Interfaz de uso de contadores hardware multiarquitectura

TRABAJO FIN DE GRADO

Jorge Casas Hernán
Abel Serrano Juste

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería del Software / Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Junio 2015

Interfaz de uso de contadores hardware multiarquitectura

Memoria de Trabajo Fin de Grado

Jorge Casas Hernán
Abel Serrano Juste

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería del Software / Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Junio 2015

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, Jorge Casas Hernán y Abel Serrano Juste, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: "Interfaz de uso de contadores hardware multiarquitectura ", realizado durante el curso académico 2014-2015 bajo la dirección de Juan Carlos Sáez Alcaide en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Jorge Casas Hernán

Abel Serrano Juste

Madrid, a 18 de junio de 2015.

"Insanity is doing the same thing,
over and over again,
but expecting different results."
– Albert Einstein

Agradecimientos

Queremos agradecer en primer lugar a nuestro director de proyecto, Juan Carlos Sáez Alcaide, por su infinita paciencia e inestimable ayuda. Nos sentimos afortunados de haber contado con el mejor director de proyecto que podríamos haber tenido. Ha estado siempre disponible para todas nuestras dudas, nos ha motivado durante el desarrollo del proyecto, nos ha enseñado muchísimo y, en definitiva, nos ha proporcionado todos los requisitos necesarios para llevar a cabo un proyecto de calidad.

Queremos agradecer también a nuestros familiares su fuente de apoyo constante e incondicional en toda nuestra vida y, más aún, en aquellos momentos difíciles donde más apoyo sentimental necesitábamos. Sin su ayuda llegar hasta aquí no habría sido posible.

Finalmente, queremos agradecer a nuestros amigos, quienes han celebrado nuestras alegrías, compartido nuestro dolor, y jamás nos han juzgado por nuestros errores.

A todas estas personas, muchas gracias de corazón.

Resumen

Nuestro proyecto ha consistido en la ampliación de la herramienta PMCTrack para el kernel Linux, cuyo fin es permitir la monitorización del rendimiento de un programa mediante el uso de los contadores hardware del procesador.

Esta ampliación ha supuesto la inclusión de tres nuevas características. La primera ha consistido en la modificación de PMCTrack para dar soporte a la monitorización de programas multihilo desde espacio de usuario. En segundo lugar se ha dotado a PMCTrack de una interfaz de programación para la monitorización del rendimiento en fragmentos de código específicos. Por último, se ha procedido al diseño e implementación de una Interfaz Gráfica de Usuario o GUI (*Graphical User Interface*), que simplifica la configuración de eventos hardware y permite visualizar gráficas de los datos obtenidos en tiempo real.

Para poner a prueba estas tres nuevas características y mostrar la utilidad de nuestras aportaciones, se han llevado a cabo diversos casos de estudio, los cuales los presentamos también dentro de este documento.

Palabras clave: Monitorización rendimiento, Monitorización hardware, Contadores hardware, Kernel Linux, Análisis código fuente, Aplicaciones multihilo, Monitorización de la memoria caché.

Abstract

Our project focused on augmenting the PMCTrack tool for the Linux kernel, whose purpose is to enable monitoring application performance via hardware monitoring counters. The enhancement process entailed the inclusion of three new features in PMCTrack. First, we augmented the tool with support for performance monitoring of multithreaded programs from user space. Second, a programming interface was built on top of PMCTrack's kernel module making it possible to monitor the performance of specific code fragments with hardware counters. Third, we designed and implemented PMCTrack-GUI, a graphical frontend for PMCTrack enabling real-time visualization of high-level performance metrics and specifically designed to simplify the configuration of hardware events to the end user.

To demonstrate the effectiveness of our contributions, we test the functionality of the various PMCTrack extensions carried out in this project by means of several case studies, we also include those studies in this document.

Keywords: Hardware Profiling, Profiling tools, Performance monitoring counters, Linux kernel, Source code analysis, Multithreaded applications, Cache Monitoring.

Índice

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. PMCTrack: gestión de los contadores hardware de monitorización del rendimiento

La mayor parte de los procesadores actuales cuentan con una serie de contadores hardware para monitorización o **PMCs** (*Performance Monitoring Counters*). Estos contadores permiten a los usuarios obtener métricas de rendimiento de sus aplicaciones, tales como el número de instrucciones por ciclo (IPC) o la tasa de fallos del último nivel de la caché (*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 esté ejecutando 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 [23, 10, 7].

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 [11, 24, 15, 25, 12, 20, 18, 19]. Las herramientas de dominio público que hacen uso de los PMCs permiten monitorizar el rendimiento de aplicaciones desde el espacio del usuario, pero no proporcionan una API independiente de la arquitectura para que el propio sistema operativo pueda utilizar la información de los PMCs para llevar a cabo 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 [11, 12, 20, 19]. Sin embargo, esta aproximación deja "atada" la implementación del planificador a una cierta arquitectura o modelo del procesador, y adicionalmente, obliga a los desarrolladores a tratar con las rutinas de bajo nivel que acceden a los PMCs en cada arquitectura soportada

por el planificador. Para evitar enfrentarse a estos graves problemas, otros investigadores han recurrido al desarrollo de sencillos prototipos ejecutados en el espacio de usuario [24, 25, 18]. Estos prototipos dependen de herramientas de PMCs existentes orientadas a ser utilizadas en el espacio de usuario.

