PMCs** (*Performance Monitoring Counters*). Estos contadores permiten a los usuarios monitorizar métricas de rendimiento de sus aplicaciones, tales como el número de instrucciones por ciclo (IPC) o la tasa de fallos del último nivel de cache (*last-level-cache (LLC) miss rate*). Estas métricas ayudan a identificar posibles cuellos de botella en desarrollos software, proporcionando pistas que pueden resultar muy valiosas para programadores y diseñadores de microprocesadores. Sin embargo, el acceso a estos PMCs está normalmente restringido a código que se ejecute en el nivel privilegiado reservado al sistema operativo. Para permitir el acceso a estos contadores desde el espacio del usuario es preciso implementar una herramienta a nivel de kernel, un código integrado en el propio sistema operativo o un driver, que ofrezca una interfaz de alto nivel para el usuario final [21, 5, 1].

Trabajos previos han demostrado que el planificador del sistema operativo (SO) puede beneficiarse de los datos proporcionados por los PMCs, haciendo posible la realización de sofisticadas y efectivas optimizaciones en tiempo de ejecución en sistemas multicore [7, 20, 23, 9, 24, 8, 15, 12, 17]. Se ha prestado especial atención a las optimizaciones en el planificador del sistema operativo [8, 15, 12, 17, 3, 4]. Muchos de estos algoritmos se basan en modelos de predicción. Estos modelos requieren monitorizar un conjunto específico de eventos hardware, que puede diferir sustancialmente entre modelos de procesador y arquitecturas [8, 15].

Las herramientas de dominio público que hacen uso de los PMCs permiten monitorizar el rendimiento de las aplicaciones desde el espacio de usuario, pero no proporcionan una API independiente de la arquitectura para que el propio sistema operativo pueda utilizar la información de los PMCs para tomar decisiones de planificación. Ante tal situación, algunos investigadores han recurrido al desarrollo de código *ad-hoc* específico de la arquitectura para acceder a los PMCs, usándolos para realizar implementaciones de distintas estrategias de planificación [7, 8, 19, 17]. Otros investigadores han recurrido al desarrollo de sencillos prototipos de planificador que se ejecutan en el espacio de usuario [23, 24, 12]. Sin embargo, estas aproximaciones dejan "atada" la implementación del planificador a una cierta arquitectura o modelo de procesador.

Para superar esta limitación se desarrolló PMCTrack, una herramienta de gestión de contadores hardware para el kernel Linux, pero diseñada principalmente para que el SO usará los contadores para llevar a cabo tareas internas, como la planificación de procesos. La novedad de la herramienta PMCTrack está ligada al concepto *módulo de monitorización*, una extensión específica de la arquitectura responsable de proporcionar a cualquier algoritmo de planificación del SO que aprovecha los datos de los PMCs, aquellas métricas de rendimiento necesarias para poder realizar su función. Esta abstracción permite la implementación de algoritmos de planificación del SO independientes de la arquitectura [18].

A pesar de ser una herramienta diseñada principalmente para ayudar al planificador del SO, PMCTrack también cuenta con un conjunto de herramientas de línea de comandos y componentes en el espacio de usuario. Estas herramientas ayudan a los diseñadores de algoritmos de planificación para el SO durante todo el ciclo de vida del desarrollo, complementando así a las herramientas existentes de depuración a nivel de kernel con información extraída de los PMCs. Por otra parte, dada la flexibilidad de los módulos de monitorización de PMCTrack, se puede exponer fácilmente al usuario cualquier tipo de información de monitorización proporcionada por los procesadores modernos pero que no está modelada directamente a través de contadores hardware, como el consumo de energía o el espacio que una aplicación utiliza en una cache compartida [16], mediante la abstracción de "contadores virtuales" que ofrece PMCTrack. En un trabajo previo [18] se discuten ampliamente las ventajas que PMCTrack ofrece frente a otras herramientas de monitorización del rendimiento.

El resto de este capítulo se estructura de la siguiente forma. En la sección 1.1 se presenta la arquitectura interna de PMCTrack, así como los diferentes modos de uso de la herramienta. La sección 1.2 ilustra la motivación de este Trabajo Fin de Grado. En la sección 1.3 se describe brevemente el entorno experimental empleado en el proyecto. En las secciones 1.4 y 1.5 se presentan los objetivos del proyecto y el plan de trabajo para el mismo. Finalmente, la sección 1.6 describe la estructura de la memoria.

1.1. Diseño de PMCTrack

1.1.1. Arquitectura

La figura 1.1 ilustra la arquitectura interna de PMCTrack. La herramienta consta de un conjunto de componentes en el espacio de usuario y de kernel. Esencialmente, el usuario final interactúa con PMCTrack usando las herramientas de línea de comandos disponibles o PMCTrack-GUI (interfaz gráfica). Como alternativa, las aplicaciones pueden acceder a la funcionalidad de PMCTrack directamente a través de la librería de espacio de usuario libpmctrack. Estos componentes se comunican con el módulo del kernel de PMCTrack por medio de un conjunto de entradas /proc de Linux exportadas por el módulo. El módulo del kernel implementa la gran mayoría de la funcionalidad de PMCTrack. Para recopilar datos de contadores de rendimiento por hilo, el módulo tiene que ser plenamente consciente de distintos eventos de planificación (por ejemplo,

cambios de contexto, la creación/terminación del hilo). Además de exponer los datos de contadores de rendimiento de una aplicación a las herramientas de usuario, el módulo implementa un mecanismo simple para alimentar con datos de monitorización por hilo a cualquier política de planificación que requiere información de los contadores de rendimiento para funcionar. Debido a que tanto el planificador del kernel de Linux como las clases de planificación se implementan en su totalidad en el kernel, para que el módulo del kernel de PMCTrack esté al tanto de los eventos relacionados con el ciclo de vida de los hilos y las solicitudes del planificador del SO se requieren algunas modificaciones menores en el propio kernel de Linux. Estas modificaciones, representadas en la figura 1.1 con el nombre de "PMCTrack kernel API", se encarga de enviar un conjunto de notificaciones al módulo desde el planificador. Para poder recibir estas notificaciones, el módulo del kernel de PMCTrack implementa la interfaz de operaciones `pmc_ops_t` [18].

Tal y como se ilustra en la figura 1.1, el módulo del kernel de PMCTrack consta de varios componentes. El núcleo independiente de arquitectura implementa la interfaz `pmc_ops_t`, e interactúa con la herramienta `pmctrack` de línea de comandos a través del sistema de ficheros /proc de Linux. El módulo del kernel de PMCTrack también proporciona una API para construir *módulos de monitorización*. Como se ha indicado anteriormente, el propósito principal de un módulo de monitorización es proporcionar un algoritmo de planificación que se implementa en el kernel con las métricas de rendimiento de alto nivel u otra información de tiempo de ejecución detallada que quede expuesta por el hardware, tales como consumo potencia/energía. Además, un módulo de monitorización puede exponer esta información a los componentes del espacio de usuario de PMCTrack por medio de contadores virtuales. El núcleo independiente de arquitectura en el módulo del kernel hace uso de un backend (BE) compatible con la Unidad de Monitorización de Rendimiento (conocida como PMU por sus siglas en inglés) para llevar a cabo el acceso de bajo nivel a los PMCs y para realizar la traducción de los *strings* de configuración proporcionados por el usuario a estructuras de datos internas para la plataforma en cuestión. Actualmente existen backends compatibles con la mayoría de los procesadores modernos de Intel y AMD. Recientemente se ha incorporado soporte para el coprocesador Intel Xeon Phi, y para distintos modelos de procesadores de la familia Cortex de ARM.

1.1.2. Modos de uso

PMCTrack se puede utilizar para recoger datos del contador de rendimiento del planificador del SO (utilizando una interfaz en el kernel) y desde el espacio de usuario.

1.1.2.1. Acceso a los datos de PMC desde el planificador del SO

Esta característica permite que cualquier algoritmo de planificación en el kernel (implementado como una clase de planificación) pueda obtener datos de monitorización de cada hilo, haciendo posible la toma de decisiones de planificación en función de estos datos. La activación de esta característica en un determinado hilo desde el código del planificador se reduce a la activación de un flag en el descriptor del hilo [18].

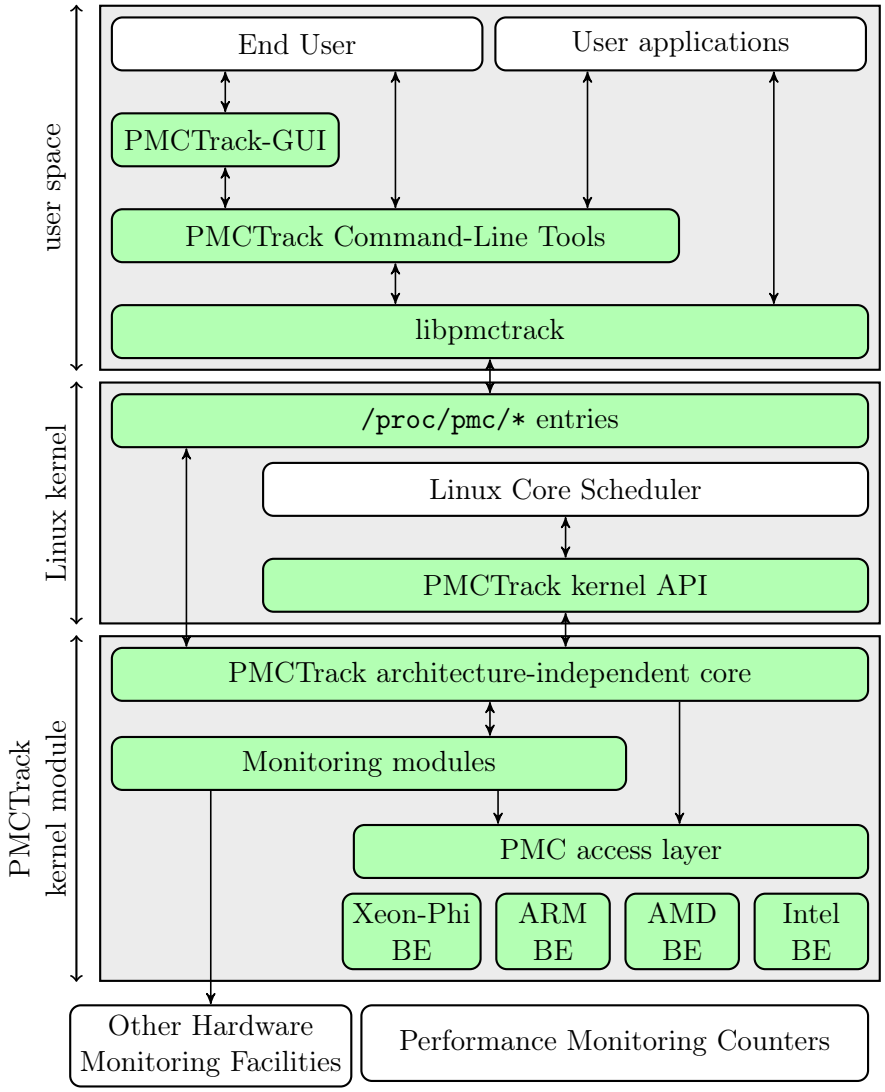


Figura 1.1: Arquitectura de PMCTrack

Para garantizar que la implementación del algoritmo de planificación que explota esta característica de uso se mantiene independiente de arquitectura, el planificador no configura los PMCs ni se ocupa de ellos directamente. En lugar de eso, uno de los módulos de monitorización se encarga de proporcionar a la clase de planificación las métricas de monitorización del rendimiento de alto nivel necesarias, tales como el número de instrucciones por ciclo (IPC) o la tasa de fallos de cache.

PMCTrack puede incluir varios módulos de monitorización que son compatibles con una plataforma dada. Sin embargo, sólo uno de ellos puede estar habilitado al mismo tiempo: el que proporciona al planificador la información de los PMCs para que pueda llevar a cabo su función. En el caso de que haya varios módulos de monitorización, el administrador del sistema puede indicar al sistema cuál de ellos usar, escribiéndolo en un fichero especial de /proc. El planificador puede comunicarse con el módulo de monitorización activo para obtener datos de monitorización de un hilo a través de la siguiente función de la API del kernel de PMCTrack:

```
int pmcs_get_current_metric_value(struct task_struct*
   task, int metric_id, uint64_t* value);
```

Por simplicidad, a cada métrica se le asigna un ID numérico, conocido por el planificador y el módulo de monitorización. Para poder obtener datos de métricas de un hilo en tiempo de ejecución, la función arriba mencionada puede invocarse desde la función de tratamiento de *tick* del planificador.

Los módulos de monitorización hacen posible que una política de planificación que está basada en el uso de contadores de rendimiento pueda ser usada en nuevas arquitecturas o nuevos modelos de procesador. Hacer esto se reduce a construir un *módulo de monitorización* o adaptar uno existente a la plataforma en cuestión. Desde el punto de vista del programador, la creación de un módulo de monitorización implica la implementación de una interfaz de operaciones. Esta interfaz se compone de varias llamadas a funciones que permiten notificar al módulo de monitorización sobre activaciones y desactivaciones solicitadas por el administrador del sistema, cambios de contexto de hilos, salidas y entradas de un hilo en el sistema (*sleep/resume*), solicitudes de valores de métricas de PMCs por parte del planificador, etcétera. El programador normalmente solo implementa el subconjunto de llamadas a funciones requeridas para llevar a cabo el proceso interno necesario. Cabe destacar que, al hacerlo, el desarrollador no tiene que acceder directamente a los registros de los PMCs. En concreto, el programador indica la configuración del contador deseado (codificado en un *string*) usando el API de desarrollo del módulo del kernel de PMCTrack. Al obtener nuevas muestras de PMCs de un hilo, se invoca una función *callback* del módulo de monitorización, pasando las muestras obtenidas como parámetro.

1.1.2.2. Uso de PMCTrack desde espacio de usuario

Además del mecanismo interno del kernel presentado anteriormente, PMCTrack también permite obtener datos PMC desde el espacio de usuario mediante el uso de la herramienta `pmctrack` de línea de comandos o `libpmctrack`.

El comando `pmctrack` permite al usuario recopilar datos de rendimiento de una aplicación en intervalos de tiempo regulares (*Time-Based Sampling* - TBS) o cuando un cierto recuento de eventos alcanza un umbral determinado (*Event-Based Sampling* - EBS). Además, el programa soporta monitorización de aplicaciones tanto multihilo como monohilo y tiene capacidades de multiplexación de eventos [18]. Para ilustrar cómo funciona la herramienta `pmctrack`, consideremos el siguiente comando de ejemplo para el modo de TBS (por defecto):

```
$ pmctrack -c instr,llc_misses ./mcf06
[Event-to-counter mappings]
pmc1=instr
pmc2=llc_misses
[Event counts]
nsample    pid      event          pmc1          pmc2
      1  11960       tick     797055120       8912054
      2  11960       tick     316689383      11242700
      3  11960       tick     282149642      10327292
      4  11960       tick     274180995      10164450
      5  11960       tick     259539536       9709397
      6  11960       tick     241565909       9274640
      7  11960       tick     233002034       9008892
      8  11960       tick     234823905       9007262
...
```

Este comando proporciona al usuario el número de instrucciones retiradas y los fallos del último nivel de memoria cache (*Last-Level Cache - LLC*) por segundo[1] en la placa Odroid XU4 (descrita en la sección 1.3), que integra un procesador big.LITTLE de ARM. Para indicar los conjuntos de eventos hardware a monitorizar, es preciso utilizar la opción -c. Como se muestra en el ejemplo, la herramienta de línea de comandos permite especificar los eventos hardware a monitorizar usando mnemotécnicos, de la misma manera que otras herramientas orientadas a espacio de usuario [5, 1, 11]. El comienzo de la salida del comando muestra el mapeo de evento a contador para los diversos eventos hardware. En la salida la sección *Event counts*, se muestra una tabla con el número de filas de los diversos eventos; cada muestra (una por segundo) está representado por una fila diferente. Al final de la línea, se especifica el comando para ejecutar la aplicación asociada que deseamos monitorizar (por ejemplo, ./mcf06).

Para complementar la herramienta de línea de comandos `pmctrack` con la visualización en tiempo real de las métricas de rendimiento de alto nivel (como el IPC o la tasa de fallos LLC). PMCTrack consta de un *front-end* gráfico, llamado PMCTrack-GUI. La figura 1.2 muestra una captura de pantalla de esta aplicación. PMCTrack-GUI amplía las capacidades básicas de las herramientas de línea de comandos con otras funcionalidades relevantes, como la monitorización remota por SSH o la posibilidad de que el usuario defina sus propias métricas de rendimiento, cuyos valores se representan de forma gráfica en tiempo real.

[1] El periodo de muestreo de los contadores puede configurarse mediante la opción -T.

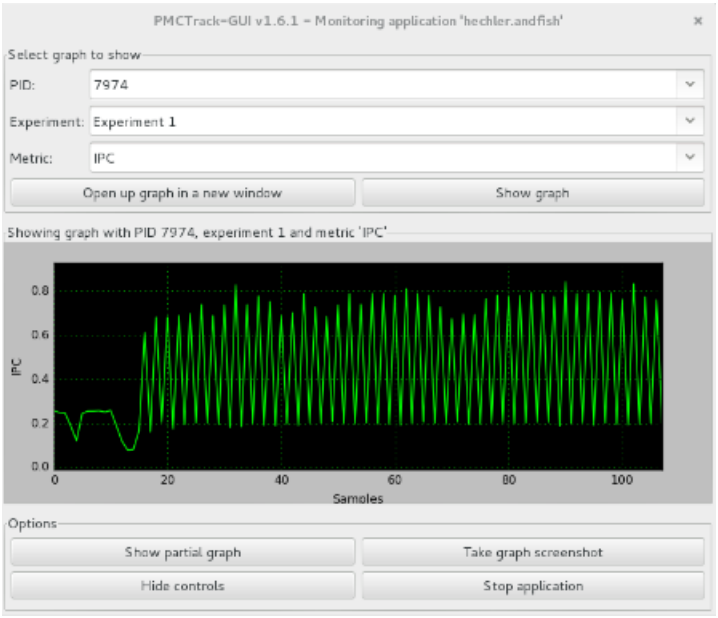


Figura 1.2: PMCTrack-GUI

`Libpmctrack` permite caracterizar el rendimiento de los fragmentos de código a través de los PMCs en programas secuenciales y multihilo. Con este fin, la API de `libpmctrack` ofrece un conjunto de llamadas para indicar la configuración PMC deseada al módulo del kernel en cualquier punto del código de la aplicación. El programador puede entonces recuperar los recuentos de eventos asociados para cualquier fragmento de código (ya sea a través de *TBS* o *EBS*) simplemente encerrando el código entre las invocaciones a las funciones `pmctrack_start_counters()` y `pmctrack_stop_counters()`.

Libpmctrack y el comando pmctrack permiten al usuario recopilar datos de PMC de la aplicación, pero también tienen la capacidad de proporcionar a los usuarios con cualquier otra información pertinente exportada por el módulo de monitorización activo como un contador virtual. Para proporcionar información de los PMCs y de los contadores virtuales a libpmctrack y al programa pmctrack, el módulo del kernel de PMCTrack almacena esta información en el espacio del kernel en *buffers* circulares. Las herramientas de modo usuario de PMCTrack consumen la información de dichos *buffers* leyendo de entradas /proc exportadas por el módulo del kernel.

1.2. Motivación

El uso masivo del sistema operativo Android (basado en el kernel Linux) en la inmensa mayoría de los dispositivos móviles actuales, hace que resulte atractivo para los usuarios finales, desarrolladores de aplicaciones y los fabricantes de dispositivos tener acceso a herramientas de monitorización del rendimiento basadas en contadores hardware para este sistema.

Actualmente, los usuarios de Android pueden acceder a la información de los contadores hardware mediante el subsistema PerfEvents [21] del kernel Linux, que tiene la herramienta `perf` (de línea de comandos) como *frontend*. No obstante, el uso de PerfEvents en Android presenta dos limitaciones importantes. En primer lugar, la herramienta `perf` de línea de comandos no constituye una buena alternativa para usuarios no experimentados. Esencialmente, el acceso a la misma requiere tener instalada una aplicación de emulación de terminal en el dispositivo móvil o establecer una conexión vía ADB al mismo (p.ej., `adb shell`). En segundo lugar, el subsistema PerfEvents del kernel no ofrece un soporte robusto ni independiente de arquitectura para acceder a información de monitorización del rendimiento (ni de consumo energético) desde distintos subsistemas del kernel, como el gestor de memoria o el planificador [18]. Esto dificulta llevar a cabo optimizaciones basadas en contadores hardware dentro del sistema, que además sean fácilmente extensibles a distintas arquitecturas o modelos de procesador [16].

La herramienta PMCTrack ha sido diseñada específicamente para hacer frente a la segunda limitación. Por otra parte, el *front-end* gráfico PMCTrack-GUI es fácilmente adaptable para realizar sesiones de monitorización en cualquier dispositivo con Android vía ADB[2]. Lamentablemente, aún no existe una versión funcional de PMCTrack para Android. Por lo tanto, aún se desconocen los beneficios potenciales de usar esta herramienta en este sistema operativo. Analizar estos beneficios es la principal motivación de este Trabajo Fin de Grado.

1.3. Entorno experimental: placa Odroid XU4

Para el desarrollo del Trabajo Fin de Grado se ha usado la placa Odroid XU4. Esta placa tiene soporte para distintas versiones de Android, por lo que resulta una plataforma muy adecuada para llevar a cabo nuestro trabajo.

La placa Odroid XU4 tiene un SoC (sistema en chip) Samsung Exynos 5422, que integra un procesador ARM big.LITTLE de 8 núcleos. Este procesador está provisto de 4 cores Cortex A15 (*big*) que operan a 2Ghz y 4 cores Cortex A7 (*little*) que funcionan a 1.4Ghz. La placa Odroid XU4 también integra una GPU ARM Mali-T628 MP6 (compatible con OpenGL ES 3.0/2.0/1.1 y OpenCL 1.1), motivo por el cual los creadores han implementado refrigeración activa en la placa Odroid XU4, como se puede ver en la figura 1.3. Para complementar el potente SoC fabricado por Samsung, la placa Odroid XU4 incorpora 2 Gb de memoria RAM LPDDR3 (PoP Stacked, integrada en el PCB) y un controlador de memoria flash eMMC 5.0 de 8 bits con conector para módulos externos de memoria eMMC, por lo que no incorpora almacenamiento integrado de serie. No obstante, admite tanto módulos eMMC como tarjetas microSD. De este modo, al igual

[2]El soporte para Android en PMCTrack-GUI fue incorporado a la rama oficial de PMCTrack durante el desarrollo de este Trabajo Fin de Grado.

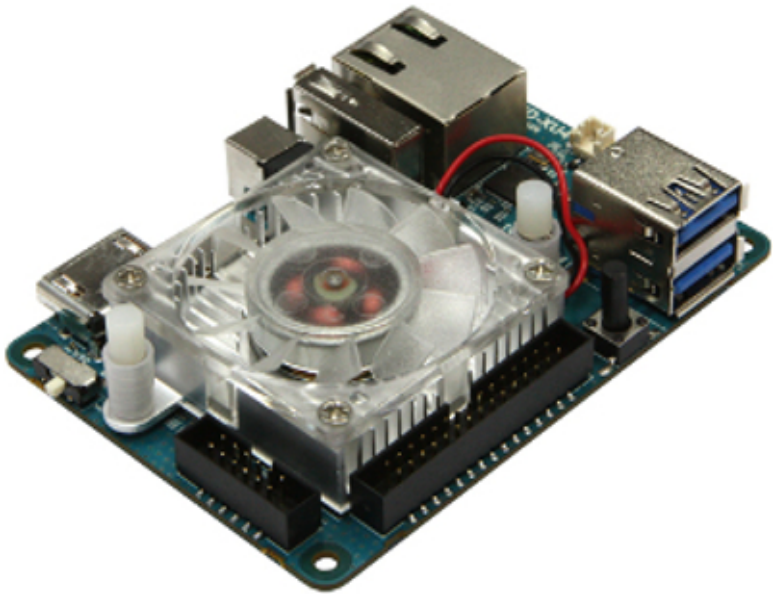


Figura 1.3: Odroid XU4

que en la Raspberry Pi es posible instalar un sistema operativo en una tarjeta externa sin problema. Para terminar con el apartado técnico, cabe mencionar que incorpora dos puertos USB 3.0 y un USB 2.0, además del ya mencionado lector de tarjetas microSD. También tiene una tarjeta de red Gigabit y salida de vídeo HDMI.

En la figura 1.4 se muestra una comparativa del rendimiento de varias placas en las que se ejecutaron varios benchmarks. Cabe destacar que la potencia de cálculo de la placa Odroid XU4 fue entre 3 y 4 veces más rápido que la placa Raspberry Pi 2 gracias a los núcleos de 2 GHz Cortex-A15 y el mayor ancho de banda de memoria.

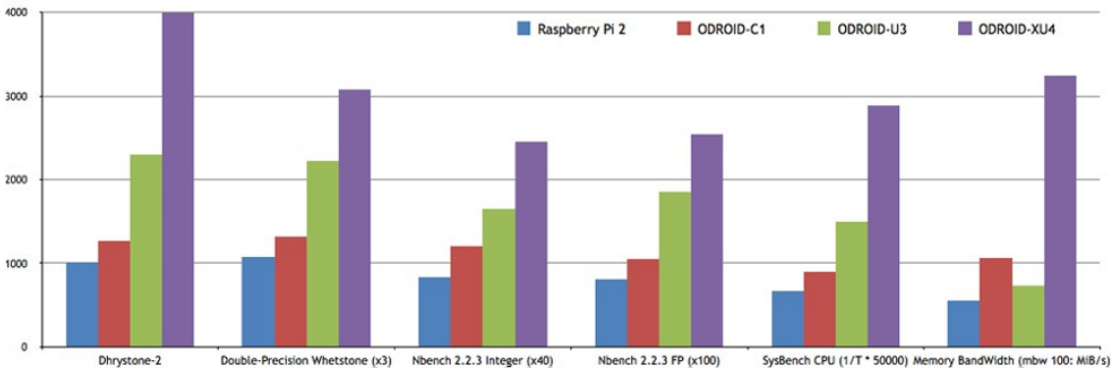


Figura 1.4: Rendimiento

Los usos que se le pueden dar a este mini PC (o placa de desarrollo) son muy amplios, y podemos hacer básicamente lo mismo que con Raspberry Pi pero con una mayor potencia y versatilidad (al tener más conectores y más rápidos).

La placa Odroid XU4 puede ejecutar varias versiones de Linux, incluyendo Ubuntu 15.04 y Android 4.4 KitKat y 5.0 Lollipop.

Se puede tanto compilar un kernel Android e instalar la imagen del kernel desde la tarjeta SD o de eMMC como desarrollar aplicaciones y ver cómo funcionan en esta placa. Además con una pantalla y un ratón ya dispones de un dispositivo con Android muy potente e independiente. Para conectarse a la placa se puede hacer tanto por puerto en serie como por ADB.

Odroid XU4 no tiene una comunidad masiva detrás. Por lo tanto, no existen tantos tutoriales ni foros de ayuda como en el caso de Raspberry Pi. Odroid XU4 no tiene USB OTG, sensores de monitorización de energía para medir el consumo de energía de los distintos clusters de cores, de la GPU, o de la DRAM en tiempo real.

1.4. Objetivos del proyecto

La herramienta PMCTrack ha sido diseñada específicamente para el sistema operativo GNU/Linux. Sus reducidas dependencias de librerías y herramientas externas permiten que pueda usarse en la mayor parte de distribuciones de Linux. El objetivo de este proyecto es realizar una adaptación de PMCTrack al sistema operativo Android, usando la placa Odroid XU4 como plataforma de desarrollo y evaluación. Llevar a cabo esta adaptación conlleva la realización de las siguientes tareas:

1. Modificación de la variante del kernel Linux propia de Android para incluir las extensiones requeridas por el módulo del kernel de PMCTrack.

2. Adaptación de las herramientas de modo usuario y el módulo del kernel de PMCTrack para el correcto funcionamiento de la herramienta en Android.

3. Desarrollo de una aplicación Android (Java), llamada PMCTrackApp, que permite visualizar en tiempo real las medidas de los contadores recabadas para las distintas aplicaciones que están siendo monitorizadas.

Como se muestra en el siguiente capítulo, las herramientas de modo usuario y el módulo del kernel de PMCTrack pueden compilarse de forma sencilla para Android/ARM con ayuda de un compilador cruzado. Estas herramientas funcionan, sin modificación en el código fuente, sobre cualquier kernel de Android que incorpore el parche de PMCTrack para Linux. En este TFG se creó un parche específico para una versión del kernel de Android con soporte para la placa Odroid XU4. Este parche ofrece a los desarrolladores del kernel acceso al API del kernel de PMCTrack, que facilita el desarrollo de algoritmos de planificación basados en contadores hardware [18].

A pesar de que las herramientas de modo usuario y el módulo del kernel de PMCTrack no requieren modificación para funcionar sobre Android, las diferencias sustanciales entre GNU/Linux y Android hacen que estos componentes software no ofrezcan un soporte adecuado de monitorización desde el espacio de usuario en Android. Esto se deriva del hecho de que PMCTrack no permitía originalmente la monitorización de aplicaciones que ya están en ejecución. De hecho, para obtener información de métricas de rendimiento (p.ej., IPC o tasa de fallos de cache) de un programa era preciso lanzar dicho programa usando la herramienta `pmctrack` de línea de comandos, o bien instrumentar el código del programa usando libpmctrack. En Android, los usuarios no disponen del código fuente de la mayor parte de aplicaciones, ni tampoco lanzan sus aplicaciones desde una ventana de terminal, sino que usan para ello el icono de la aplicación en el *Launcher* de Android. Para ofrecer un soporte adecuado en Android ha sido necesario incluir el modo *attach* en PMCTrack durante este Trabajo Fin de Grado. Este nuevo modo permite al usuario final iniciar una sesión de monitorización con una aplicación que ya se encuentre en ejecución. Como se describe en el capítulo 2, la inclusión de esta nueva funcionalidad ha requerido cambios tanto en la herramienta de línea de comandos `pmctrack` (nueva opción -p) como en el módulo del kernel. Más concretamente, la figura 1.5 indica qué componentes de la arquitectura de PMCTrack han sido modificados durante este TFG (marcados en color rojo).

1.5. Plan de trabajo

Para alcanzar los objetivos del proyecto, presentados en la sección anterior, el desarrollo del proyecto se dividió en las siguientes etapas:

- Lectura de documentación sobre desarrollo de aplicaciones en Android.
- Aprendizaje del entorno de desarrollo sobre la placa Odroid XU4, usando ADB y Android Studio.
- Creación del parche de PMCTrack para el kernel de Android.
- Instalación del kernel de Android modificado en la placa Odroid XU4.
- Generación de binarios para Android de los componentes de espacio de usuario y de espacio de kernel de PMCTrack sin modificar e instalación de estos componentes en la placa Odroid XU4.
- Realización de modificaciones en herramientas de línea de comandos y módulo del kernel de PMCTrack para proporcionar funcionalidad adicional necesaria en Android.
- Diseño de la interfaz de usuario de la aplicación PMCTrackApp.
- Implementación de la aplicación PMCTrackApp.
- Realización de análisis experimental sobre la placa Odroid XU4 usando la versión de PMCTrack creada para Android.

Hay que destacar que el orden de estas etapas no es estrictamente secuencial ya que fue necesario trabajar en aspectos de distintas etapas de forma simultánea. Por otra parte, durante el desarrollo de la aplicación PMCTrackApp fue necesario asignar distintas tareas a cada miembro para poder trabajar en paralelo y así avanzar más rápido con el desarrollo.

1.6. Estructura de la memoria

El resto del contenido de esta memoria se organiza de la siguiente forma:

- **El capítulo 2** explica el proceso de creación del parche de PMCTrack para Android, la incorporación del modo *attach* en PMCTrack, la modificación de los scripts de compilación para soportar la compilación cruzada de PMCTrack y la instalación de PMCTrack en la placa Odroid XU4.
- **El capítulo 3** presenta la aplicación PMCTrackApp, proporcionando un resumen de las características principales, su modo de uso y algunos detalles de diseño e implementación.
- **El capítulo 4** pone a prueba las extensiones de PMCTrack desarrolladas en este proyecto mediante dos casos de estudio.
- **El capítulo 5** expone las conclusiones finales de este Trabajo Fin de Grado y presenta el trabajo futuro.
- Finalmente, se proporcionan varios apéndices. En ellos se incluyen, la introducción y conclusiones del proyecto traducidos al inglés, un diagrama de diseño UML con detalles concretos del diseño de PMCTrackApp, y las contribuciones de cada miembro del equipo de trabajo de este TFG.

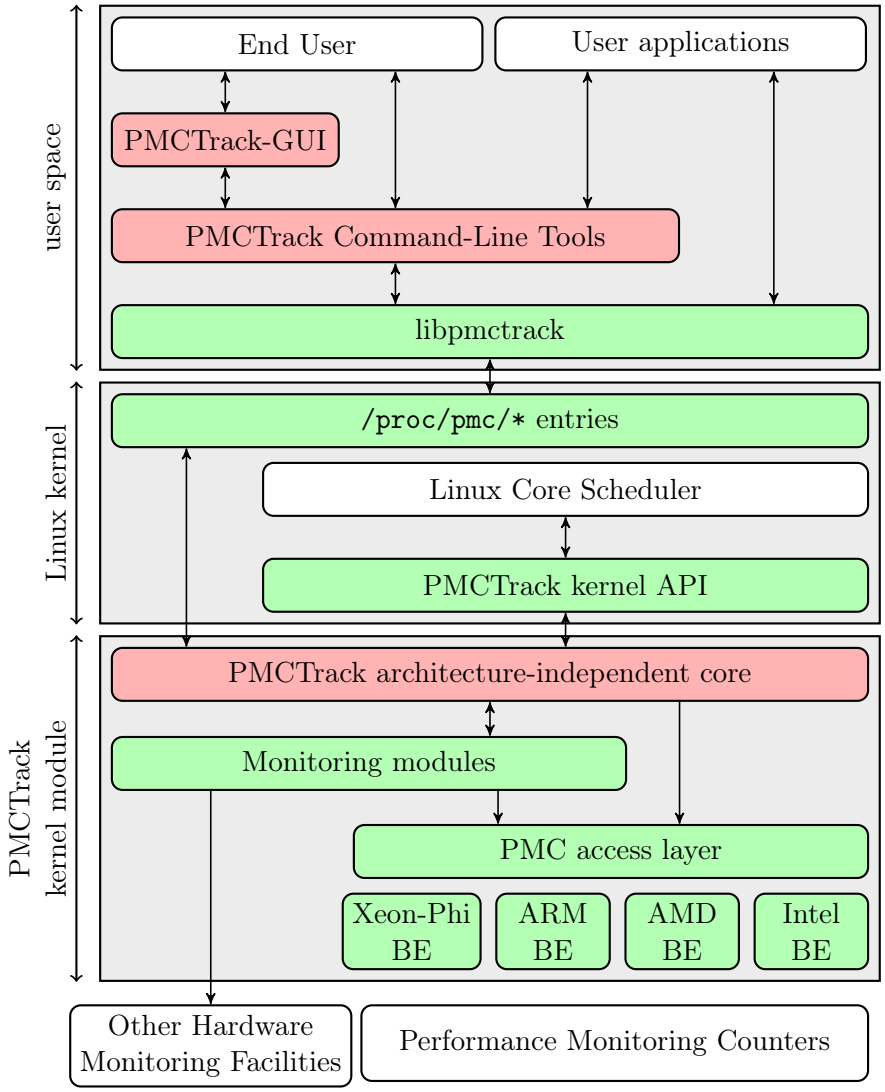


Figura 1.5: Cambios en la arquitectura de PMCTrack

Capítulo 2

Adaptación de PMCTrack a Android

En este capítulo se describen las modificaciones necesarias en los componentes de espacio de usuario y espacio de kernel de PMCTrack para el correcto funcionamiento de esta herramienta en Android.

Android es un sistema operativo que emplea una variante del kernel Linux. En Android, cada aplicación se ejecuta en su propio proceso, que ejecuta su propia instancia de la máquina virtual de Java[1] de Android. Debido a las diferencias entre GNU/Linux y Android que describimos a continuación, la adaptación de PMCTrack no resulta trivial.

El resto de este capítulo se organiza como sigue. En la sección 2.1 se ilustran las principales diferencias entre GNU/Linux y Android. En la sección 2.2 se explica la necesidad de incorporar una nueva funcionalidad `attach`. En la sección 2.3 se explican las modificaciones necesarias del kernel de Android para hacer funcionar PMCTrack. En la sección 2.4 se trata la compilación cruzada tanto del kernel como de los binarios de PMCTrack. En la sección 2.5 se describe la instalación de PMCTrack en la placa.

2.1. GNU/Linux vs. Android

Las principales diferencias en la arquitectura interna entre ambos sistemas están en el modelo de aplicaciones, en el modelo de drivers, en la estructura del sistema de ficheros y a nivel de kernel. Las características diferenciales se detallan a continuación.

[1] En la actualidad, coexisten dos entornos de ejecución de Java para Android: Dalvik y Android RunTime (ART). ART es la opción por defecto a partir de las versión 5.0 de Android.