Para superar estas limitaciones se propuso el desarrollo de PMCTrack, una herramienta de gestión de contadores hardware para el kernel Linux, pero diseñada principalmente para que el SO usara los contadores para llevar a cabo tareas internas, como la planificación de procesos. Esta herramienta fue desarrollada inicialmente en un proyecto de Sistemas Informáticos por estudiantes de esta misma facultad, en el año 2012 [14]. Actualmente, distintos miembros del Grupo de investigación en Arquitectura y Tecnología de Sistemas de Computación (ArTeCS) de esta universidad, se ocupan del desarrollo y mantenimiento de PMCTrack. De hecho, de forma simultánea a la realización de nuestro Trabajo de Fin de Grado se han añadido nuevas funcionalidades a esta herramienta más allá de las extensiones propuestas en nuestro TFG.

La novedad de la herramienta PMCTrack está ligada a la abstracción del *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. En concreto, para hacer funcionar el planificador en una nueva arquitectura o modelo de procesador basta con desarrollar el módulo de monitorización correspondiente para esa nueva arquitectura o modelo de procesador en un módulo cargable del kernel. Además, PMCTrack ofrece una interfaz independiente de la arquitectura para configurar fácilmente los eventos y recopilar datos de los PMCs. Gracias a esto, el desarrollador del módulo de monitorización no tiene que lidiar con el código de bajo nivel específico de la arquitectura para acceder a los PMCs, lo que simplifica enormemente la implementación.

A pesar de ser una herramienta diseñada específicamente 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, 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 nivel de ocupación de una caché compartida, se puede exponer facilmente al usuario mediante la abstracción de *contadores virtuales* que ofrece PMCTrack.

1.2. Alternativas a PMCTrack

Se han creado varias herramientas para el kernel Linux en los últimos años [7, 10, 17, 5, 22, 21], ocultando la gran diversidad existente de interfaces hardware a los usuarios finales y proporcionando a estos un acceso cómodo a los PMCs en el espacio de usuario. En general, estas herramientas se pueden dividir en dos grandes grupos. El primer grupo incluye herramientas como Oprofile [7], perfmon2 [10], o perf [17], los cuales exponen al usuario los contadores de monitorización a través de un conjunto reducido de herramientas de línea de comandos. Estas herramientas no requieren modificar el código fuente de la aplicación que se desea monitorizar, sino que actúan como procesos externos con la capacidad de recibir datos de los PMCs de otra aplicación. El segundo grupo de herramientas provee al usuario librerías para acceder a los contadores desde el código fuente de la aplicación, lo que constituye una potente interfaz de acceso a los PMCs. Las librerías libpfm [10] y PAPI [5] siguen este enfoque.

La herramienta perf [17], que se basa en el subsistema de *Eventos Perf* [23] del kernel Linux, es posiblemente la herramienta más completa del primer grupo comentado en la actualidad. Aunque perf comenzó como una herramienta de uso de PMCs que soportaba un amplio abanico de arquitecturas, ahora dota a los usuarios de potentes capacidades de monitorización permitiéndoles hacer un seguimiento de las llamadas al sistema de un proceso o de las actividades relacionadas con el planificador. Además, al igual que PMCTrack, perf también tiene la capacidad de exponer al usuario otra información de monitorización hardware presente en procesadores modernos (pero no proporcionada por los PMCs), como por ejemplo la tasa de ocupación de la memoria caché.

A pesar del potencial de perf y de las otras herramientas relacionadas, ninguna de ellas implementa un mecanismo que proporcione a nivel del kernel una interfaz independiente de la arquitectura que permita al planificador del SO aprovechar la información de los PMCs para sus decisiones internas. Este es el principal propósito de PMCTrack.

Al igual que PMCTrack, algunas herramientas de PMCs requieren la realización de modificaciones en el kernel Linux para que puedan funcionar correctamente [10, 1]. KerMon [1] se basa en una clase de planificación separada en el kernel para llevar a cabo el acceso a bajo nivel a los PMCs. Para extraer los datos de los PMCs de una aplicación a través de KerMon, la aplicación debe ser planificada con la nueva clase de planificación. Esta clase de planificación realmente no es más que un clon de la clase por defecto de Linux (CFS - *Completely Fair Scheduler*), por lo que no explota la información de los PMCs para tomar decisiones. PMCTrack, por el contrario, hace posible que prácticamente cualquier clase de planificación creada en el kernel pueda obtener métricas de rendimiento a través de un mecanismo independiente de la arquitectura. Para ello, es necesario realizar solo pequeñas modificaciones en el kernel, ya que como se muestra en la siguiente sección, la mayor parte de la funcionalidad de PMCTrack se encapsula en un módulo del kernel.

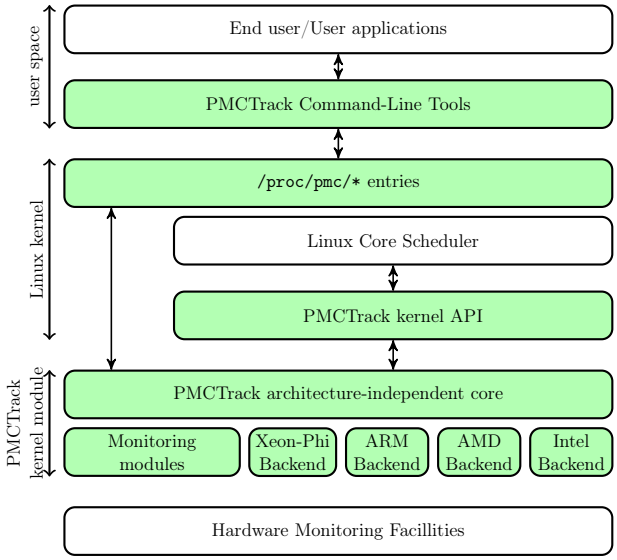


Figura 1.1: Arquitectura de PMCTrack

1.3. Diseño de PMCTrack

Esta sección describe la arquitectura interna de PMCTrack tal y como era antes de iniciarse nuestro TFG, así como los distintos modos de uso que soportaba.