2.1.1. Modelo de aplicaciones

En GNU/Linux las aplicaciones nativas interactúan con el kernel a través de la libc. En Android existen dos tipos de aplicaciones: aplicaciones Android/Java (*Android user-space*) y aplicaciones/demonios nativos (*Native user-space*), similares a aplicaciones nativas en Linux. Cada aplicación Android se ejecuta en un proceso Linux separado y puede constar de varios hilos. Asimismo, cada aplicación posee un usuario UNIX propio (UID por aplicación), tiene su propio sistema de ficheros, preferencias y bases de datos "privados" (sandbox). La compartición de datos entre aplicaciones se ha de solicitar explícitamente.

2.1.2. Modelo de drivers

En GNU/Linux la funcionalidad de un driver se suele encapsular completamente en un módulo del kernel. Los procesos de usuario acceden a las funciones del driver a través de la libc, del VFS (Virtual File System) o vía ficheros de dispositivo (por ejemplo, /dev/sda1).

En Android se ofrece un modelo alternativo (HAL) más adecuado para drivers propietarios de HW específico (GPS, Audio, Telefonía-RIL, ...). El fabricante distribuye un módulo del kernel que implementa solamente soporte de bajo nivel (puede ser GPL). La mayor parte de la funcionalidad del driver se encapsula en una biblioteca dinámica (.so) propietaria (la biblioteca se carga bajo demanda, por ejemplo, libaudio.so). Las aplicaciones Android interactúan con la biblioteca a través de los servicios del sistema.

2.1.3. Estructura del sistema de ficheros

En GNU/Linux se usa el estándar FHS (Filesystem Hierarchy Standard) para estructurar el sistema de ficheros. El FHS define el nombre, utilidad y contenidos de los principales directorios del sistema:

- /bin: Comandos especiales.
- /boot: Ficheros para el gestor de arranque, imágenes del kernel, ficheros de configuración del kernel.
- /etc: Ficheros de configuración del sistema.

En Android se evita explícitamente usar directorios definidos por el FHS (Permite crear estructuras híbridas del SF. GNU/Linux + Android).

Android utiliza dos directorios principales:

- /system: Directorio inmutable generado durante la compilación del SO.
  - Binarios, librerías estáticas y dinámicas.
  - Ficheros .jar.
  - Ficheros de configuración.
  - Aplicaciones Android estándar (Stock Apps).

- /data: Directorio donde se almacena el contenido mutable.
  - Aplicaciones de usuario.
  - Datos de las aplicaciones y del usuario.

Estos dos directorios se montan en particiones separadas del sistema de ficheros raíz, que se monta como RAM disk. Además de los dos directorios aparecen muchos otros con la funcionalidad habitual de FHS (/proc, /dev, /sys, /etc, /sbin, /mnt).

2.1.4. Nivel de kernel

El sistema operativo Android necesita un kernel Linux con extensiones para ofrecer su funcionalidad básica. La mayor parte de estas extensiones no forman parte de la versión *mainline* del kernel. A continuación se enumeran algunas de las extensiones del kernel más destacadas:

- Binder: Mecanismo de comunicación entre procesos.
- Wakelocks: Mecanismo que activa o suspende una aplicación.
- Ashmem: Gestor de memoria compartida.
- Low Memory Killer: Mecanismo que reduce el consumo de memoria de una aplicación.
- Paranoid Network: Mecanismo que otorga o deniega permiso de acceso a cada aplicación por cada tipo de protocolo de comunicación.

Más información acerca de las distintas extensiones del kernel puede encontrarse [22].

2.2. Incorporación de la funcionalidad “attach”

En Android, los usuarios no lanzan sus aplicaciones desde una ventana de terminal, sino que usan para ello el icono de la aplicación en el *Launcher* de Android. Para ofrecer un soporte más adecuado en la monitorización de aplicaciones en el contexto de Android, se ha incorporado la funcionalidad “attach” en PMCTrack. Esta nueva funcionalidad permite monitorizar cualquier aplicación que se encuentre ya en ejecución. Para ello, el usuario ha de proporcionar a PMCTrack el PID de la aplicación. A continuación se describen las modificaciones que se han llevado a cabo en los distintos componentes de PMCTrack para ofrecer esta funcionalidad.

El módulo del kernel de PMCTrack es el encargado de llevar a cabo la monitorización. Esto implica leer los contadores hardware cada cierto tiempo y almacenar las cuentas de eventos recopiladas en un buffer circular. Los componentes de modo usuario (como `libpmctrack` o la herramienta `pmctrack` de línea de comandos) consumen los datos de este buffer circular, uno por aplicación, leyendo de entradas /proc que exporta el módulo del kernel. La configuración de eventos hardware y de otros parámetros de la monitorización que establece el usuario final, se comunican al kernel también mediante el sistema de ficheros */proc*.

2.2.1. Cambios en modo Kernel

Para incorporar la funcionalidad "attach" en el módulo del kernel de PMCTrack se ha creado una nueva función `pmctrack_pid_attach()` en el fichero *mchw_ core.c*. Esta función se encarga de realizar las siguientes acciones:

- En primer lugar, se obtiene una referencia al descriptor del proceso monitor y del proceso a monitorizar. En caso de éxito se guarda una referencia al descriptor del monitor en la estructura del proceso monitorizado. De este modo, se garantiza que los procesos creados desde el proceso monitorizado tengan el mismo proceso monitor. A continuación, se activa un flag en el descriptor del proceso monitorizado (campo `attached`), para indicar que este proceso está siendo monitorizado. Cabe destacar, que el módulo del kernel almacenará todas las muestras de los contadores hardware de todos los hilos del proceso monitorizado en el mismo buffer circular (uno por aplicación).

- En segundo lugar se asignará al proceso monitorizado la configuración de los contadores hardware especificada por el usuario.

- Finalmente, la función buscará la CPU donde se ejecuta el proceso que desea ser monitorizado. Esto es necesario para realizar la configuración inicial de los contadores hardware en esa CPU. Al llevar a cabo dicha configuración hay que tener en cuenta que el proceso puede estar en los siguientes estados:

  - *Bloqueado*: En este caso, no es necesario realizar la configuración inicial de los contadores hardware, ya que dicha configuración se llevará a cabo automáticamente cuando este proceso entre a ejecutar en una CPU.
  - *En ejecución*: En este escenario el proceso se encuentra ejecutando instrucciones en la CPU identificada. Para llevar a cabo la configuración de los contadores de forma segura, es preciso desencadenar una interrupción (*Inter Process Interrupt* - IPI) en esa CPU. El proceso de configuración de los contadores se lleva a cabo durante el procesamiento de esa interrupción, durante el cual el proceso no ejecuta instrucciones.
  - *Listo para ejecutar*: En este caso se opera de la misma forma que cuando el proceso se encuentra bloqueado: la configuración se llevará a cabo automáticamente cuando este proceso entre a ejecutar en una CPU.
  - *Migrado*: Se puede dar el caso de que a la hora de generar una interrupción de tipo IPI, el planificador haya migrado el proceso a otra CPU. En este caso se volverá a generar una nueva IPI, en la CPU actual donde está asignado el proceso monitorizado.

2.2.2. Cambios en modo Usuario

Para iniciar una sesión de monitorización con una aplicación que se encuentra actualmente en ejecución es preciso invocar la herramienta `pmctrack` de línea de comandos especificando el PID del proceso a monitorizar como parámetro de la nueva opción `-p`. Al hacer esto la herramienta pmctrack que ha sido modificada para ofrecer esta funcionalidad, inicia una sesión individual de monitorización con cada uno de los hilos del proceso. Para ello la herramienta pmctrack invoca la nueva función `attach_pid_set()` que escribe el PID de cada hilo a monitorizar en la entrada */proc/pmc/monitor*. Además, el módulo del kernel de PMCTrack garantiza que los nuevos hilos que el proceso monitorizado vaya creando también sean monitorizados.

2.2.3. Detach

El procedimiento de *detach* es el encargado de romper el enlace entre el proceso monitor y monitorizado (incluyendo a los hilos que el proceso monitorizado vaya creando). Asimismo se encarga de la correcta liberación de la memoria asociada a las estructuras de datos empleadas en la monitorización. Para ello el módulo del kernel localiza el descriptor de proceso de cada hilo monitorizado y establece el valor del campo `attached` de cada hilo a cero.

2.3. Modificación de la variante del kernel Linux propia de Android

Para poder adaptar la herramienta PMCTrack (modo kernel) a nuestro SO. Hemos creado un *patch* específico que ya se encuentra integrado en la rama principal de PMCTrack [14]. En la tabla 2.1 se proporciona un listado de los archivos modificados y el motivo de la modificación.

2.3.1. Problema Vermagic

Para la correcta instalación del kernel parcheado en la placa, Odroid XU4, es necesario preservar el *vermagic* del kernel original de Android instalado. De este modo, los módulos del kernel ya instalados en la plataforma y compilados contra el kernel original, pueden cargarse correctamente con el kernel modificado.

Para preservar el *vermagic* del kernel, la investigación de los distintos ficheros del kernel nos ha llevado a hacer las siguientes modificaciones.

Archivos Modificados	**Objetivos**
kernel/sched/core.c	Modificaciones del planificador requeridas por PMCTrack
kernel/exit.c	Notificaciones para el módulo del kernel (*exit callback*)
kernel/fork.c	Notificaciones para el módulo del kernel (*fork callback*)
kernel/pmctrack.c	Implementación de la interfaz `pmc_ops_t` del las funciones del API de kernel PMCTrack
kernel/Makefile	Inclusión del fichero objeto `pmctrack.o` en la imagen del kernel
include/linux/sched.h	Modificaciones del planificador requeridas por PMCTrack
include/linux/pmctrack.h	Declaración de la interfaz `pmc_ops_t` y del API de kernel PMCTrack
init/Kconfig	Inclusión de entrada de configuración CONFIG_PMCTRACK en Kconfig
arch/arm/kernel/perf_event_cpu.c	Desactivación del sistema *PerfEvents*
kernel/irq/manage.c	Modificaciones de interrupciones requeridas por PMCTrack
include/linux/interrupt.h	Modificaciones de interrupciones requeridas por PMCTrack
arch/arm/boot/dts/exynos5422_evt0.dtsi	Modificaciones de interrupciones combinadas requeridas por PMCTrack
Makefile	Solucionar el problema del vermagic

Tabla 2.1: Modificaciones de los archivos del kernel.

2.3.1.1. La modificación de vermagic.h (Solución fallida)

Hemos observado que la compilación del kernel obtiene el *vermagic* del fichero `vermagic.h`, seguida de la fecha y la hora de compilación. En dicho fichero se ha introducido el *vermagic* del kernel original. Lo que se ha conseguido es que el kernel tenga el *vermagic* original pero seguido de la fecha de compilación. Tras esa modificación fallida se ha intentado quitar la fecha pero en su lugar se asignaba *dirty*.

2.3.1.2. La modificación del Makefile (Solución exitosa)

Por los problemas anteriormente descritos, fue necesario modificar el *Makefile* del directorio raíz de las fuentes del kernel. Además para que la compilación se realizara correctamente, generando una imagen del kernel con el *vermagic* deseado, fue preciso pasar la cadena de versión del kernel original como parámetro del comando `make`. Como la modificación del *Makefile* es esencial para generar la imagen adecuada de PMCTrack, los cambios del *Makefile* están incluidos en el parche de PMCTrack.

2.4. Compilación cruzada y modificación de sistema de compilación

En los dispositivos con Android, como la placa Odroid XU4 usada en este TFG, no se suelen distribuir herramientas de compilación. Por lo tanto, no es posible genenerar ejecutables a partir de las fuentes de un programa desde un terminal de Android (p.ej., vía `adb shell`). Los desarrolladores de aplicaciones realizan la compilación de sus aplicaciones desde un *host* de desarrollo, empleando las herramientas proporcionadas por Google, como el Android SDK, el NDK o el Android Studio. Para este TFG, se utilizó como *host* de desarrollo una estación de trabajo con Debian. Como esta plataforma tiene arquitectura x86, fue necesario hacer uso de un compilador cruzado para generar los binarios ARM asociados a los distintos componentes de PMCTrack, que están programados en C, como el módulo del kernel, la biblioteca `libpmctrack` o la herramienta de línea de comandos PMCTrack.

2.4.1. Compilación cruzada del kernel y de `pmctrack`

Después de modificar el kernel de Android se ha creado un archivo *config* donde se guardan las configuraciones de la compilación del kernel modificado. Para la compilación del kernel se ha usado un compilador cruzado de ARM. Una vez teniendo el kernel compilado, mediante otro compilador cruzado, se han compilado los componentes de pmctrack para el modo usuario. Asimismo se ha compilado el módulo de PMCTrack contra el kernel para que se pueda cargar en ese kernel.

2.4.2. Makefiles

La compilación del kernel y de la herramienta `pmctrack` del espacio de usuario ha dado lugar a la creación de tres Makefiles. Uno dentro del kernel para poder incluir la herramienta `pmctrack` al modo kernel, otro en el directorio del código fuente de Android para compilar el kernel con el vermagic específico y otro para poder generar los binarios de la herramienta `pmctrack` para el modo usuario.

2.4.3. Compilación del Kernel

La compilación se realiza para generar la imagen binaria del kernel, luego hay que instalarla en la placa. Para compilar el kernel hay que ejecutar los siguientes comandos en el host de compilación.

```
$ export PATH=/scratch/tfg-android/Toolchain/arm-eabi-4.6/bin:$PATH
$ make ARCH=arm CROSS_COMPILE=arm-eabi- EXTRAVERSION=-g10c00c9 -j4
```

El primer comando se encarga de indicar al sistema la ruta del compilador. El segundo se encarga de compilar el kernel modificado con el vermagic adecuado, en nuestro caso *-g10c00c9*. La opción -j4 es para que los 4 cores trabajen en la compilación.

Tras la compilación se genera una imagen binaria comprimida del kernel *zImage-dtb* en el directorio *arch/arm/boot/*.

2.4.4. Generación de los binarios de `pmctrack`

Para obtener los binarios de `pmctrack`. Hay que bajar el código fuente de PMCTrack de la página oficial [13]. Ir al directorio PMCtrack donde está el código fuente y compilar con el *Makefile* de la Odroid. Ese Makefile se encarga de lanzar los comandos necesarios de llevar acabo tanto la compilación de los binarios de PMCTrack de modo usuario, como la compilación del módulo de PMCTrack contra el kernel. Para conseguir lo explicado anteriormente, hay que seguir los siguientes pasos.

```
. shrc
pmctrack-manager build-android-odroid arm-linux-gnueabihf- /scratch/tfg-android/
    linux
**** Target system information ***
Processor_vendor=ARM
Kernel_HZ=200
Processor_bitwidth=64
Cross_compiler=arm-linux-gnueabihf-gcc
********************************************************************************
Press ENTER to start PMCTrack's cross-compilation for Android on the Odroid-XU4
    Board ...
********************************************************************************
Cleaning existing object files for mchw_odroid_xu.ko module
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -C /scratch/tfg-android/linux O=
    M=/scratch/tfg-android/pmctrack/src/modules/pmcs/odroid-xu  clean
make[1]: Entering directory '/scratch/tfg-android/linux'
  CLEAN   /scratch/tfg-android/pmctrack/src/modules/pmcs/odroid-xu/.tmp_versions
  CLEAN   /scratch/tfg-android/pmctrack/src/modules/pmcs/odroid-xu/Module.symvers
make[1]: Leaving directory '/scratch/tfg-android/linux'
rm -f mchw_core.c mc_experiments.c pmu_config_arm.c cbuffer.c monitoring_mod.c
    syswide.c ipc_sampling_sf_mm.c oracle_sf_mm.c sched_prototype_mm.c ../*.o
==========================================
Building kernel module odroid-xu....
==========================================
make MODULE_VERSION=kk ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -C /scratch/tfg
    -android/linux O=  M=/scratch/tfg-android/pmctrack/src/modules/pmcs/odroid-xu
     modules
make[1]: Entering directory '/scratch/tfg-android/linux'
  CC [M]  /scratch/tfg-android/pmctrack/src/modules/pmcs/odroid-xu/mchw_core.o
....
```

Tras terminar la compilación se generan los siguientes binarios.

- En el directorio */pmctrack/bin/*
  - pmctrack: Es el ejecutable de la herramienta PMCTrack.
  - pmc-events: Es el ejecutable para saber los eventos hardware del procesador.
- En el directorio */pmctrack/src/modules/pmcs/odroid-xu*
  - mchw_odroid_xu.ko: Es el módulo del kernel.

2.5. Instalación del PMCTrack

La instalación de PMCTrack en Android consiste en instalar el nuevo kernel en la placa y copiar los componentes de espacio de usuario y espacio de kernel a la placa.

2.5.1. Instalación del kernel

Para instalar el kernel debemos transferir la imagen del kernel desde el *host* de desarrollo a la tarjeta SD de placa, donde el sistema Android está instalado. Esto puede llevarse a cabo empleando el comando `adb push` desde el *host* de desarrollo con el sistema Android arrancado en la placa. Alternativamente, se puede montar el sistema de ficheros de Android en el *host* y copiar el fichero correspondiente; esto obliga a extraer la tarjeta SD de la placa y conectarla al *host*.

Una vez que el fichero está copiado a la tarjeta SD, es preciso reiniciarla y acceder al `shell` del gestor de arranque (`uboot` en nuestra plataforma). Para poder acceder a este *shell* debemos iniciar una sesión por puerto serie con la placa desde el *host* de desarrollo. En nuestro entorno experimental empleamos el programa `minicom` para realizar la conexión. Para instalar el kernel en la placa debemos reemplazar el kernel ya instalado por nuestra nueva imagen (fichero `zImage-dtb`). Esto se realiza escribiendo en una zona determinada de la tarjeta SD (tabla de particiones *custom*) indicada por el fabricante [10].

A continuación se indican los pasos detallados para llevar a cabo una correcta instalación del kernel.