1.3.1. Arquitectura

La figura A.1 representa la arquitectura interna de PMCTrack antes de comenzar nuestro desarrollo del TFG. La herramienta consta de un conjunto de componentes en el espacio de usuario y del kernel. Esencialmente, el usuario final interactúa con PMCTrack a través de las herramientas de línea de comandos disponibles. Estos componentes se comunican con el módulo del kernel de PMCTrack por medio de un conjunto de entradas del sistema de ficheros `/proc` exportadas por el módulo.

El módulo del kernel implementa la mayor parte de la funcionalidad de PMCTrack. Para recopilar los datos de los contadores de rendimiento de cada hilo es necesario que el módulo sea plenamente consciente de los eventos de planificación que suceden en todo momento, como por ejemplo los cambios de contexto. Además de exponer los datos de los PMCs de las aplicaciones a las herramientas de modo usuario, el módulo implementa una API sencilla para proporcionar datos de los PMCs a cualquier clase de planificación que requiera esa información para su correcto funcionamiento. Debido a que tanto el núcleo del planificador de Linux como las clases de planificación se implementan en su totalidad en el kernel, para que el módulo de PMCTrack del kernel pueda ser consciente

de estos eventos y solicitudes es imprescindible la realización de pequeñas modificaciones en el propio kernel Linux. Estas modificaciones del kernel, representadas en la figura A.1 con el nombre de "PMCTrack kernel API", se encargan de enviar un conjunto de notificaciones al módulo desde el núcleo del planificador.

Para poder recibir estas notificaciones, el módulo PMCTrack del kernel implementa la siguiente interfaz:

```
typedef struct pmc_ops{
  /* invoked when a new thread is created */
  void* (*pmcs_alloc_per_thread_data)(unsigned long,struct task_struct*);
  /* invoked when thread leaves the CPU */
  void (*pmcs_save_callback)(void*, int);
  /* invoked when thread enters the CPU */
  void (*pmcs_restore_callback)(void*, int);
  /* invoked every clock tick on a per-thread basis */
  void (*pmcs_tbs_tick)(void*, int);
  /* invoked when a process invokes exec() */
  void (*pmcs_exec_thread)(struct task_struct*);
  /* invoked when a thread exists the system */
  void (*pmcs_exit_thread)(struct task_struct*);
  /* invoked when a thread's descriptor is freed up */
  void(*pmcs_free_per_thread_data)(struct task_struct*);
  /* invoked when the scheduler requests per-thread
        monitoring information  */
  int  (*pmcs_get_current_metric_value)(struct task_struct* task, int key
      , uint64_t* value);
} pmc_ops_t;
```

La mayoría de estas notificaciones son enviadas únicamente cuando el módulo PMCTrack del kernel está cargado y el usuario (o el propio planificador) está usando la herramienta para monitorizar el rendimiento de una determinada aplicación.

Tal y como se ilustra en la figura A.1, el módulo PMCTrack del kernel consta de varios componentes. La capa del núcleo de PMCTrack independiente de la arquitectura implementa un interfaz llamado `pmc_ops_t` e interactúa con la herramienta PMCTrack de línea de comandos a través del sistema de ficheros */proc* de Linux. El componente independiente de la arquitectura se basa en 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. Además, de forma simultánea al desarrollo de nuestro TFG, se llevó a cabo el desarrollo de dos *backends* extras, uno compatible con procesadores Cortex de ARM de 32 y 64 bits, y otro para el coprocesador Xeon Phi de Intel. El módulo del kernel de PMCTrack también incluye un conjunto de *módulos de monitorización* específicos de la plataforma. El objetivo principal de un módulo de monitorización es proporcionar a un algoritmo de planificación implementado en el kernel aquellas métricas de rendimiento de los hilos necesarias para que éste pueda funcionar.

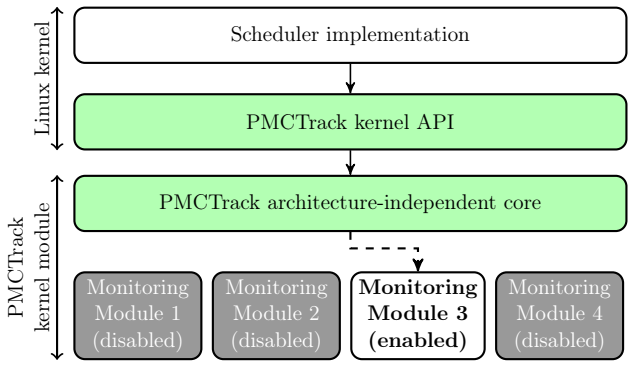


Figura 1.2: Módulos de monitorización de PMCTrack

Dotar al kernel Linux de soporte para PMCTrack implica incluir dos nuevos ficheros en el propio kernel y añadir menos de 20 líneas de código a las fuentes del núcleo. Estos cambios pueden ser aplicados fácilmente a diferentes versiones del kernel a partir de la versión 2.6.38.

1.3.2. Modos de uso de PMCTrack

Antes de iniciar el desarrollo de este TFG, PMCTrack soportaba tres modos de uso: modo planificador, muestreo por tiempo y muestreo basado en eventos.

1.3.2.1. Modo planificador

Este modo permite que cualquier algoritmo de planificación en el kernel (es decir, clase de planificación) pueda obtener datos de monitorización de cada hilo, haciendo posible la toma de decisiones en función de estos datos. La activación de este modo en un determinado hilo desde el código del planificador se reduce a la activación del flag `prof_enabled`[1] en el descriptor del hilo.