1. Buscar IP de la placa: Para poder conectarse a la placa mediante *adb* hay que conocer la dirección *IP*. Para ello se lanza el siguiente comando.

```
$ netcfg
ip6tnl0  DOWN            0.0.0.0/0   0x00000080 00:00:00:00:00:00
lo       UP           127.0.0.1/8   0x00000049 00:00:00:00:00:00
sit0     DOWN            0.0.0.0/0   0x00000080 00:00:00:00:00:00
eth0     UP        192.168.1.19/24   0x00001043 00:1e:06:30:11:73
```

2. Conectarse a la placa mediante ADB: Hay que conectarse a la placa con el siguiente comando.

```
$ adb connect 192.168.1.19
connected to 192.168.1.19:5555
$ adb root
adbd is already running as root
$ adb connect 192.168.1.19
connected to 192.168.1.19:5555
```

3. Copiar la imagen binaria del kernel a la placa Odroid XU4: No se puede llevar a cabo la instalación del nuevo kernel sin haber copiado su imagen binaria *zImage-dtb* a la placa, los siguientes comandos se encargan de ese paso.

```
$ cd $HOME
$ mkdir tmp
$ cp (zImagen-dtb) tmp
$ adb push tmp/zImage-dtbl storage/sdcard1/
```

3. Instalar el nuevo kernel: Consiste en la instalación del kernel modificado reemplazando al antiguo. Existe una zona específica de la tarjeta (*offset* 0x40008000) destinada a almacenar el kernel [10].

```
$ minicom ttyUSB0
welcome to minicom 2.7
$ su -l
$ reboot
pulse (enter)
Exynos5422 # fatls mmc 0:1
    48536443 zimage-dtb
Exynos5422 # fatload mmc 0:1 40008000 zImage-dtb
Exynos5422 # movi write kernel 0 40008000
Exynos5422 # boot
```

2.5.2. Instalación del módulo en el kernel y copia de los binarios de PMCTrack

No se puede garantizar el funcionamiento de la herramienta PMCTrack a nivel de usuario sin la instalación del módulo de PMCTrack y la existencia de los ejecutables de `pmctrack`. Para poder hacer funcionar PMCTrack en la placa hay que seguir los siguientes pasos.

1. Conectarse a la placa mediante ADB: Se conecta a la placa conociendo su IP mediante los siguientes comandos.

```
$ adb connect 192.168.1.19
connected to 192.168.1.19:5555
$ adb root
adbd is already running as root
$ adb connect 192.168.1.19
connected to 192.168.1.19:5555
```

2. Copiar los binarios de los componentes de modo usuario y modo kernel de PMCTrack

```
$ cd $HOME
$ mkdir tmp
$ cp (pmctrack,pmc-events,mchw_odroid_xu.ko) tmp
$ adb push tmp/(pmctrack,pmc-events,mchw_odroid_xu.ko) storage/sdcard1/
```

3. Cargar el módulo en el kernel: Esto es necesario para poder iniciar sesiones de monitorización de aplicaciones con PMCTrack.

```
$ insmod mchw_odroid_xu.ko
```

Para lanzar `pmctrack` desde el terminal basta con acceder al directorio donde se encuentra y lanzarlo como se indica en la documentación [13].

Capítulo 3

Desarrollo de una aplicación Android (PMCTrackApp)

Para ampliar la funcionalidad de la herramienta PMCTrack, en lo que respecta a la monitorización de aplicaciones desde modo usuario, hemos desarrollado PMCTrackApp. Esta aplicación permite la visualización de gráficas de rendimiento en tiempo real de aplicaciones de usuario, usando por debajo la herramienta `pmctrack` de línea de comandos para obtener los datos de monitorización proporcionados por el módulo del kernel.

Este capítulo se estructura como sigue. En la sección 3.1 se justifica la necesidad del desarrollo de PMCTrackApp. La sección 3.2 presenta las características principales de la aplicación. La sección 3.3 describe detalladamente el modo de uso del usuario de PMCTrackApp. En la sección 3.4 se describen la interfaz de usuario y las tecnologías utilizadas. En la sección 3.5 explica el diseño interno de la aplicación. Finalmente en la sección 3.6 se mencionan algunas consideraciones de diseño.

3.1. Motivación

Como se mencionó en el Capítulo 1, a pesar del gran potencial de la herramienta PMCTrack, esta cuenta con limitaciones al ser usada para monitorizar aplicaciones desde el espacio de usuario. El principal problema consiste en que la herramienta `pmctrack` de línea de comandos proporciona tanta información al usuario a través de los PMCs que éste no puede interpretar, siendo necesario procesarla a posteriori.

Para solucionar este problema, es necesario el desarrollo de un *front-end* gráfico que facilite al usuario la tarea de monitorización de aplicaciones usando PMCs. PMCTrackApp ha sido desarrollada para superar esta limitación.

3.2. Características de PMCTrackApp

PMCTrackApp es una aplicación que se desarrolló para que funcione en el SO Android. Esencialmente la aplicación proporciona de forma visual y totalmente automática la lista de PMCs disponibles y permite configurarlos haciendo clic en el evento que se desea asociar a cada contador hardware. Cabe destacar que PMCTrackApp soporta la monitorización de aplicaciones multihilo. Es posible visualizar gráficas en tiempo real de un determinado hilo de la aplicación que se esté monitorizando. PMCTrack da la posibilidad de poder cambiar de métrica y de hilo monitorizado del mismo experimento.

Para especificar las métricas de alto nivel que serán visualizadas posteriormente en gráficas en tiempo real, el usuario tan sólo debe escribir fórmulas en las cuales las variables serán los nombres de los contadores configurados previamente (pmc0, pmc1, pmc2, ...). Por ejemplo si el contador 0 lleva la cuenta del número de instrucciones retiradas y el contador 3 contabiliza el número de fallos de último nivel de cache (LLC), la fórmula (pmc3 * 1000)/pmc0 define la métrica de rendimiento "Número de fallos de LLC por cada 1K instrucciones". El conjunto de contadores y métricas forman un experimento.

El código fuente de PMCTrackApp será liberado en los próximos meses con licencia GPL, junto con el resto del código de la herramienta PMCTrack, estando disponible para todo el mundo de forma gratuita. En el momento de la redacción de este documento, PMCTrackApp es la única aplicación de monitorización existente en Android. PMCTrack cuenta con las siguientes características:

- Permite crear experimentos independientes.
- Permite crear métricas para luego poder mostrarlas en una gráfica.
- Permite observar en tiempo real la gráfica de métricas de rendimiento, para el mismo o distintos hilos de ejecución de la aplicación monitorizada.
- Permite pausar/reanudar la ejecución de la aplicación que se está monitorizando.
- Permite realizar capturas de las gráficas.

3.3. Modo de uso

Cuando el usuario abre la aplicación, accede a la ventana principal de PMCTrackApp, que se muestra en la figura 3.1. En dicha ventana se puede seleccionar qué aplicación monitorizar especificando su PID o seleccionando el nombre de la aplicación en una lista desplegable. Para configurar un experimento se dispone del botón "SETTINGS". Disponemos de una lista desplegable con las métricas (si hay alguna añadida).

Un vez hecho clic en el botón "SETTINGS" se mostrará una nueva ventana (figura 3.2). Donde el usuario podrá configurar los PMCs con los que cuenta la máquina a monitorizar, asignando eventos a dichos contadores. Para realizar la asignación de eventos, el usuario tiene que hacer clic en el botón "ASSIGN EVENT" del contador que quiera configurar. A continuación, hacer clic en el evento que quiera asignar de entre todo el listado de eventos disponibles que se mostrarán por pantalla (figura 3.3).

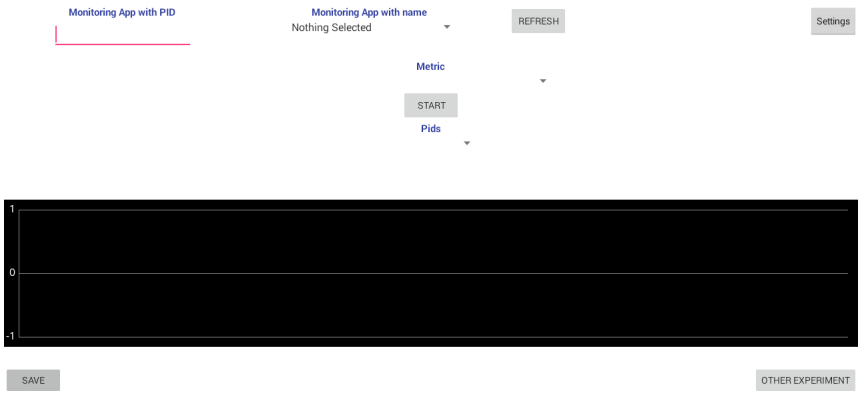


Figura 3.1: Pantalla principal

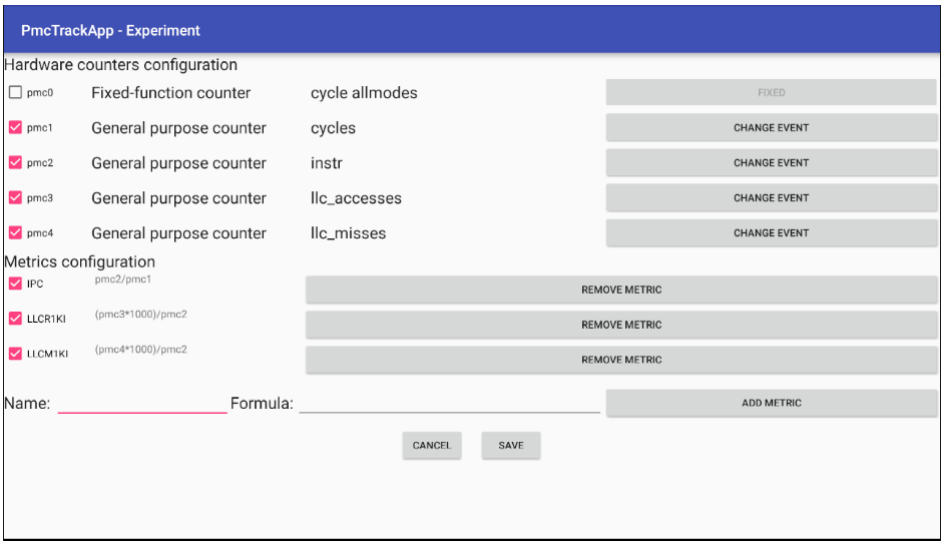


Figura 3.2: Configuración de experimento

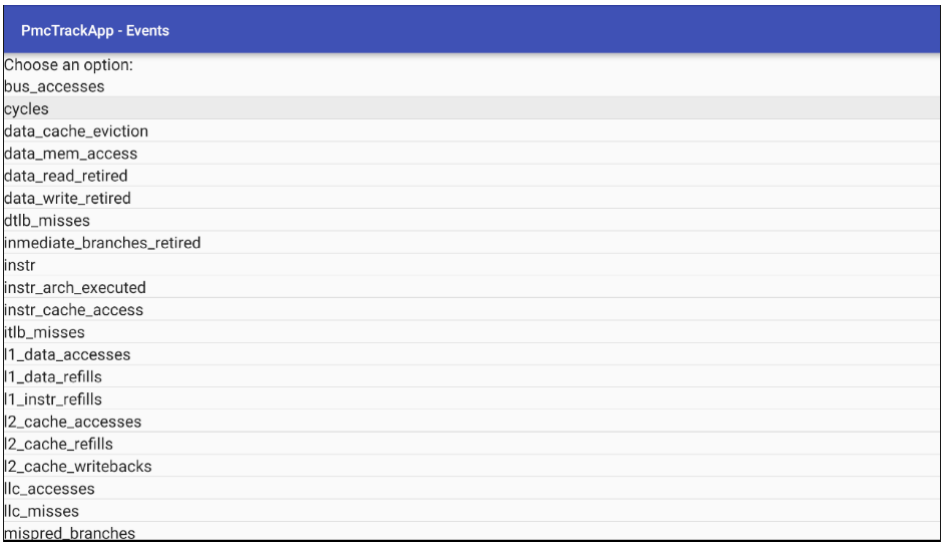


Figura 3.3: Configuración de contador

Una vez configurados los contadores que se quieren utilizar, debajo del cuadro de configuración de contadores el usuario se encontrará con la configuración de métricas. En la Figura 3.2 se puede observar la ventana de configuración de contadores y métricas donde se muestran las distintas definiciones de métrica. El cuadro de configuración de métricas permite al usuario configurar métricas de alto nivel que podrán verse posteriormente en forma de gráfica en tiempo real. Para la generación de métricas se usan fórmulas cuyas variables son los contadores que el usuario configuró anteriormente (pmc0, pmc1, ...). No hay ninguna limitación a la hora de generar las fórmulas, de tal manera que el usuario podrá escribir fórmulas tan complejas como desee, como por ejemplo ((pmc0^2)/pmc1 * 1000) * pmc4. Cabe destacar que es posible crear más de un experimento, más de un conjunto de contadores y métricas. De este modo, es posible configurar un contador con un determinado evento y usarlo en varias métricas.

Una vez esté todo configurado, el usuario hará clic en el botón "START" de la ventana principal para iniciar la monitorización. Al hacer clic se mostrará una lista con todos los PIDs de los hilos de la aplicación monitorizada en la ventana principal. En dicha lista los PIDs de los hilos activos (valores de los contadores distintos de cero) aparecerán en color verde y los inactivos en rojo. Se dispone de un botón "REFRESH" para actualizar tanto los PIDs como su actividad. Al seleccionar un PID y hacer clic en "GRAPH" se visualizará la gráfica en tiempo real de la métrica seleccionada como se observa en la figura 3.4.

Para que el usuario pueda visualizar gráficas de diferentes métricas. Se ha decidido que después del arranque de la primera métrica, el botón "START" cambie de nombre a "CHANGE". Ofreciendo la funcionalidad de resetear la gráfica y mostrar los resultados de la métrica seleccionada. No se puede hacer de otra manera porque Android no es multiventana. Por lo que no es posible visualizar dos métricas a la vez.

Figura 3.4: Monitorización de una aplicación de usuario

En cualquier momento podemos realizar las siguientes acciones:

- Mostrar otra gráfica distinta. Seleccionando otra métrica de ese experimento u otro PID.
- Hacer captura de la gráfica actual. El usuario puede hacer en cualquier momento de la monitorización una captura de una gráfica tal y como se está visualizando en ese instante, guardándola con formato de imagen PNG (figura 3.4) .
- Pausar/reanudar la aplicación. El usuario puede pausar la ejecución de la aplicación que se está monitorizando cuando lo desee, reanudarla posteriormente. Esto puede servir para realizar capturas de gráficas en un punto escogido de la ejecución con más precisión.
- Realizar otros experimentos sin salir de la aplicación.

3.4. Interfaz de usuario y tecnologías utilizadas

El diseño de la interfaz de usuario se hizo a partir de bocetos y el desarrollo de la aplicación se hizo usando varias tecnologías que se describen a continuación.

3.4.1. Interfaz de usuario

Para realizar el diseño de la interfaz de usuario hicimos dos bocetos para poder tener variedad dónde elegir.

El primer diseño se muestra en la figura 3.5. Este diseño está basado en un conjunto de ventanas en las que el usuario puede seleccionar unos experimentos que se han creado por defecto. Después, se muestra otra ventana en la que se puede seleccionar una aplicación de las que estén en ejecución. Por último se muestra la ventana en la que aparece la gráfica del experimento y la aplicación que haya elegido el usuario. Este diseño resulta muy sencillo para el usuario, pero tiene el inconveniente de que no explota gran parte del potencial de PMCTrack.

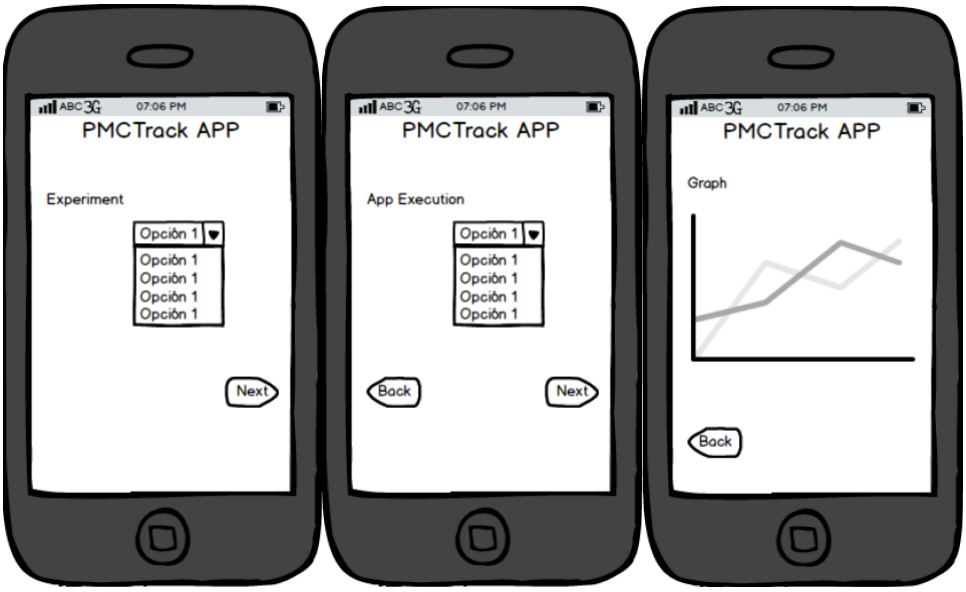


Figura 3.5: Bocetos del primer diseño para la aplicación PMCTrackApp

El segundo diseño que se planteó se muestra en la figura 3.6, es el diseño que hemos elegido. Esta interfaz consta de un conjunto de ventanas en las que el usuario puede ir configurando los contadores y añadir las métricas que le interesen. Después se muestra otra ventana en la que se puede seleccionar un PID de las aplicaciones que estén en ejecución. Por último se muestra la ventana en la que se muestra la gráfica del PID y la métrica que haya elegido el usuario. Esta interfaz de usuario es compleja de utilizar, pero saca el máximo partido a PMCTrack.

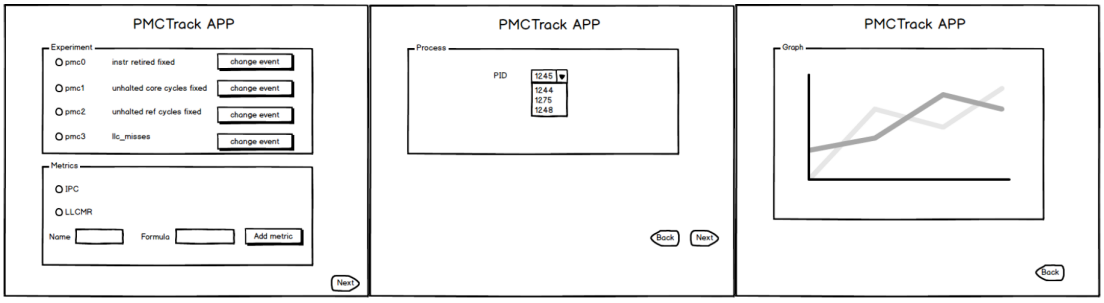


Figura 3.6: Bocetos del segundo diseño para la aplicación PMCTrackApp

3.4.2. Tecnologías utilizadas

La aplicación PMCTrackApp se ha implementado en Java (lenguaje de desarrollo de aplicaciones móviles en Android). Usando el entorno de desarrollo integrado "Android Studio" el entorno usado por los desarrolladores de aplicaciones Android. La instalación de la aplicación se hace mediante SDK-tools. De este modo permite conectarse a la placa mediante *ADB*.

El diseño de ventanas y diálogos está especificado mediante documentos XML.

Para mostrar las métricas en forma de grafo se ha usado la librería *mChart*. Esa librería no está implementada para que funcione en "Android Studio". Para ello había que adaptarla a ese entorno antes de usarla. El uso de *mChart* nos ha resultado muy útil ya que ofrece una variedad de funcionalidades.

Para parsear las métricas y calcular su resultado, hemos optado por usar la librería *Interpreter*. Esa librería se encarga de interpretar fórmulas matemáticas y calcular su valor.

3.5. Componentes internos de PMCTrack

En esta sección vamos a explicar la estructura interna de nuestra aplicación y el motivo de tal estructura con el objetivo de facilitar lo máximo posible el trabajo al usuario. La figura 3.7 muestra un diagrama *UML* de componentes de nuestra aplicación. En este diagrama se ve que PMCTrackApp se compone de tres principales procedimientos expuestos a continuación.

- Parse: es el procedimiento que se encarga de parsear las métricas y guardar sus valores en las estructuras correspondientes.
- Appsname: ese procedimiento se encarga de ejecutar el comando *ps* y actualiza la lista de los nombres de las aplicaciones Java arrancadas.
- Printgraph: es el procedimiento encargado de sacar los valores de las métricas correspondientes y mostrarlas en tiempo real.

Los componentes más destacables de PMCTrackApp son los siguientes:

- activity_main.xml: Contiene la implementación de la interfaz de usuario asociada a la actividad principal.
- events.xml: Contiene la implementación de la interfaz de usuario de la configuración de los contadores.
- experiment.xml: Contiene la implementación de la interfaz de usuario de la configuración de los distintos experimentos.

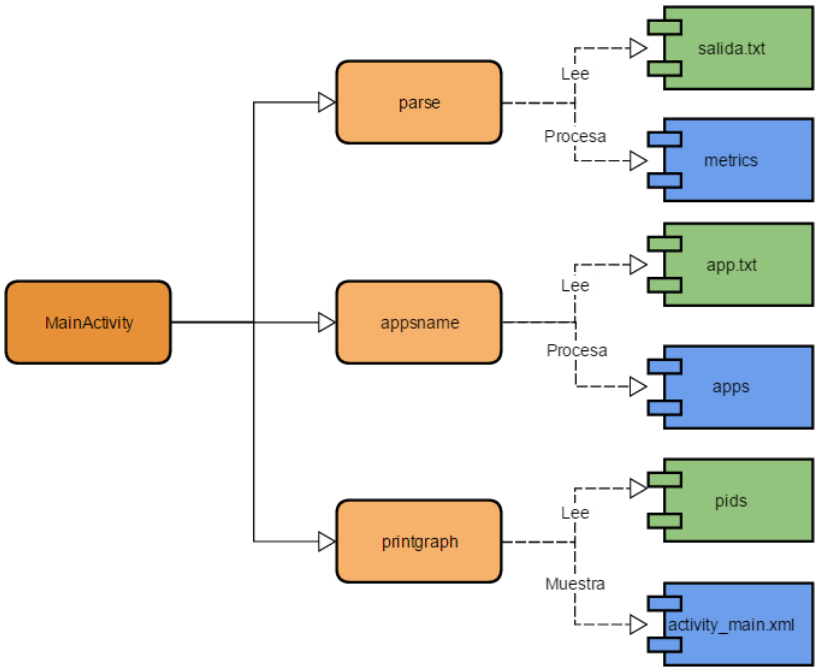


Figura 3.7: Diagrama de componentes de PMCTrackApp

- app.txt: Este archivo guarda toda la información de las aplicaciones Java que están en ejecución con la creación de un pipe entre el comando ps y app.txt. En Android cada vez que se reinicia la máquina se resetean los permisos de las ubicaciones. Por este motivo hemos encontrado una manera cien por cien segura para poder acceder a las ubicaciones restringidas de Android de forma sutil y se trata de ejecutar un comando como root desde la propia aplicación.
- salida.txt: Este archivo va actualizándose con la salida del comando `pmctrack` correspondiente a la configuración del usuario. A partir del cual vamos parseando la salida para dar al usuario una gráfica en tiempo real.

3.6. Consideraciones generales de diseño

Hay que destacar, que cuando se arranca la aplicación esta se encarga de cargar el módulo (si previamente no había sido cargado) para asegurar el correcto funcionamiento de PMCTrackApp.

Para facilitar al usuario la elección de la aplicación a monitorizar. Consultamos */proc* para mostrar las aplicaciones Java que están corriendo. Se dispone de un botón "REFRESH" para actualizar la lista.

En Android las aplicaciones se quedan pausadas en segundo plano. Por este motivo hemos tenido que ejecutar el comando `pmctrack` adecuado en un proceso de *bash* desde código Java. La ubicación de los *pmc* queda fuera del espacio de usuario de PMCTrackApp. Esto nos ha llevado a redirigir la salida del comando `pmctrack` al archivo *salida* con la opción -o.

Capítulo 4

Casos de estudio

En este capítulo evaluamos la nueva funcionalidad de PMCTrack realizada en este proyecto, mediante dos casos de estudio. En el primer caso de estudio 4.1 explotamos el potencial de PMCTrackApp y PMCTrack en su conjunto para recabar información de rendimiento mediante contadores hardware de los dos tipos de cores presentes en la placa Odroid XU4. El segundo caso de estudio 4.2 ilustra las capacidades de PMCTrackApp para monitorizar el rendimiento a lo largo del tiempo de las aplicaciones.

4.1. Monitorización del rendimiento con PMCTrackApp

Para llevar a cabo nuestro análisis, hemos utilizado los siguientes benchmarks:

- AndEBench: Incluye distintos test de rendimiento general de la CPU tanto del Java *userspace* como el *Native userspace*.
- AnTuTu: Incluye distintos test de rendimiento general de la CPU, gráficos 2D y 3D, memoria RAM y velocidades de lectura y escritura de la memoria interna y tarjetas SD.
- GFXBench: Realiza tests de gráficos 2D y 3D (*graphics-oriented benchmark*).
- Vellamo: Incluye distintos tests para medir el comportamiento del navegador web.

Para más información sobre los Benchmarks elegidos [2].

Las figuras 4.1, 4.2, y 4.3 muestran respectivamente el número de instrucciones por ciclo (IPC) medio, el número de fallos de último nivel de cache (LLC) y el número de accesos al último nivel de cache (LLC) por cada mil instrucciones retiradas para los benchmarks seleccionados en los dos tipos de core usados en nuestro estudio. Para asegurar que un benchmark se ejecuta en un tipo de core específico, usamos la opción `-b` de la herramienta de línea de comandos `pmctrack`. El IPC constituye una métrica global

del rendimiento de una aplicación, mientras que las otras dos métricas pueden permitir explicar la disminución del rendimiento de una aplicación debido a paradas en el pipeline del procesador. Típicamente, a mayor número de fallos o accesos al último nivel de cache, menor rendimiento experimentará la aplicación.

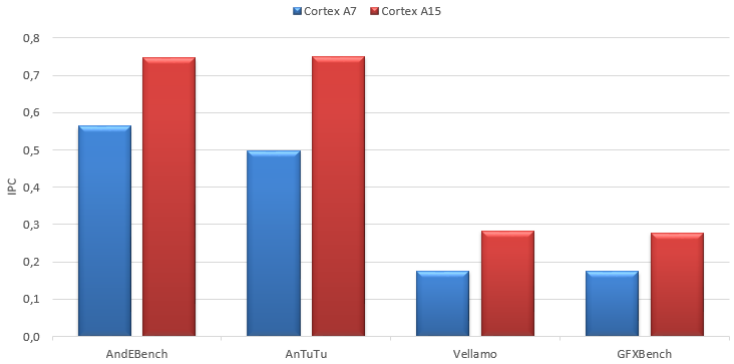


Figura 4.1: Número de instrucciones retiradas por ciclo (IPC) para los distintos benchmarks.

Para monitorizar las métricas de rendimiento consideradas, empleamos la herramienta PMCTrackApp desarrollada en este TFG. PMCTrackApp no sólo simplifica de forma sustancial la configuración de eventos hardware y automatiza la representación gráfica de métricas de rendimiento, sino que también permite almacenar los resultados obtenidos para su posterior procesamiento. En particular, tras la ejecución de cada benchmark, PMCTrackApp genera un fichero de texto con los resultados con las cuentas de eventos hardware obtenidos a lo largo del tiempo. Los datos que se muestran en las figuras 4.1, 4.2, y 4.3 se han obtenido procesando la información almacenada en esos ficheros de texto y capturando la media de cada métrica para la ejecución completa de cada aplicación.

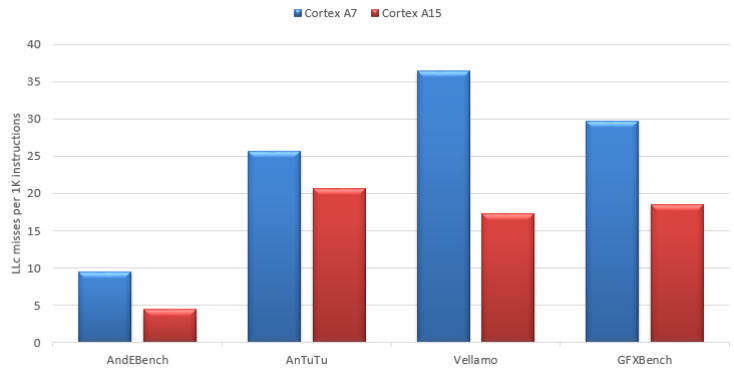


Figura 4.2: Número de fallos de último nivel de cache (LLC) por cada 1K instrucciones retiradas para los distintos benchmarks.

Los resultados revelan que en todos los casos el IPC que se obtiene en los cores Cortex A15 (de alto rendimiento) es mayor que en el Cortex A7 (de bajo consumo) para todos los benchmarks analizados. Esto no supone un resultado inesperado, ya que el pipeline del core de alto rendimiento es más complejo (fuera de orden) que el de bajo consumo (en orden). Asimismo, el cluster de cores Cortex A15 posee una cache L2 compartida de mayor tamaño (2MB) que el cluster de cores Cortex A7 (512KB). Como puede observarse en la figura 4.2 la tasa de fallos de cache que experimentan las aplicaciones analizadas es mayor al usar los cores de bajo consumo (Cortex A7).

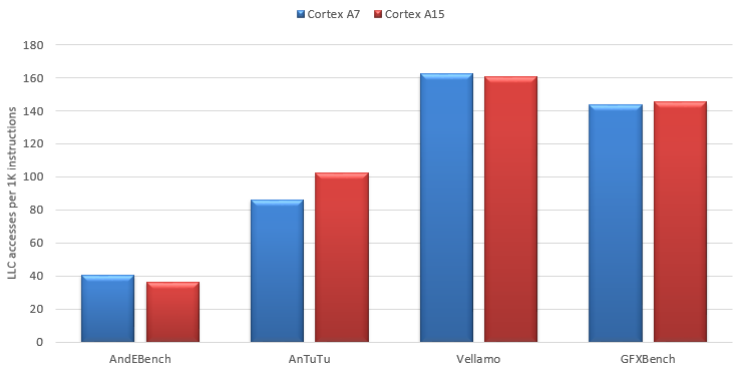


Figura 4.3: Número de accesos al último nivel de cache (LLC) por cada 1K instrucciones retiradas para los distintos benchmarks.

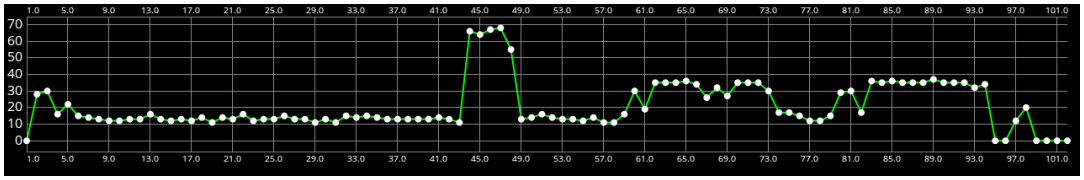
Por otra parte, los resultados revelan que las aplicaciones Vellamo, GFXBench son intensivas en memoria. Como se observa en la figura 4.2 estos benchmarks realizan un número elevado de accesos a memoria (fallos de último nivel de cache) por cada mil instrucciones en los dos procesadores usados. En particular, Vellamo, la aplicación con mayor tasa de fallos de cache, es también la que obtiene un menor número de instrucciones por ciclo, seguida de cerca por GFXBench. Hay que destacar también que GFXBench hace mayor uso de la GPU que la CPU; eso explica el bajo IPC. El caso de Vellamo es similar porque se encarga de enviar y recibir datos del navegador web.

Los datos obtenidos también indican que AndEBench es el benchmark más intensivo en CPU entre todos los considerados, por su baja tasa de fallos de LLC. Asimismo, este programa exhibe IPC más alto que el resto ya que es el que menos accede al último nivel de cache.

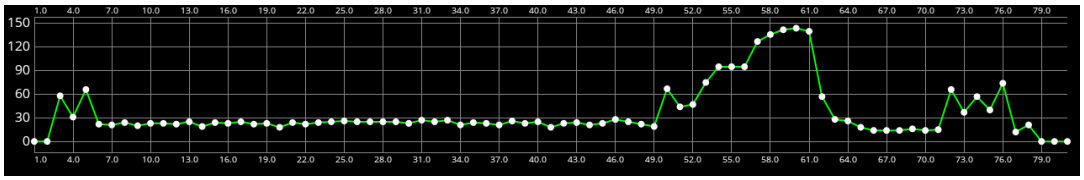
En el caso de AnTuTu se observa que tiene un IPC y una tasa de fallos de LLC relativamente elevados. Porque cuando ejecuta los tests intensivos de la CPU, el IPC aumenta. Asimismo cuando ejecuta los tests de GPU aumenta la tasa de fallos de cache.

4.2. Análisis de aplicaciones con PMCTrackApp

Para mostrar el correcto funcionamiento de PMCTrackApp se procedió a monitorizar el IPC y la tasa de fallos de LLC para el benchmark AndEBench. Las figuras 4.4 y 4.5 muestran estas dos métricas a lo largo del tiempo para AndEBench. Nótese que el valor de IPC que se muestra en la figura está multiplicado por 100. Los resultados indican que la aplicación AndEBench alcanza un IPC máximo de 1.5 en el core de alto rendimiento (Cortex A15), mientras que se observa un valor de IPC máximo de 0.75 en el core de bajo consumo (Cortex A7). En cuanto a la tasa de fallos de cache (figura 4.5) los valores observados en el Cortex A15 son menores que en el Cortex A7. En particular, se observa que la tasa alcanza un máximo de 40 fallos de cache de último nivel en el Cortex A7; el valor máximo de esa tasa se reduce a la mitad en el caso de los cores de alto rendimiento.

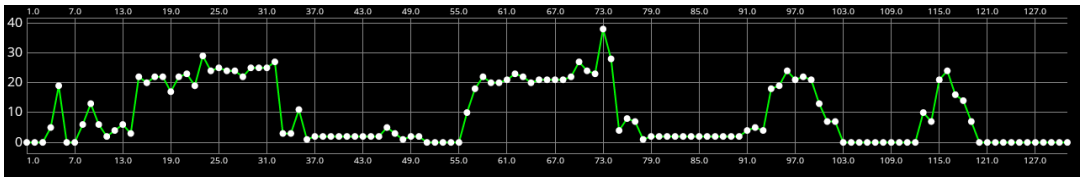


(a) Cortex A7

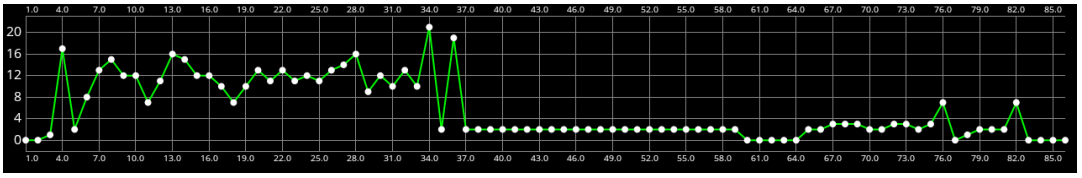


(b) Cortex A15

Figura 4.4: IP100C A7 vs A15



(a) Cortex A7



(b) Cortex A15

Figura 4.5: LLC A7 vs A15

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

Después de la realización de este proyecto, PMCTrack cuenta con tres nuevas características: (1) soporte para funcionar en la plataforma Android, (2) capacidad de iniciar una sesión de monitorización con un proceso que está ya en ejecución y (3) la aplicación PMCTrackApp –un *frontend* gráfico para PMCTrack en Android que facilita su uso y permite obtener gráficas en tiempo real de métricas de alto nivel definidas por el usuario–. En los capítulos 2 y 3, se ha explicado en detalle el funcionamiento y el proceso de diseño e implementación de cada una de estas características. En el capítulo 4, se ha puesto a prueba su funcionamiento y mostrado su gran utilidad mediante dos casos de estudio. Para finalizar, analizaremos ahora a fondo el impacto general de cada una de estas nuevas características.

El nuevo soporte de PMCTrack para la plataforma Android permite explotar las ventajas de esta herramienta en el contexto de los dispositivos móviles. Para incorporar este nuevo soporte ha sido necesario crear un parche específico del kernel de Android para incorporar diversas extensiones que necesita el módulo del kernel de PMCTrack para funcionar. Para la correcta instalación del kernel parcheado en nuestro entorno experimental, la placa Odroid XU4, fue necesario preservar el *vermagic* del kernel de Android original instalado en la placa. De este modo, los módulos del kernel ya instalados en la plataforma, y compilados contra el kernel original, pueden cargarse correctamente con el kernel modificado. Para preservar el *vermagic* del kernel, optamos por modificar el *Makefile* principal del kernel de Android. Las modificaciones de este *Makefile* se encuentran en el parche de PMCTrack para el kernel (versión 3.10.96, para la placa Odroid XU4) creado durante este TFG, que ya se encuentra integrado en la rama principal de PMCTrack.

Por otra parte, hemos incorporado en PMCTrack la capacidad de iniciar sesiones de monitorización con procesos que se encuentran en ejecución. La implementación de esta nueva característica ha precisado la modificar tanto la herramienta `pmctrack` de línea de comandos, como el módulo del kernel de PMCTrack (soporte de bajo nivel para modo `attach`). Esencialmente, para poder iniciar una sesión de monitorización con un proceso con PID conocido, es preciso iniciar una sesión individual de monitorización con

cada uno de los hilos del proceso, y garantizar que los nuevos hilos que el proceso vaya creando también sean monitorizados. A pesar de la relativa complejidad de los cambios efectuados a bajo nivel para llevar esto a cabo, el usuario final puede iniciar la sesión de monitorización de forma sencilla. Para ello basta con especificar el PID del proceso a monitorizar como parámetro de la nueva opción `-p` del comando `pmctrack`. Gracias a este nuevo soporte, el usuario puede tener una visión global de todos los datos de rendimiento de todos los hilos involucrados en el funcionamiento de la aplicación monitorizada.