Para garantizar que la implementación del algoritmo de planificación que explota este modo de uso se mantiene independiente de la arquitectura, el propio planificador (implementado en el kernel) 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 de alto nivel necesarias para el algoritmo, como por ejemplo el número de instrucciones retiradas por ciclo (IPC) o la tasa de fallos de caché.

Como puede verse en la figura A.2, PMCTrack puede incluir varios *módulos de monitorización* 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 disponibles varios

[1]Este flag se añade a la estructura `task_struct` de Linux al aplicar el parche del kernel para PMCTrack.

módulos de monitorización, el administrador del sistema puede indicar al sistema cual de ellos usar, escribiéndolo en el fichero `/proc/pmc/mmon_manager`. De manera similar, el período de muestreo de datos de los PMCs utilizado en el módulo de monitorización se puede configurar a través del sistema de ficheros */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 de PMCTrack del kernel:

```
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 actualizados, la función mencionada podrá ser invocada desde la función de tratamiento de señal 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 modelos de procesador que aparezcan en un futuro. Todo lo que hay que hacer es 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 muy similar a `pmc_ops_t`. En concreto, se compone de varias llamadas a funciones que permiten notificar al módulo sobre activaciones y desactivaciones solicitadas por el administrador del sistema, cambios de contexto de hilos, salidas y entradas de un hilo del sistema (*sleep/resume*), solicitudes de valores de métricas de PMCs por parte del planificador, etcétera. Sin embargo, el programador normalmente solo implementa el subconjunto de llamadas a funciones requeridas para llevar a cabo el proceso interno necesario.

La creación de nuevos módulos de monitorización es una tarea bastante sencilla por varias razones. En primer lugar, el programador no necesita acceder directamente a los registros de los PMCs. En lugar de ello, el módulo PMCTrack del kernel ofrece una API independiente de la arquitectura que permite al módulo de monitorización especificar la configuración del contador a través de *strings*, para recibir muestras de PMCs periódicamente y controlar la multiplexación de eventos. En segundo lugar, debido a que el módulo recibe notificaciones cuando se crea un nuevo hilo o cuando éste termina, el módulo de monitorización puede asignar datos referentes a un hilo concreto para simplificar cualquier tipo de procesamiento específico de hilo. Finalmente, como el código de un módulo de monitorización reside en el módulo del kernel de PMCTrack, subsanar un error de programación no precisa reiniciar el sistema en la mayoría de los casos. El módulo del kernel puede recompilarse, descargarse y volverse a cargar una vez se hayan arreglado los errores.

1.3.2.2. Muestreo por tiempo (TBS - *Time-Based Sampling*)

Esta característica permite al usuario recopilar datos de rendimiento de una aplicación secuencial desde espacio de usuario a intervalos de tiempo regulares. La herramienta `pmctrack` de línea de comandos, cuya interfaz de usuario está inspirada en el programa `cputrack` de Solaris, hace posible esta función. Para ilustrar el funcionamiento de la herramienta consideremos el siguiente ejemplo:

```
$ pmctrack -T 1 -c pmc0,pmc3=0x2e,umask3=0x41 ./mcf06
nsample  event          pmc0          pmc3
      1   tick    1961001132        110634
      2   tick    1247853112          8323
      3   tick    1230836405          3859
      4   tick    1358134323        409386
      5   tick    1280630906       1199270
      6   tick    1231578609      15488307
...
```

En un sistema con un procesador moderno de Intel, este comando proporcionará al usuario el número de instrucciones retiradas (columna pmc0) y los fallos de caché (columna pmc3) en cada segundo. Cada muestra está representada por una fila diferente. El período de muestreo se puede especificar en segundos a través de la opción -T, siendo posible especificar también fracciones de segundo (por ejemplo 0.3 para 300ms). Al final de la línea se especifica el comando para ejecutar la aplicación asociada que se desea monitorizar (por ejemplo ./mcf06).

La opción -c acepta como argumento un *string* de configuración que sigue el formato de configuración de eventos internos reconocido por el módulo PMCTrack del kernel. El formato de esta línea da flexibilidad a usuarios experimentados permitiéndoles decidir los eventos que contarán cada uno de los contadores, especificando el código hexadecimal que será escrito en los registros de bajo nivel de los PMCs expuestos por el módulo del kernel. Como hemos visto, el string `pmc0,pmc3=0x2e,umask3=0x41` permite obtener el recuento de eventos mencionados en la mayoría de los procesadores modernos de Intel. En procesadores de la familia ARM Cortex Ax este conjunto de eventos puede representarse mediante el *string* `pmc1=0x8,pmc2=0x17`. Si el usuario desconoce los códigos hexadecimales que permiten asociar un evento a un contador es necesario consultar esos códigos en el manual de la arquitectura en cuestión.

Una característica muy destacable del programa `pmctrack` es su capacidad de obtener también los valores de los contadores virtuales que exporta el módulo de monitorización activo. De este modo, es posible contabilizar eventos con los contadores hardware al mismo tiempo que se extrae otro tipo de información de monitorización relevante, como el consumo de potencia media o energía consumida en un intervalo de tiempo prefijado. Para obtener los valores de los contadores virtuales es preciso usar la opción -V seguida de la cadena de caracteres que indica qué contadores virtuales (numerados a partir del 0) se desean consultar. Por ejemplo, la cadena `virt0,virt2` constituiría una cadena válida para consultar el valor de dos contadores virtuales siempre y cuando el módulo de monitorización activo exportase dichos contadores. Para consultar la semántica de los contadores virtuales actualmente exportados es preciso consultar una entrada */proc* gestionada por PMCTrack.

En caso de que un modelo específico del procesador no integre suficientes PMCs como para monitorizar a la vez un conjunto determinado de eventos, el usuario puede activar la función de multiplexación de eventos de PMCTrack. Esto se reduce a especificar varios conjuntos de eventos mediante la inclusión de varias instancias de la opción -c en la línea de comandos. En este caso, en la salida aparecerá un nuevo campo *expid* que indicará al usuario a qué experimento corresponde cada una de las muestras que se imprimen por pantalla.

Cabe destacar que en el caso de que el módulo de monitorización esté utilizando actualmente los PMCs en nombre de un algoritmo de planificación, al usuario no se le permite especificar un *string* de configuración a través de la opción -c de PMCTrack. Sin embargo, para propósitos de depuración el comando `pmctrack` puede seguir utilizándose (sin la opción -c) para obtener los recuentos de los eventos asociados con la configuración impuesta por el módulo de monitorización.

Para soportar la característica del muestreo por tiempo, el módulo del kernel de PMCTrack almacena las muestras de los contadores en un buffer circular. La herramienta de línea de comandos obtiene las muestras del buffer circular del kernel mediante la lectura de un fichero */proc* que bloquea el proceso de monitorización hasta que se generan nuevas muestras o hasta que termina la aplicación.

1.3.2.3. Muestreo basado en eventos (EBS - *Event-Based Sampling*)

El muestreo basado en eventos constituye una variante del muestreo basado en tiempo en el que los valores de los PMCs se recogen cuando el número de ocurrencias de un cierto evento alcanza un cierto umbral U. Para soportar EBS, el módulo PMCTrack del kernel explota la característica de interrupción por desbordamiento presente en la mayoría de las Unidades de Monitorización del Rendimiento (PMU) de los procesadores actuales. Esencialmente, cuando se activa EBS, el módulo del kernel de PMCTrack inicializa el contador a $-U$; cuando este contador se desborda la PMU genera una interrupción, y entonces el módulo del kernel lee todos los *PMCs*.

Para usar EBS desde el espacio de usuario se debe especificar el flag *ebs*, junto al número de contador asociado, al final del *string* de configuración de eventos pasado al comando `pmctrack`. Al hacer esto, también es posible especificar un valor del umbral como en el ejemplo siguiente:

```
$ pmctrack -c pmc0,pmc3=0x2e,umask3=0x41,ebs0=500000000 ./mcf06
nsample  event          pmc0          pmc3
      1    ebs     500000087         10677
      2    ebs     500000002         22336
      3    ebs     500000004         17131
      4    ebs     500000007         12995
      5    ebs     500000014          9348
      6    ebs     500000010          5804
...
```

La columna *pmc3* muestra el número de fallos de caché por cada 500 millones de instrucciones retiradas. Hay que tener en cuenta, sin embargo, que los valores de la columna *pmc0* no reflejan con exactitud el número especificado en el flag *ebs*. Esto tiene que ver con el hecho de que, en los procesadores modernos, la interrupción del PMU no se sirve inmediatamente después del desbordamiento del contador. Por el contrario, debido a la ejecución especulativa y fuera de orden, el procesador podría llegar a ejecutar cientos de instrucciones antes de procesar la interrupción. Estas imprecisiones no suponen un gran problema, siempre y cuando se establezcan tamaños de umbral suficientemente altos.

1.4. Objetivos del proyecto

La herramienta PMCTrack ofrece grandes funcionalidades y permite que el planificador del SO pueda explotar los contadores hardware para realizar optimizaciones en tiempo de ejecución. Sin embargo, PMCTrack aún cuenta con importantes limitaciones que ensombrecen su gran potencial, sobre todo en lo que respecta a la monitorización de aplicaciones desde modo usuario.

En primer lugar, su uso resulta un tanto complicado para un usuario poco experimentado, ya que éste tiene que consultar manuales técnicos de distintas arquitecturas para poder especificar los códigos hexadecimales de los eventos que desea contabilizar en cada uno de los PMCs.

En segundo lugar, la obtención del recuento de eventos específicos en los PMCs cada cierto tiempo no proporciona al usuario una visión global acerca de la evolución temporal del valor de esos contadores, y mucho menos de métricas de alto nivel compuestas por la combinación de valores de dos o más PMCs. Esta valiosa información para el usuario puede conseguirse mediante la construcción de gráficas. No obstante, esto requiere (1) procesar los datos proporcionados por el comando `pmctrack` para obtener los valores de las métricas de alto nivel a lo largo del tiempo y (2) emplear alguna utilidad como *Gnuplot* para la generación de las gráficas finales. Finalmente, otra limitación significativa de la herramienta es el hecho de que el comando `pmctrack` (y el propio módulo del kernel en el que se basa) no permite la monitorización de aplicaciones multihilo desde el espacio de usuario.

Para proporcionar una solución a estas y otras limitaciones de PMCTrack, este Trabajo de Fin de Grado persigue los siguientes objetivos:

1. Proporcionar soporte para la monitorización de aplicaciones multihilo desde el espacio de usuario con PMCTrack.

2. Diseñar e implementar un *frontend* gráfico para PMCTrack (llamado PMCTrack-GUI) que permita visualizar en tiempo real el valor de distintas métricas de rendimiento definidas por el usuario.

3. Crear *libpmctrack*, una librería que permita la monitorización de fragmentos de código de aplicaciones en el espacio de usuario mediante PMCs.

1.5. Plan de trabajo

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

1. Planificación del trabajo a realizar por cada uno de los dos integrantes del proyecto.
2. Lectura de documentación acerca de los lenguajes de programación y demás tecnologías utilizadas para llevar a cabo los desarrollos (Python, librería WX, matplotlib...). Esta etapa también conlleva familiarizarse con la arquitectura interna de PMCTrack.
3. Realización de bocetos o *mockups* de PMCTrack-GUI para estudiar distintas alternativas de diseño y ayudar a llegar a un consenso sobre el diseño definitivo.
4. Implementación del soporte para la monitorización de aplicaciones multihilo en el espacio de usuario con PMCTrack.
5. Implementación de PMCTrack-GUI.
6. Diseño e implementación de *libpmctrack* y refactorización del código del programa de línea de comandos `pmctrack`.
7. Realización de casos de estudio poniendo a prueba las herramientas desarrolladas.

Cabe destacar que el orden de estas etapas es meramente orientativo, ya que dichas etapas no se realizaron de forma estrictamente secuencial. En particular, fue necesario realizar planificaciones particulares en cada una de las etapas y distintos componentes de las aplicación PMCTrack-GUI se realizaron de forma simultánea. Adicionalmente, se dio soporte en nuestros desarrollos a funcionalidades de PMCTrack añadidas mientras llevábamos a cabo nuestro proyecto (PMCTrack es una herramienta en continuo desarrollo), y se añadieron nuevas funcionalidades a nuestros desarrollos que no estaban inicialmente previstas (como la inclusión de un modo SSH para PMCTrack-GUI).

1.6. Estructura de la memoria

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