Finalmente, la aplicación PMCTrackApp para Android, desarrollada en Java, facilita en gran medida el uso de PMCTrack para usuarios menos experimentados. Cabe destacar que el comando `pmctrack` (usado internamente por PMCTrackApp) requiere especificar muchos detalles, como los eventos hardware, algunos de los cuales requieren tener cierto conocimiento de la unidad de monitorización del rendimiento (PMU) integrada en el procesador. Esto puede suponer una barrera para numerosos usuarios de Android y también para los desarrolladores de aplicaciones de usuario. Aún siendo la temática de la monitorización con contadores hardware ciertamente no apta para todo tipo de usuarios, con PMCTrackApp esperamos haber resuelto, o, al menos, rebajado ese obstáculo y conseguir que un usuario, con ciertas nociones mínimas del funcionamiento del hardware en un smartphone/tablet, se centre en su objetivo principal: obtener datos de monitorización del rendimiento de las aplicaciones. Con mucha frecuencia, se precisa la construcción de gráficas a partir de los datos obtenidos de la monitorización. PMCTrackApp ofrece un soporte completo en este aspecto, ya que construye gráficas en tiempo real con los valores de las métricas de rendimiento que el usuario está monitorizando para una aplicación. Además, ofrece la posibilidad de guardar los resultados obtenidos durante la monitorización.

5.2. Valoración del TFG

Para la realización de este proyecto, ha sido necesario trabajar en tres niveles muy distintos: programando unas veces en C al nivel del kernel del sistema operativo; otras veces en C a nivel de usuario, cuando estuvimos desarrollando las extensiones del programa `pmctrack` para la monitorización de un programa que ya se encuentra en ejecución; y otras en Java, para desarrollar la aplicación gráfica PMCTrackApp sobre Android.

La particularidad de haber sido un proyecto muy transversal, con esta gran variedad de niveles de abstracción, creemos que ha aumentado sensiblemente su dificultad. De hecho, durante el transcurso del proyecto, hemos tenido que documentarnos profundamente para poder comprender la interacción entre cada uno de estos niveles.

A continuación, listamos los aspectos más relevantes sobre los que hemos tenido que estudiar y entender:

- El funcionamiento y la documentación de los contadores hardware de cada fabricante, arquitectura y modelo.
- El funcionamiento interno del kernel Linux.

- La arquitectura interna de la herramienta PMCTrack (componentes de espacio de usuario y espacio de kernel).
- El desarrollo de aplicaciones en Android.
- Diversas librerías externas de Java usadas en el desarrollo de PMCTrackApp: mChart, Interpreter, ...

Para concluir, creemos que hemos hecho una importante aportación a la herramienta PMCTrack y esperamos que resulte de gran utilidad para la comunidad de usuarios de esta herramienta.

5.3. Trabajo futuro

Seguidamente, presentamos una lista de posibles ampliaciones de PMCTrack que se podrían añadir al trabajo realizado en este TFG en el futuro:

- Creación de un *widget* para poder visualizar las gráficas de monitorización en Android desde el *Launcher* de Android.
- Soporte para otras arquitecturas hardware (más allá de x86 y ARM)
- Capacidad para salvar configuraciones de contadores en ficheros para su carga posterior en PMCTrackApp.
- Soporte para almacenar no sólo la salida de `pmctrack` en un fichero, sino también incorporar en éste los valores de las métricas de rendimiento que el usuario de PMCTrackApp ha solicitado mostrar gráficamente.

Apéndice A

Introduction

Most modern complex computing systems are equipped with hardware Performance Monitoring Counters (PMCs) that enable users to collect application's performance metrics, such as the number of instructions per cycle (IPC) or the Last-Level Cache (LLC) miss rate. These PMC-related metrics aid in identifying possible performance bottlenecks, thus providing valuable clues to programmers and computer architects. Notably, direct access to PMCs is typically restricted to code running at the OS privilege level. Thus, a kernel-level tool, implemented in the OS itself or as a driver, is usually in charge of providing user-space tools with a high-level interface enabling to access performance counters [21, 5, 1].

Previous work has demonstrated that the OS scheduler can also benefit from PMC data making it possible to perform sophisticated and effective run time optimizations on multi-core systems [7, 20, 23, 9, 24, 8, 15, 12, 17]. Special attention has been paid to optimizations in the OS scheduler. Notably, many of the proposed PMC-based OS scheduling schemes rely on per-thread high-level metrics that are estimated by means of platform-specific prediction models [8, 15, 12, 17, 3, 4]. In this scenario, determining the necessary per-thread high-level metrics (e.g., energy efficiency or performance ratios across cores) at run time entails monitoring a specific set of hardware PMC events that may differ substantially across processor models and architectures [8, 15].Unfortunately, public-domain PMC monitoring tools, which are largely user-space oriented, do not provide an architecture-independent mechanism that enables feeding PMC-based OS scheduling schemes with the necessary high-level monitoring information they require to function. Due to the limited support for in-kernel monitoring in public-domain PMC tools, some researchers have employed architecture-specific *ad-hoc* code to access performance counters in the scheduler implementation [7, 8, 15]. However, this approach still leads the scheduler to be tied to certain processor models. Other researchers have resorted to evaluating their proposals by means of simplistic user-space scheduling prototypes [23, 24, 12, 4] that rely on existing user-space-oriented PMC tools. To overcome these limitations, we propose PMCTrack, an OS-oriented PMC tool for the Linux kernel. PMCTrack is novelty lies in

the `monitoring module` abstraction, a platform-specific component that is responsible for collecting the necessary high-level metrics that a given OS scheduling algorithm requires to function. This abstraction makes it possible to create architecture-independent implementations of OS scheduling algorithms that leverage PMC data.[18]

Despite being an OS-oriented tool, PMCTrack is also equipped with a set of command-line tools and user_space components to assist OS-scheduler designers during the entire development process. These user-space tools, by complementing the existing kernel-level debugging tools with PMC-related off-line analysis and tracing support. Moreover, due to the flexibility of PMCTrack's monitoring modules, any kind of metric provided by modern hardware but not modeled directly via performance counters, such as power consumption or an application's cache footprint, can be also exposed to the OS scheduler or to the user applications as PMCTrack's *virtual counters*. In a previous work [18] are discussed the PMCTrack advantages offers over other performance monitoring tools.

The rest of this chapter is structured as follows. Section A.1 PMCTrack internal architecture and the different modes of use of the tool is presented. Section A.2 illustrates the motivation for this work. Section A.3 briefly describes the experimental environment used in this project. In sections A.4 and A.5 project goals and work plan are presented.

A.1. Design PMCTrack

A.1.1. Architecture

Figure A.1 depicts PMCTrack internal architecture. The tool consists of a set of user and kernel space components. Essentially, the end user interacts with PMCTrack using the available command-line tools or PMCTrack-GUI (a graphical front-end). Alternatively, applications may access PMCTrack's functionality directly via the libpmctrack user space library. These components communicate with thePMCTrack's kernel module by means a set of inputs /proc Linux exported by the module. The kernel module implements the vast majority of PMCTrack's functionality. To gather per-thread performance counter data, the module needs to be fully aware of thread scheduling events (e.g., context switches, thread creation/termination). In addition to exposing an application's performance counter data to the user tools, the module implements a simple mechanism to feed with per-thread monitoring data to the scheduling policy that requires performance-counter information to function. Because both the core Linux Scheduler and scheduling classes are implemented entirely in the kernel, making PMCTrack's kernel module aware of the thread-related events and request from the SO scheduler requires some minor modifications to the Linux kernel itself. These modifications, referred to a "PMCTrack kernel AP" in Figure A.1, comprise a set of notifications issued from the core scheduler to the module. To receive key notifications, PMCTrack's kernel module implements the operations interface `pmc_ops_t` [18].

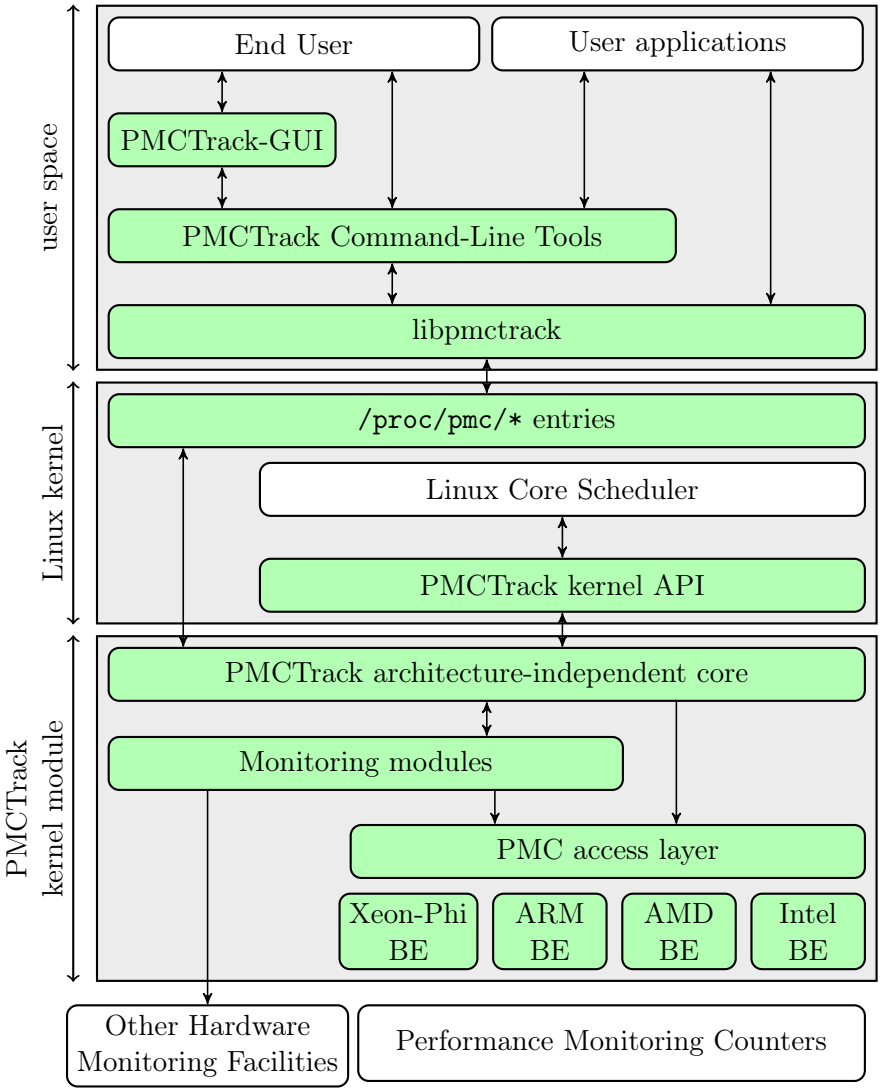


Figure A.1: PMCTrack architecture

As shown in Figure A.1, PMCTrack's kernel module consists of various components. The architecture-independent core layer implements the `pmc_ops_t` interface, and interacts with the `pmctrack` command-line tool via the Linux proc file system. PMCTrack's kernel module also provides an API to build *monitoring modules*. As stated above, the primary purpose of a monitoring module is to provide a scheduling algorithm that is implemented in the kernel with high-level performance metrics or another insightful runtime information that is potentially exposed by the hardware, such as power/energy consumption. In addition, a monitoring module may expose this information to PMCTrack's user-space components by means of virtual counters. The architecture-independent component relies on a Performance Monitoring Unit Backend (PMU BE) to carry out low-level access to the PMCs, as well as for translating user-provided counter configuration strings into internal data structures for the platform in question. Currently there are back-ends compatible with most modern processors from Intel and AMD. Recently it has added support for Intel Xeon Phi co-processor, and for different models of processors ARM Cortex family.

A.1.2. Usage Modes

PMCTrack can be used to gather performance counter data from the OS scheduler (using an in-kernel interface) and from user space.

A.1.2.1. Accessing PMC data from the OS scheduler

This mode enables any scheduling algorithm in the kernel (i.e., scheduling class) to collect per-thread monitoring data, thus making it possible to drive scheduling decisions based on tasks memory behavior or other runtime properties. Turning on this mode for a particular thread from the scheduler's code simply involves activating a flag in the thread's descriptor [18]. A scheduling algorithm relying on PMCTrack typically enables in-kernel monitoring for all threads belonging to its scheduling class.

To ensure that the implementation of the scheduling algorithm that benefits from this feature remains architecture independent, the scheduler itself (implemented in the kernel) does not deal with performance counters or hardware events directly, but instead requests the necessary per-thread high-level performance monitoring metrics from a platform-specific *monitoring module*, such as the number of instructions per cycle (IPC) or the miss rate of cache.

PMCTrack may include several monitoring modules that are compatible with a given platform. However, only one can be enabled at a time: the one that provides the scheduler with the PMC-related information it requires to function. In the event that several compatible monitoring modules are available, the system administrator may tell the sys-

tem which one to use by writing in the `/proc/pmc/mmon_manager` file. The scheduler can communicate with the active monitoring module to obtain per-thread data via the following function from PMCTrack's kernel API:

```
int pmcs_get_current_metric_value(struct task_struct*
   task, int metric_id, uint64_t* value);
```

For simplicity, each metric is assigned a numerical ID, which is known by the scheduler and the monitoring module. To obtain up-to-date metrics, the aforementioned function may be invoked from the tick processing function in the scheduler.

Monitoring modules make it possible for a scheduling policy relying on performance counters to be seamlessly extended to new architectures or processor models as long as the hardware enables the collection of necessary performance data. All that needs to be done is to build a monitoring module or adapt an existing one to the platform in question. From the programmer's standpoint, creating a monitoring module entails implementing an operations interface. Specifically, it consists of several callback functions that make it possible to notify the module on activations/deactivations requested by the system administrator, on threads context switches, every time a thread enters/exits the system, whenever the scheduler requests the value of a per-thread PMC-related metric, etc. Nevertheless, the programmer typically implements the subset of callbacks required to carry out the necessary internal processing. Notably, in doing so, the developer does not have to deal with performance-counter registers directly. Specifically, the programmer indicates the configuration of the desired counter (encoded in a string) using the development of PMCTrack kernel module API. Having a new samples of PMCs by a thread, a callback function monitoring module is invoked, passing the samples obtained as a parameter.

A.1.2.2. Using PMCTarck from user space

In addition to the in-kernel mechanism presented above, PMCTrack also enables the gathering of PMC data from user space by using the `pmctrack` command-line tool and `libpmctrack`.

The command `pmctrack` allows the user to gather an application's performance data at regular time intervals (a.k.a., time-based sampling - TBS) or when a certain event count reaches a specified threshold (a.k.a., event-based sampling - EBS).Notably, both modes support monitoring multi-threaded and single-threaded applications and provide information on performance counters as well as on any monitoring information exposed by the active monitoring module as a virtual counter, such as energy consumption readings. This command allows to specify counters and events configurations using mnemonic

in the same way as other existing tools oriented at monitoring from user space[1, 5, 11]. To illustrate how the `pmctrack` tool works, let us consider the following sample command for the TBS (default) mode:

```
$ pmctrack -c instr,llc_misses ./mcf06
[Event-to-counter mappings]
pmc1=instr
pmc2=llc_misses
[Event counts]
nsample    pid      event          pmc1          pmc2
      1  11960       tick     797055120       8912054
      2  11960       tick     316689383      11242700
      3  11960       tick     282149642      10327292
      4  11960       tick     274180995      10164450
      5  11960       tick     259539536       9709397
      6  11960       tick     241565909       9274640
      7  11960       tick     233002034       9008892
      8  11960       tick     234823905       9007262
...
```

This command provides the user with the number of instructions retired and last-level cache (LLC) misses every second (default setting for the configurable sampling period) on Odroid XU4 (described in section A.3), integrating a ARM processor big.LITTLE. The option -c must be used to indicate the sets of hardware events to monitor. As is evident, the command-line tool makes it possible to specify counter and event configurations using mnemonics in much the same way as other user-space-oriented tools [5, 1, 11]. The beginning of the command output shows the event-to-counter mapping for the various hardware events. The "Event counts" section in the output displays a table with the raw counts for the various events; each sample (one per second) is represented by a different row. At the end of the line, we specify the command to run the associated application we wish to monitor (e.g., ./mcf06).

To complement the `pmctrack` command-line tool with real-time visualization of high-level performance metrics (such as the *IPC* or the *LLC* miss rate) we also created PMCTrack-GUI, a Python front-end for `pmctrack`. Figure A.2 shows a screen-shot of PMCTrack-GUI. This application extends the capabilities of the PMCTrack stack whit other relevant features, such as an SSH-based/ADB-based remote monitoring mode or the ability to plot user-defined performance metrics.

Libpmctrack enables the characterization of the performance of code fragments via PMCs in sequential and multi-threaded programs. To this end, libpmctrack's API offers a set of calls to indicate the desired PMC configuration to the kernel module at any point in the application's code. The programmer may then retrieve the associated event counts for any code snippet (either via *TBS* or *EBS*) simply by enclosing the code between invocations to the `pmctrack_start_counters()` and `pmctrack_stop_counters()` functions.

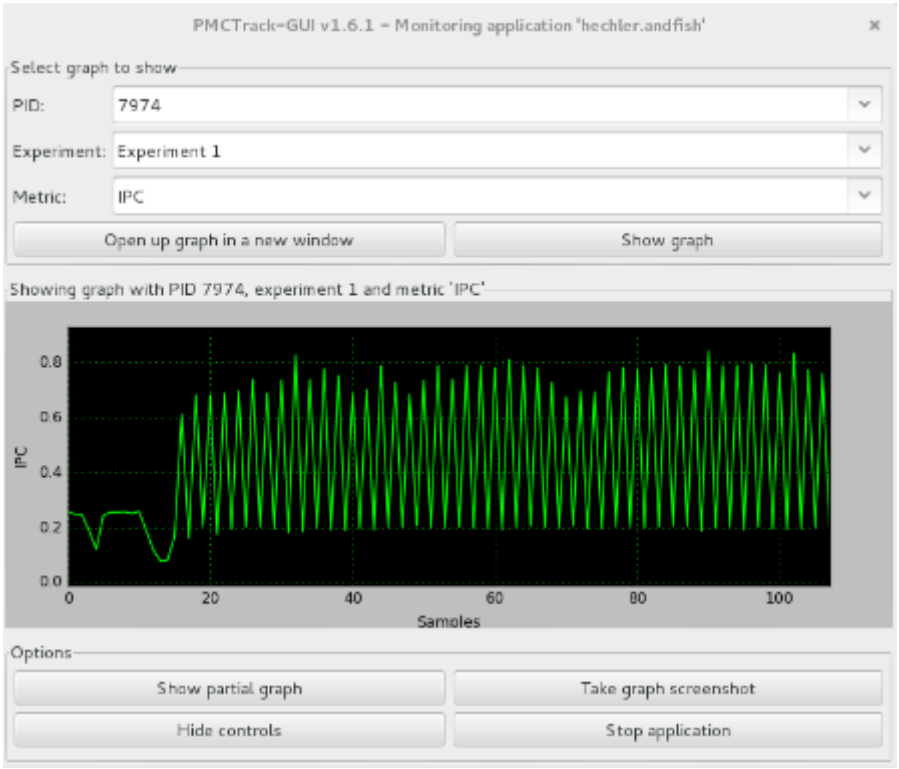


Figure A.2: PMCTrack-GUI

Libpmctrack and pmctrack command allow the user to collect PMC application data, but also have the ability to provide users with any other relevant information exported by the monitoring active module as a virtual counter. To provide PMCs and virtual counters information to libpmctrack and pmctrack, PMCTrack kernel module stores this information on ring buffers in kernel space. The PMCTrack user mode tools consume these buffers information reading input /proc exported by the kernel module.

A.2. Motivation

Today's, the massive use of the Android operating system (based on the Linux kernel) in the vast majority of mobile devices, makes it attractive to end users, application developers and device manufacturers to have access to performance tools for monitoring based hardware counters for this system.

Currently, Android users can access to hardware counters information by Linux kernel subsystem PerfEvents [21], which has the perf tool (command line) as front-end. However, the use of PerfEvents in Android presents two major limitations. First, the perf command line tool is not a good choice for inexperienced users. Essentially, access to it requires to have installed an application terminal emulation on the mobile device or a connection via ADB to it (e.g., `adb shell`). Second, the PerfEvents kernel subsystem does not offer a robust support and independent architecture for accessing to performance monitoring information (or energy consumption) from different subsystems kernel, as memory manager or Scheduler [18]. This makes the optimizations based on hardware counters inside the system difficult to perform, which they are also easily expandable to different architectures or processor models.

The PMCTrack tool has been specifically designed to resolve the second limitation. Moreover, the graphical front-end PMCTrack-GUI is easily adaptable for monitoring any Android device via ADB. Unfortunately, there is not a functional version of PMCTrack for Android. Therefore, the potential benefits of using this tool in the operating system still unknown. Analyze these benefits is the main motivation of this work.

A.3. Experimental environment: Odroid XU4 board

To develop the Final Project has been used Odroid XU4 board. This board supports different versions of Android, making it a very suitable platform to carry out our work.

Featuring an octa-core Exynos 5422 big.LITTLE processor(4-core ARM Cortex A15 ARM Cortex A7 4) operating at 2GHz maximum, advanced Mali GPU T628 MP6 (compatible with OpenGL ES 3.0/2.0/1.1 and OpenCL 1.1), for this reason the creators have implemented an active cooling as you can be seen in Figure A.4. Odroid XU4 board incorporates 2GB of RAM LPDDR3 (PoP Stacked, integrated into the PCB) and a memory flash controller eMMC 5.0 of 8-bit connector for external modules eMMC memory, so it does not incorporate series integrated storage. However, eMMC supports both modules as microSD cards. Thus, as in the Raspberry Pi it is possible to install an operating

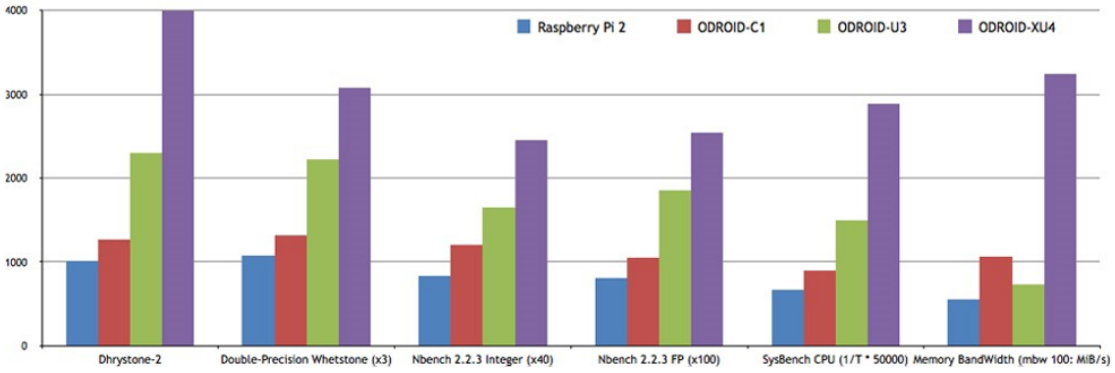


Figure A.3: Performance Odroid XU4 comparison

system on an external card without problem. To end the technical section, we should mention that incorporates two USB 3.0 ports and one USB 2.0, addition to the aforementioned microSD card reader. It also has a Gigabit network card and HDMI video output.

The Figure A.3 show a comparative performance of several boards in which several benchmarks executed. Note that the computing potency of the Odroid XU4 was between 3 and 4 times faster than the Raspberry Pi 2 due to 2GHz of Cortex A15 and higher memory bandwidth.

The uses that we can give this mini PC (board development) are very broad, we can do basically the same things that with Raspberry Pi but with greater power and versatility. For this reason Odroid XU4 is the best lab for us.

Odroid XU4 can run multiple versions of Linux, including Ubuntu 15.04, Android 4.4 KitKat and 5.0 Lollipop.

We can compile Android kernel and burn his image from the SD or eMMC card, develop applications and see how it works on this board, along with a screen and a mouse you have already a device with Android very powerful and independent and follow the execution of the application. It can be connected by serial port and also by ADB for passings applications to Odroid XU4.

Odroid XU4 haven't a massive community behind it. We don't have so many tutorials and people to ask for help that Pi for example. Odroid XU4 don't have USB OTG, Power Monitoring Sensors to measure the power consumption of Big CPU, Little CPU, GPU, DRAM in real time.

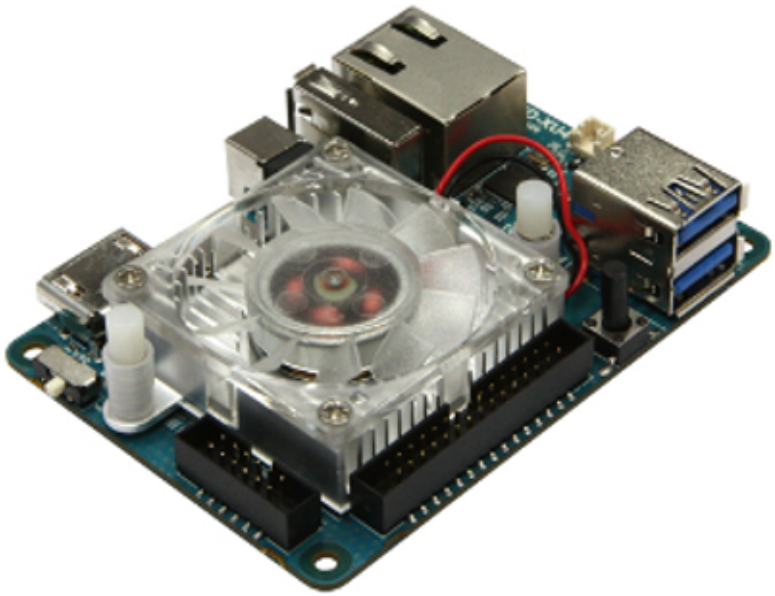


Figure A.4: Odroid XU4

A.4. Project goals

The PMCTrack tool has been designed specifically for the GNU/Linux OS. Its small library dependencies and external tools allow that can be used in most Linux distributions. The goal of this project is an adaptation of PMCTrack the Android OS, using the Odroid XU4 board as development and evaluation platform. Accomplishing this adaptation involves performing the following tasks:

1. Changing the Linux kernel own variant of Android to include extensions required by the kernel module PMCTrack.

2. Adapting user mode tools and PMCTrack kernel module for the proper functioning of the tool in Android.

3. Development of Android (Java) application, called PMCTrackApp, which can display real-time measurements collected counters for different applications being monitored.

As shown in the next chapter, the user mode tools and PMCTrack kernel module can be compiled easily for Android/ARM using a cross compiler. These tools work without modification to the source code on any Android kernel incorporating PMCTrack patch for Linux. In this TFG a specific patch for kernel version of Android with support for Odroid XU4 board was created. This patch gives developers access to the kernel API PMCTrack kernel, which facilitates the development of scheduling algorithms based on hardware counters [18].

Although user mode tools and kernel module PMCTrack not require modification to run on Android, the substantial differences between GNU/Linux and Android make these software components do not provide adequate support monitoring from user space Android. This stems from the fact that not originally PMCTrack allows monitoring of applications that are already running. In fact, for information on performance metrics (e.g., IPC or cache miss rate) of a program was needed to launch the program using the `pmctrack` command line or tool to implement the program code using libpmctrack. Android, users do not have the source code for most applications, or launch applications from a terminal window, but use for this the application icon in the Android *Launcher*. To provide adequate support Android has been necessary to include the *attach* way PMCTrack during this Final Project. This new mode allows the end user to log monitoring with an application that is already running. As described in Chapter 2, the inclusion of this new functionality has required changes in both `pmctrack` line commands tool (new -p option) as in the kernel module. More specifically, the Figure A.5 indicates which components of the architecture of PMCTrack have been modified during this TFG (marked in red).

A.5. Work plan

To achieve the project goals, described above, the project development consisted of the following steps:

- Reading documentation in Android application development.
- Learning development environment on the Odroid XU4 board using ADB and Android Studio.
- Creating Track PMC patch for the kernel of Android.
- Android kernel installation modified in Odroid XU4 board.
- Generation of binary components Android user space and kernel space PMCTrack unmodified and installation of these components on the Odroid XU4 board.
- Making changes in command-line tools and kernel module PMCTrack to provide additional functionality needed in Android.
- Design user interface PMCTrackApp application.
- Implementation of PMCTrackApp application.
- Performing experimental analysis on the Odroid XU4 board using PMCTrack version created for Android.

It is worth noting that the order of these steps is merely illustrative, since such steps were completed in a strictly sequential way. Specifically, when we implemented PMCTrackApp has been assigned different tasks to each member to work in parallel and move faster with the implementation.

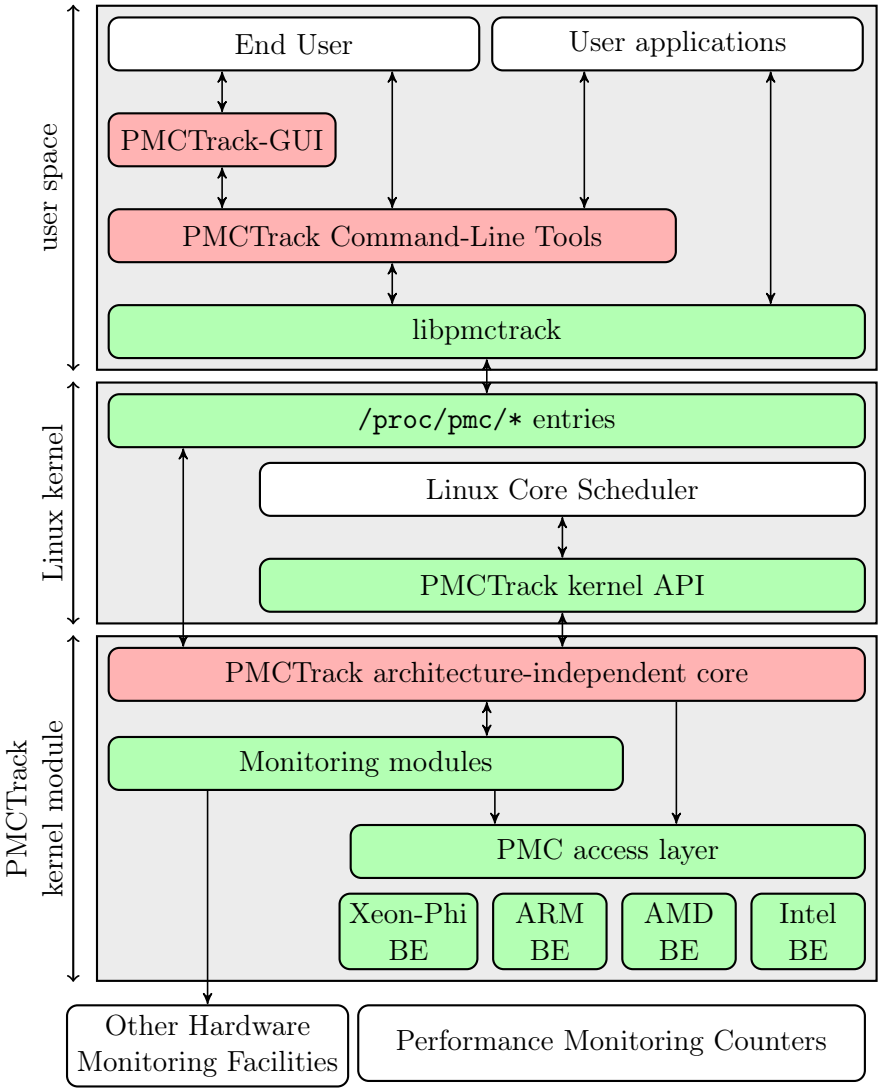


Figure A.5: Changes in the architecture of PMCTrack

Apéndice B

Conclusions and future work

B.1. Conclusions

After the completion of this project, PMCTrack has three brand new features: (1) support to run on the Android OS, (2) the ability to start a monitoring session with a process that is already running and (3) the PMCTrackApp application –a graphical *front-end* for PMCTrack on Android to facilitate the use of PMCTrack and to enable real-time visualization of the high-level metric values gathered by the user–. In Chapters 2 and 3, we explained in detail the inner workings, design process and implementation of each of these features. In Chapter 4, we illustrate their capabilities and effectiveness by analyzing two case studies. Now we will discuss the impact of the various PMCTrack enhancements carried out in this project.

The PMCTrack support for the Android platform allows to leverage the potential of this tool in the context of mobile devices. To incorporate this new support into the tool, we had to create a specific patch for the Android kernel. This patch includes various extensions that the PMCTrack's kernel module requires to function. For a successful installation of the patched kernel in our experimental platform, the Odroid XU4 board, we had to compile the kernel in a special way so as to preserve the *vermagic* of the original kernel installed on the board. This was necessary to ensure that the various kernel modules found on the Android system, which where compiled against the "old" kernel, could be loaded successfully on top of the modified kernel. Setting a specific *vermagic* in the modified kernel, entailed to make changes to the Makefile found in the root directory of the Linux kernel source tree. Modifications made to this *Makefile* can be found in the PMCTrack patch for the kernel (version 3.10.96, for Odroid XU4 board) created during this project. This patch is already distributed along with the official sources of the PMCTrack tool [14].

Moreover, we have augmented PMCTrack with the ability to a start monitoring sessions with any process that is already running on the system. The implementation of this new feature requires making changes to both the `pmctrack` command-line tool and the PMCTrack's kernel module (low-level support for the `attach` mode). Essentially,

to start a monitoring session with a process identified by its PID, PMCTrack starts an individual monitoring session with each and every thread in the process. At the same time, the tool ensures that newly created threads in the process are also monitored. Despite the relative complexity of the low-level changes required by the new "attach" feature, the end user can start monitoring sessions in a seamless fashion. To this end, the user must simply specify the PID of the process to be monitored as a parameter of the new `-p` switch of the `pmctrack` command. Thanks to this new support, the user can have a global view of the performance data of all the threads involved in the operation of the monitored application.

Finally, the PMCTrackApp application for Android, developed in Java, greatly facilitates the use of PMCTrack, especially for inexperienced users. Note that the `pmctrack` command (used internally by PMCTrackApp) requires the user to specify many details, such as the hardware events. To be able to specify many of these details, the user needs to have some basic knowledge on the processor's performance monitoring unit (PMU). This can be an important burden for many Android users and also for application developers. Even though hardware monitoring is certainly not suitable for all types of users, we hope that PMCTrackApp makes it possible for most users, with basic knowledge on performance monitoring, to start monitoring sessions easily on cell phones or tablets. Typically, creating graphs with the gathered monitoring data makes it possible to gain further insight into the collected data. PMCTrackApp offers a comprehensive support on this aspect, as it builds charts in real time with the values of the various performance metrics that the user is monitoring for an application. At the same time, the results obtained during the monitoring session can be saved on disk.

B.2. Evaluation of the project

This project has involved to work on three very different levels. We had to program sometimes in C at kernel level, sometimes also in C but at user level, when we were developing extensions `pmctrack` program for monitoring a program that is already running, and some other times in Java at graphical level.

This particularity makes this a multilevel project and we think that this has been the greatest difficulty. Therefore, we had to read documentation for all these levels and we had to study carefully all the interactions between them.

We list below the most relevant aspects we had to study about:

- The internal behavior of PMCs and the documentation of them for each vendor, architecture and model.
- The structure and code of Linux kernel.
- The internal architecture of PMCTrack tool (components of user space and kernel space).
- Developing Android applications.

- Several external Java libraries used on the development of PMCTrackApp: mChart, Interpreter, ...

In summary, we think that we have done important improvements and additions to the PMCTrack tool and we hope this work will be useful inside and outside of our university.

B.3. Future work

Following, we present a list of all possible extensions that could be added in the future to PMCTrack:

- Creating a *widget* to view monitoring graphs on Android from the Android *Launcher*.
- Support for other hardware architectures (beyond x86 and ARM).
- To allow saving and loading configurations for PMCTrackApp.
- To allow saving the output for the custom metrics defined by the user in PMCTrackApp. Right now, it saves the standard output of `pmctrack` only.

Apéndice C

Diagrama del diseño de PMCTrackApp

En este apéndice incluimos el diagrama que hemos usado durante el diseño de nuestra aplicación PMCTrackApp, y que sin duda, servirá al lector para un mayor entendimiento de la aplicación.