- **El capítulo 2** explica el desarrollo del soporte a la herramienta PMCTrack para la monitorización de aplicaciones multihilo desde el espacio de usuario.
- **El capítulo 3** presenta la librería libpmctrack. En este capítulo no solo se describen los detalles de la librería sino también se presenta la motivación existente tras la misma.
- **El capítulo 4** presenta la motivación, el diseño e implementación de PMCTrack-GUI, el *frontend* gráfico de PMCTrack.
- **El capítulo 5** pone a prueba las extensiones de PMCTrack desarrolladas en este proyecto mediante tres casos de estudio.
- **El capítulo 6** expone las conclusiones finales de este Trabajo de Fin de Grado y presenta el trabajo futuro.

- Finalmente, se proporcionan varios apéndices. En ellos se incluye: (1) Introducción y conclusiones del proyecto traducidos al inglés, (2) bocetos iniciales del aspecto visual de PMCTrack-GUI, (3) diagramas de diseño UML con detalles concretos del diseño de PMCTrack-GUI, (4) pequeño manual de instalación de las dependencias software de PMCTrack-GUI en Debian/Ubuntu y MacOS X, y (5) contribuciones de cada participante al proyecto.

Capítulo 2

Soporte para monitorización de aplicaciones multihilo

El programa de línea de comandos `pmctrack` carece soporte de monitorización de aplicaciones multihilo desde espacio de usuario. Lamentablemente, esta limitación no se puede subsanar modificando únicamente dicho programa de usuario. Por el contrario, es el módulo del kernel PMCTrack el que no brinda la posibilidad de exportar datos de los contadores a las herramientas de modo usuario cuando la aplicación consta de varios hilos. Por lo tanto, para dotar a PMCTrack de este soporte es necesario realizar un rediseño profundo del módulo del kernel.

Este capítulo se estructura como sigue. La sección 2.1 presenta las limitaciones inherentes en el diseño original del módulo del kernel de PMCTrack que impedían el soporte para aplicaciones multihilo. La sección 2.2 describe el diseño alternativo y la implementación realizada para lograr dotar a PMCTrack del soporte deseado.

2.1. Diseño previo

En el kernel Linux, cada hilo del sistema está descrito internamente mediante una estructura `task_struct`. Esta estructura almacena campos críticos del hilo como su PID o el PID de su padre, los ficheros abiertos, un descriptor a las regiones de memoria usadas por el proceso al que pertenece, etcétera. Para ofrecer el soporte necesario, el módulo del kernel de PMCTrack también necesita mantener información privada de cada hilo que está siendo monitorizado, como por ejemplo la configuración de eventos hardware establecida por el usuario o el valor temporal de los contadores hardware cuando el hilo no está actualmente en ejecución. Para almacenar esta información, el parche del kernel para PMCtrack añade el campo `pmc`, un puntero a una estructura de tipo `pmon_prof_t` que almacena los datos privados del hilo usados por PMCTrack.

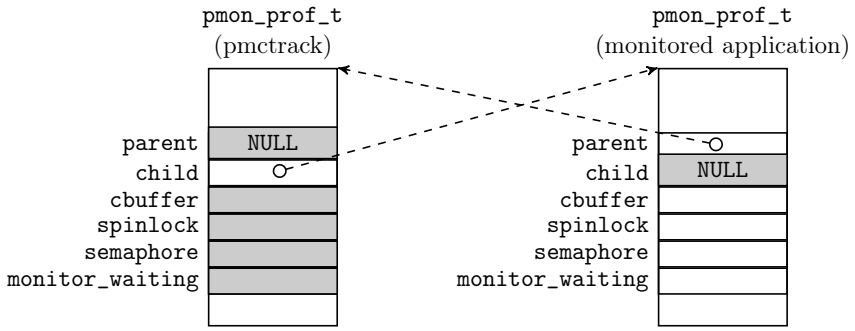


Figura 2.1: Relación entre las estructuras `pmon_prof_t` del proceso monitor y monitorizado en la implementación original del módulo de PMCTrack. Los campos sin usar en cada estructura se representan en gris.

Como se indicó en el capítulo previo, cuando una aplicación está siendo monitorizada con el programa `pmctrack`, el módulo del kernel de PMCTrack almacena los datos recabados con los contadores en un buffer circular acotado. El programa `pmctrack` consume los datos de los contadores leyendo de la entrada `/proc/pmc/monitor` que tiene semántica bloqueante. Al efectuar una lectura[1] de dicha entrada, el programa se queda bloqueado hasta que haya datos para consumir o la aplicación finalice.

Este escenario constituye claramente un caso particular del problema *Productor/-Consumidor*, con un productor –la aplicación secuencial que está siendo monitorizada– y un consumidor –el programa monitor `pmctrack`–. Nótese que para el kernel Linux el proceso monitor es padre del proceso monitorizado, ya que `pmctrack` emplea las llamadas `fork()` y `exec()` para ejecutar el comando pasado en la línea de comandos. Ambos procesos requieren acceder al buffer circular de muestras de los contadores y el acceso puede ser potencialmente concurrente. Nada impide que el kernel, en nombre de la aplicación monitorizada, desee insertar nuevas muestras en el buffer en el modo EBS o TBS, y esto ocurra al mismo tiempo que el programa `pmctrack` lee de la entrada */proc* para extraer elementos del buffer circular. Para garantizar exclusión mutua en el acceso al buffer circular, se empleaba un *spin lock* almacenado en la estructura `pmon_prof_t` del hilo que está siendo monitorizado. Adicionalmente, para dotar del carácter bloqueante necesario a la entrada */proc*, se empleaba un semáforo del kernel y un flag `monitor_waiting` que indica si el programa `pmctrack` está actualmente bloqueado a la espera de nuevas muestras. Ambos campos están también almacenados en la estructura `pmon_prof_t` del hilo monitorizado.