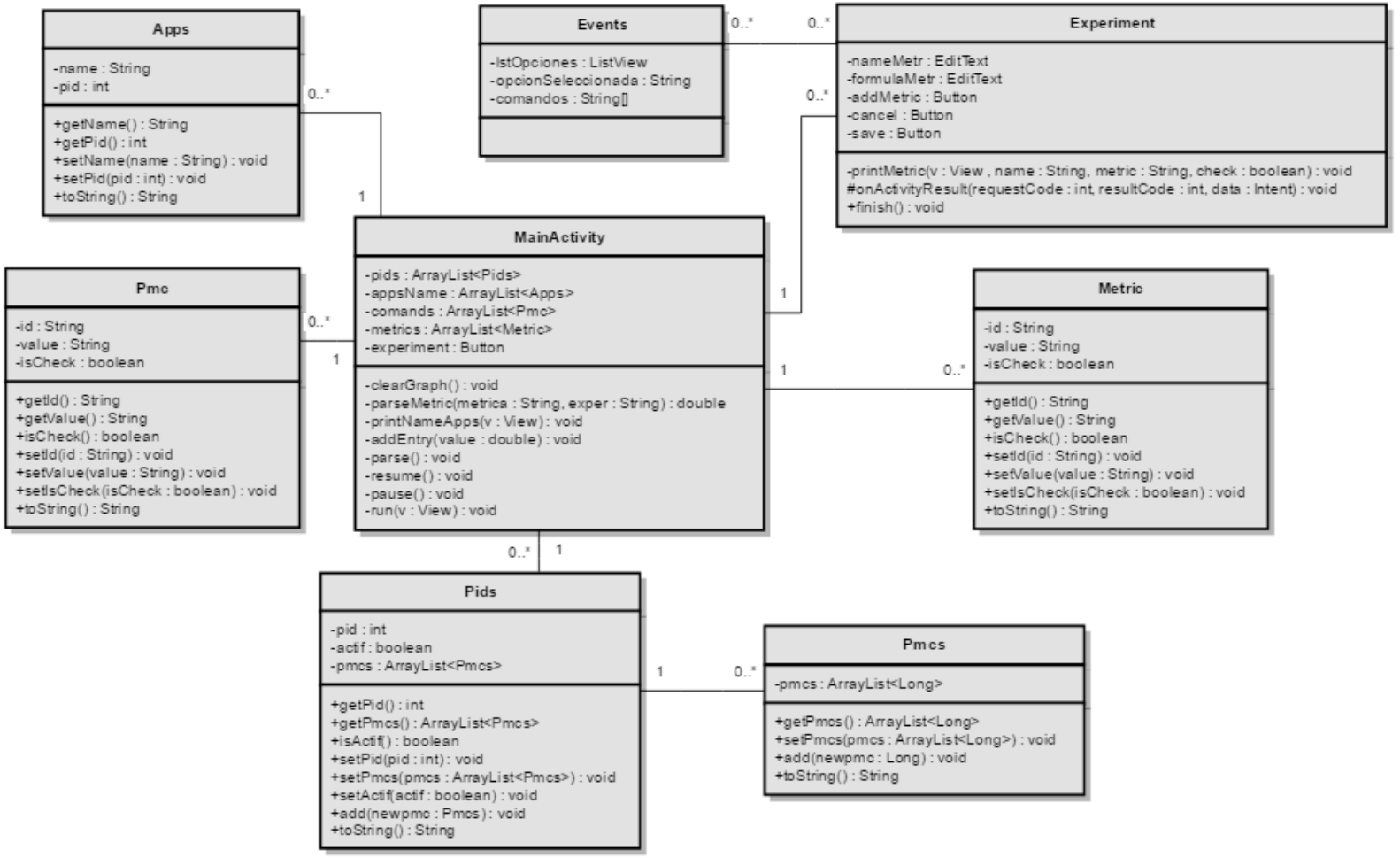


Figura C.1: Diagrama de clases de PMCTrackApp

Apéndice D

Contribuciones de cada participante

En este apéndice cada participante indicará su contribución al proyecto en su respectiva sección.

D.1. Contribución de Luis Javier Cabrera Sagbay

Mi contribución ha estado presente en todas las fases del proyecto ya que hemos optado por trabajar en grupo para poder trabajar de forma consistente. Debido a esta forma de trabajar hemos podido resolver todos los problemas que iban saliendo de manera ordenada y nos hemos ido enterando de todas las partes de nuestro trabajo.

He tenido que leer documentación sobre:

- **PMCTrack**: Debido a que el concepto de monitorización mediante contadores hardware o PMCs ha sido nuevo para mi he tenido que leer la memoria [6] del año pasado, la información expuesta en la página web de PMCTrack. Ya que había que añadir la funcionalidad de pmctrack en el sistema Android he tenido que leer los distintos códigos fuente de pmctrack de las distintas capas.
- **Android**: Tuve que leer mucha documentación para entender la arquitectura de Android, tipos de componentes de una aplicación Android (Activity, Service, Broadcast receiver, Content provider), Android SDK, lanzamiento de las aplicaciones Android (Zygote) y la instalación de las aplicaciones Android.
- **Odroid XU4**: Para poder trabajar en la placa Odroid XU4 he tenido que leer las características de la placa, conocer cómo instalar un kernel en la placa, cómo conectarse y transferir archivos entre la placa y el *host* de desarrollo, y cómo instalar la aplicación PMCTrackApp.
- **Android Studio**: Tuve que ver muchos tutoriales y leer la documentación sobre Android Studio para entender la forma de programación en ese entorno, para poder llevar a cabo la instalación segura de PMCTrackApp en nuestra placa y como usar distintas librerías de Java.

He tenido que hacer modificaciones en distintas capas descritas en el Capítulo 2 para que pmctrack pueda monitorizar las aplicaciones por PID con la opción `-p`. Debido a esas modificaciones se encontró el bug de los futex.

He tenido que crear el parche del kernel de Android modificando distintos archivos fuente del kernel para que funcione PMCTrack. Para ello había que saber que modificaciones hay que hacer y en que líneas de código. Por lo cuál había que entender el código en C de cada archivo del kernel de Android. Compilar el kernel modificado con el mismo ID mencionado en el Capítulo 2. Para ello he tenido que usar algo parecido a la ingeniería inversa para ver como asigna ese ID (modificación de Makefile). También he buscado en distintas páginas web para ver en qué zona de memoria de la Odroid XU4 se encuentra el kernel para poder instalarlo en caliente.

Tras entender el método de programación de Android me he puesto a crear posibles interfaces. Me encargué especialmente de desarrollar la estructura básica de las ventanas de configuración del usuario, basadas en los bocetos que habíamos realizado previamente. He tenido que investigar la forma de parsear la salida del comando `pmctrack` para poder pasar los datos de líneas de texto a una estructura llamada *pmcs*. De dicha estructura mi compañero sacará los valores de los PMCs para poder parsear la métrica y obtener el resultado numérico de esa métrica.

Me he encargado de todo lo que tiene que ver con las configuraciones de los contadores y de las métricas, dichas configuraciones forman un experimento. Para que dichas configuraciones se puedan usar he creado las estructuras necesarias. Los eventos asignados a contadores y las métricas introducidas por el usuario se pasan a la actividad principal para que mi compañero pueda llevar a cabo tanto el parseo como el muestreo de las métricas. Cabe destacar que el paso de datos entre actividades se hace de una forma especial en Android Studio.

Hemos tenido muchas reuniones entre nosotros para poder acoplar lo mio con lo de mi compañero y viceversa. En una de ellas discutimos cómo mejorar el diseño de nuestra aplicación. De esa reunión salimos con varias mejoras entre ellas la de poder capturar el grafo y cambiar el nombre del botón *START* a *CHANGE*. El cambio de nombre del botón ofrece la posibilidad de cambiar la métrica que hemos elegido inicialmente cuando se arranca el comando `pmctrack`. Para poder elegir las diferentes métricas creadas por el usuario, se guardan todos los valores de los PMCs asignados por el usuario. Por otro parte, cuando el usuario cambia de métrica se borra el grafo y se empieza a leer los PMCs, se parsea la métrica y se muestra. Lo que hemos ganado con el botón *CHANGE* es sacar las distintas métricas del mismo proceso a la vez con la restricción de que Android no es multiventana.

Finalmente, pusimos a prueba la aplicación PMCTrackApp mediante la realización de casos de estudio de varios *benchmarks*. La monitorización de los *benchmarks* nos ha generado varios archivos extensos de valores de las métricas, en algunos más de 40.000 líneas. Para poder realizar el análisis de los resultados hemos tenido que buscar un programa que pudiera procesar grandes cantidades de datos. En principio hemos visto varios tutoriales de *Pandoc* pero al final hemos usado *XLSTAT* que es un complemento de *Excel*. Para poder mostrar las medias en gráficas hemos usado *Excel*.

D.2. Contribución de Youness El Guennouni

Hemos trabajado en grupo en todas las tareas del proyecto de esta manera hemos asegurado el desarrollo de un proyecto consistente y de calidad. Una vez hemos llegado al desarrollo de la aplicación, ahí hemos optado por dividir el desarrollo entre dos secciones. Esta forma de trabajo nos ha beneficiado ya que los dos conocemos a las distintas partes del proyecto.

En las siguientes subsecciones desarrollaré de forma más detallada mis contribuciones, agrupándolas en los siguientes puntos:

1. Estudio de documentación
2. Uso de la placa
3. Modificación de los archivos fuente de pmctrack
4. Compilación e instalación del kernel
5. Diseño e implementación de PMCTrackApp
6. Realización de los casos de prueba

Dichos puntos no están ordenados de forma estrictamente cronológica, puesto que bastante del trabajo contenido en ellos se realizó de forma intermitente o solapada con el trabajo de otros puntos.

D.2.1. Estudio de documentación

El estudio de documentación estuvo presente a lo largo de todo el proyecto; puesto que éste constó del trabajo a muy distintos niveles y con diferentes tecnologías.

En un primer momento, la documentación que estudié fue la relativa a familiarizarme con la monitorización mediante contadores hardware o PMCs y con la herramienta PMCTrack. Comencé con la lectura de la memoria del proyecto precedente al nuestro [6], del año 2015, y el manual de la página oficial PMCTrack. A continuación, leí la documentación de la Odroid Xu4, nuestro laboratorio experimental, sobre cómo se instala un kernel en dicha placa, en que zona de memoria está escrito el kernel, cómo conectarse a esa placa y cómo instalarle una aplicación. Adicionalmente leí la documentación del kernel de Android para entender la arquitectura de Android y tipos de componentes de una aplicación Android (Activity, Service, Broadcast receiver, Content provider). Finalmente, forma parte de esta primera fase de documentación, la lectura de los diversos manuales para entender el proceso de la compilación cruzada.

En lo que podríamos llamar una segunda fase del estudio de documentación, me dediqué al aprendizaje del uso java en el Android Studio. Para ello, me leí la documentación de Android Studio, informarme sobre el uso de SDK de Android para instalar la aplicación desde el *host* de desarollo y me apunté a un foro de los desarrolladores en esa plataforma. Esta fase se corresponde con la implementación de los objetos de procesamiento para PMCTrackApp.

Por último, la última fase de mi estudio de documentación se corresponde con las contribuciones de más bajo nivel: "Modificación del código de pmctrack para poder monitorizar aplicaciones por PID con la opción *-p*". En esta fase, la documentación fue sobre todo alrededor de la lectura del código C que existía previamente en la herramienta PMCTrack, incluyendo los relativos al kernel Linux modificado, a los módulos para el kernel y a la herramienta de línea de comandos `pmctrack`.

D.2.2. Uso de la placa

He tenido que familiarizarme con el uso de la placa e instalar un kernel original de Android. Tras la instalación, averiguar en qué zona se han escrito los binarios del kernel para que cuándo tengamos el kernel modificado sepamos donde hay que instalarlo. Informarnos de cómo podemos pasar los archivos entre la placa y nuestro *host* de desarrollo, tanto mediante el cable USB usando `minicom` como mediante la red usando *ADB*. Por último, averiguar la instalación de cualquier aplicación Android mediante apk de aplicaciones o usando SDK Android para instalar PMCTrack desde Android Studio.

D.2.3. Modificación de los archivos fuente de pmctrack

He realizado cambios en los códigos fuente mencionados en el Capítulo 2 para poder añadir la opción `-p` con el objetivo de poder monitorizar una aplicación en ejecución, introduciendo su número de `PID`, asimismo monitorizar los distintos hilos de esa aplicación que se vayan creando con el tiempo.

D.2.4. Compilación e instalación del kernel

Para realizar esta tarea, comencé leyendo los distintos archivos que van a ser modificados para que funcione pmctrack, luego ver cómo se compila el kernel con el mismo ID. Para conseguirlo he tenido que seguir el hilo de la compilación paso a paso para poder superar el problema de *vermagic* mencionado en el Capítulo 2. Finalmente he tenido que buscar la manera de instalar el kernel modificado encima del antiguo sin alterar el funcionamiento de la placa y que el nuevo kernel pueda terminar de ejecutarse sin problema, para más información accede al Capítulo 2.

D.2.5. Diseño e implementación de PMCTrackApp

Primero he empezado con la creación de distintos Mockups de la interfaz principal y de la interacción con el usuario, analizarlos para poder optar por un diseño inicial para PMCTRackApp.

Segundo he tenido que adaptarme con la forma de desarrollo de aplicaciones. Hemos intentado leer de distintas ubicaciones donde nos hemos encontrado con el problema de los permisos, entonces he tenido que encontrar la forma de crear túneles entre bash y la aplicación para poder hacer todo lo que necesita permisos de root desde líneas de código, como se había explicado en el Capítulo 3. Asimismo el lanzamiento del comando *pmctrack* desde líneas de código.

Tercero, una vez hemos superado el problema de los permisos llegó el tiempo de crear las distintas estructuras necesaria guardando toda la información necesaria para poder mostrar las métricas en tiempo real.

Cuarto he tenido que ver tutoriales sobre la conexión de la placa con *Android Studio* y cómo usar el *SDK* para poder instalar la aplicación mediante *ADB*.

Quinto he dado permisos a la PMCTrackApp para que pueda escribir y leer en el espacio Java de Android y cambiar algunos aspectos como poner el botón *CHANGE* mencionado en el Capítulo 3.

Sexto he tenido que usar una librería de java, *Interpreter*, para poder parsear las métricas facilitadas por mi compañero a un valor numérico y añadirlas a las estructuras correspondientes.

Finalmente he adaptado la librería de Java *mChart* para que pueda funcionar creando un id de xml para poder integrarlo en la interfaz principal, pudiendo mostrar las métricas en el grafo.

D.2.6. Realización de los casos de prueba

He tenido que buscar distintos *benchmarks* potentes para disponer de suficiente material, garantizando un uso benéfico de los casos de pruebas. Tras buscar en varias páginas web he encontrado varios y he elegido los más importantes de distintas características de Android. Una vez elegidos los *benchmarks* he tenido que buscar cómo instalarlos en la placa y analizarlos mediante sesiones de monitorización. Después de hacer la monitorización nos hemos encontrado con archivos super largos (algunos con más de 40.000 líneas). De este modo para poder analizarlos hemos buscado algunas herramientas para poder calcular la media de esos datos y mostrarlas en forma de grafos. Con el objetivo de analizar los datos facilitados por PMCTrackApp, hemos estado leyendo la documentación de *Pandoc*, una herramienta de Python para el análisis de datos, pero al final hemos usado *XLSTAT* un complemento de *Excel*. Para poder mostrar las medias en histogramas hemos usado *Excel*.

Bibliografía

[1] W. Cohen. Tuning programs with oprofile. *Wide Open Magazine*, 1:53–62, 2004.

[2] computerhoy. More information about android benchmark. `http://computerhoy.com/listas/apps/mejores-benchmark-apps-android-5080`, 2016.

[3] K. T. Ghiasi, S. and F. Rawson. Ascheduling for heterogeneous processors in server systems. In *Proc. of Computing Frontiers 05, Como, Italy*, 2005.

[4] B. M. ingh, K. and S. A. McKee. Real time power estimation and thread scheduling via performance counters. In *IGARCH Comput. Archit.News, 37, 46-55.*, 2009.

[5] S. Jarp, R. Jurga, and A. Nowak. Perfmon2: a leap forward in performance monitoring. *Journal of Physics: Conference Series*, 119, 042017, 2008.

[6] J. C. H. A. S. Juste. Interfaz de uso de contadores hardware multiarquitectura, 2015.

[7] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, May 2008.

[8] D. Koufaty, D. Reddy, and S. Hahn. Bias Scheduling in Heterogeneous Multi-core Architectures. In *Proc. of Eurosys '10*, pages 125–138, 2010.

[9] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proc. of EuroSys*, pages 153–166, 2010.

[10] odroid. More information about memory partition odroid xu4 board. `http://odroid.com/dokuwiki/doku.php?id=en:xu3_partition_table`, 2016.

[11] Perf. Perf wiki tutorial on perf. `https://perf.wiki.kernel.org/index.php`, 2015. Accessed: 2015-01-20.

[12] V. Petrucci, O. Loques, D. Mossé, R. Melhem, N. A. Gazala, and S. Gobriel. Energy-efficient thread assignment optimization for heterogeneous multicore systems. *ACM Trans. Embed. Comput. Syst.*, 14(1):15:1–15:26, Jan. 2015.

[13] PMCTrack. project official website. `http://pmctrack.dacya.ucm.es/`, 2014.

[14] PMCTrack. pmctrack_android-kernel-odroidxu3-3.10.y_arm.patch. `https://github.com/jcsaezal/pmctrack/blob/master/src/kernel-patches/pmctrack_android-kernel-odroidxu3-3.10.y_arm.patch`, 2016.

[15] A. K. D. a. P. M. Saez, Juan Carlos Fedorova. Leveraging core specialization via os scheduling to improve performance on asymmetric multicore systems. In *ACM TOCS, 30, 6:1-6:38*, 2012.

[16] J. C. Saez, J. Casas, A. Serrano, R. Rodríguez-Rodríguez, F. Castro, D. Chaver, and M. Prieto-Matias. An OS-oriented performance monitoring tool for multicore systems. In *Proc. of Euro-Par 2015: Parallel Processing Workshops*, pages 697–709, Cham, 2015. Springer International Publishing.

[17] J. C. Saez, A. Pousa, F. Castro, D. Chaver, and M. Prieto-Matias. ACFS: A completely fair scheduler for asymmetric single-isa multicore systems. In *Proc. of the 30th ACM Symposium on Applied Computing (SAC'15)*, 2025.

[18] J. C. Saez, A. Pousa, R. Rodriguez, F. Castro, and M. Prieto-Matias. Delivering performance counter monitoring support to the OS scheduler. *Computer Journal (accepted with minor revisions)*, 2016.

[19] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A Comprehensive Scheduler for Asymmetric Multicore Systems. In *Proc. of Eurosys '10*, pages 139–152, 2010.

[20] K. S. Spiliopoulos, V. and G. Keramidas. Green governors: A framework for continuously adaptive dvfs. In *IEEE Computer Society Press, Los Alamitos, CA.*, 2011.

[21] V. Weaver. Linux perfevents features and overhead. In *Proc. of International Workshop on Performance Analysis of Workload Optimized Systems*, page 80, 2013.

[22] K. Yaghmour. *Embedded Android*. O'Reilly Media, Sebastopol, CA, 2013. Index.

[23] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Cache Contention in Multicore Processors Via Scheduling. In *Proc. of ASPLOS '10*, pages 129–142, 2010.

[24] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Akula: A toolset for experimenting and developing thread placement algorithms on multicore systems. In *Proc. of PACT '10*, pages 249–260, 2010.