[1] Antes de poder leer datos de la entrada /proc, el programa pmctrack y la aplicación que está siendo monitorizada tienen que comunicar cierta información al módulo del kernel siguiendo el protocolo descrito en [14].

La figura 2.1 ilustra la relación entre las estructuras `pmon_prof_t` del proceso monitor `pmctrack` y del hilo de la aplicación secuencial que está siendo monitorizada. Como puede observarse en la figura, además de los campos mencionados previamente, la estructura `pmon_prof_t` poseía originalmente dos campos tipo puntero adicionales: `child` y `parent`. El campo `child` se empleaba para que el descriptor del proceso monitor, que ejecuta el programa `pmctrack`, almacene la referencia al `pmon_prof_t` del hilo de la aplicación secuencial que está siendo monitorizada. Gracias a este puntero el proceso monitor, al entrar en el kernel invocando la *read callback* de la entrada `/proc/pmc/monitor`, puede acceder tanto al buffer circular de muestras del hijo, como a los recursos de sincronización necesarios para acceder de forma segura al buffer. Por el contrario, el campo `parent` se emplea dentro del módulo de PMCTrack para que la aplicación secuencial sea consciente de que que está siendo monitorizada; en tal caso el puntero será distinto de NULL. Cuando la aplicación monitorizada sale del sistema, el módulo del kernel de PMCTrack debe encargarse de poner a NULL los punteros `child` y `parent` en la estructura `pmon_prof_t` del proceso monitor y del monitorizado, respectivamente. Esto no sería posible sin el puntero `parent` almacenado en el descriptor de la aplicación monitorizada.

El diseño anteriormente descrito presenta varias limitaciones importantes. La primera y más relevante es el hecho de que como tal este modelo no ofrece soporte para la monitorización de aplicaciones multihilo desde modo usuario. Esencialmente, como se ilustra en la figura 2.1, se establece una relación *uno a uno* entre el proceso monitor y el hilo monitorizado mediante los punteros `child` y `parent`, y no una relación *uno a varios* como requeriría el escenario con una aplicación multihilo. En este caso, todos los hilos de la aplicación monitorizada deberían compartir tanto el buffer de muestras como los recursos de sincronización para garantizar exclusión mutua y sincronizarse con el hilo monitor. El segundo aspecto negativo de este diseño es la presencia de numerosos campos no utilizados en las estructuras de `pmon_prof_t` de ambos procesos (campos que aparecen en gris). Finalmente, el proceso de actualización de punteros y liberación de la memoria reservada para el buffer de muestras resulta innecesariamente complejo, ya que en la implementación hay que actuar de forma diferente dependiendo del orden de finalización del proceso monitorizado y el monitor.

2.2. Nuevo diseño interno de PMCTrack

Para superar las limitaciones arriba mencionadas y dar soporte a PMCTrack para la monitorización de programas con más de un hilo de ejecución desde modo usuario, hemos optado por un cambio en el diseño inicial de la herramienta. Los cambios más significativos se realizaron en el módulo del kernel de PMCTrack.

Cuando se desea monitorizar una aplicación multihilo, el buffer de muestras debe estar compartido por todos los hilos de la aplicación monitorizada y por el proceso monitor. Como el acceso al buffer es de naturaleza concurrente, en este caso se manifiesta el problema *Productor/Consumidor* con n productores –hilos de la aplicación *multithread*– y un consumidor –proceso que ejecuta el programa `pmctrack`–. Existen múltiples diseños

alternativos, para que todos los hilos del sistema concurrente compartan tanto el buffer como los recursos de sincronización. No obstante, se ha intentado apostar por un diseño que reduzca al máximo el número de campos no utilizados dentro de la estructura `pmon_prof_t`.

Uno de los principales desafíos que surgió en el proceso de diseño es la gestión de la memoria dinámica asociada al buffer circular de muestras y a los recursos de sincronización. Si bien compartir un buffer entre dos procesos ya complicaba la implementación en la monitorización de aplicaciones secuenciales debido a la casuística en el orden de terminación de los procesos, la compartición con n procesos introduce complicaciones adicionales. En una aplicación multihilo, cada hilo puede finalizar en un orden distinto, y solo el hilo que termine en último lugar debería ocuparse de liberar la memoria del buffer. Adicionalmente, en este escenario se debe tener en cuenta que no se debe liberar la memoria del buffer hasta que el proceso monitor termine, ya que éste debe recoger del buffer todas las muestras de los contadores hardware hasta que cada hilo de la aplicación haya finalizado.

El diseño que ofrece la solución a este problema se basa en la estructura `pmc_samples_buffer_t` que se muestra a continuación:

```
typedef struct {
    cbuffer_t* pmc_samples;
    spinlock_t lock;
    struct semaphore sem_queue;
    volatile int monitor_waiting;
    atomic_t ref_counter;
}pmc_samples_buffer_t;
```

Esta estructura alberga múltiples campos que se encontraban dentro de la estructura `pmon_prof_t` en el diseño previo, como es el caso del buffer circular de muestras (`pmc_samples`) y los recursos de sincronización (`lock`, `sem_queue`, `monitor_waiting`). Adicionalmente, la nueva estructura consta de un contador de referencias para solucionar el problema de la gestión de memoria dinámica de forma más robusta, como se describe a continuación.

Como ilustra la figura 2.2 en el nuevo diseño la estructura `pmon_prof_t` almacena un puntero `sbuf` a una estructura de tipo `pmc_samples_buffer_t`. Esta modificación del diseño permite que múltiples procesos o hilos puedan compartir la estructura simplemente almacenando el mismo puntero en su campo de `sbuf` dentro de `pmon_prof_t`. El contador de referencias `ref_counter` de la estructura `pmc_samples_buffer_t`, permite llevar la cuenta del número de hilos que comparten la misma instancia de dicha estructura. Así por ejemplo, cuando se cree un nuevo hilo en una aplicación que esté siendo monitorizada desde modo usuario con PMCTrack, el campo `sbuf` dentro de la estructura `pmon_prof_t` del nuevo hilo apuntará a la misma dirección de memoria que el resto de hilos de la aplicación. Asimismo, al crear un nuevo hilo, el módulo del kernel incrementa en una unidad el contador de referencias para dejar constancia que que un hilo extra comparte la estructura `pmc_samples_buffer_t`. Análogamente, cuando un hilo finalice, el contador

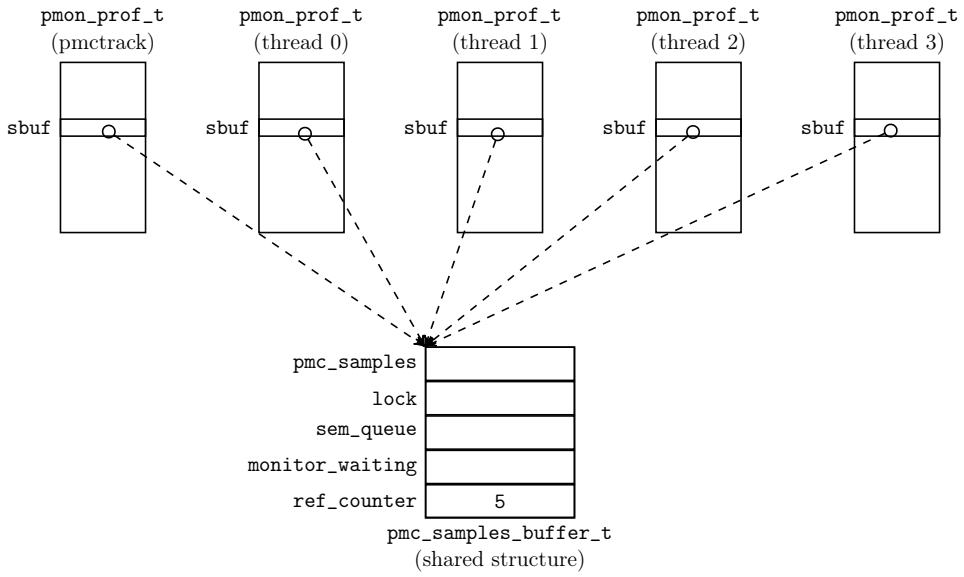


Figura 2.2: Relación entre la estructura `pmc_samples_buffer_t` y las estructuras `pmon_prof_t` del proceso monitor y de los hilos de una aplicación monitorizada (cuatro hilos) en la nueva implementación del módulo de PMCTrack.

de referencias se decrementa y el puntero `sbuf` se pone a `NULL`. Cuando el contador de referencias de la estructura alcance el valor cero (el último hilo que usa la estructura termina), el módulo del kernel de PMCTrack liberará la memoria de la estructura para que esté disponible de nuevo para el SO.

Para que el programa de usuario `pmctrack` pueda identificar las muestras que provienen de distintos hilos de ejecución de la aplicación, se incluyó un nuevo campo `pid` en la estructura que describe a cada muestra de los contadores (`pmc_sample_t`). Como en el kernel Linux, cada hilo tiene su propio identificador interno (campo `pid` de su `task_struct`) cada muestra que se inserta en el buffer está identificada por dicho campo identificativo. El código del programa `pmctrack` fue también modificado para garantizar que dicho identificador se muestra por pantalla junto con el valor recabado de los contadores hardware. Para ilustrar esta característica, el siguiente comando muestra distintos valores en la columna `pid` al monitorizar una aplicación paralela con cuatro hilos desde modo usuario empleando muestreo por tiempo:

```
$ pmctrack -T 1 -c pmc0,pmc1,pmc3=0x2e,umask3=0x41./rnaseq 4
nsample    pid      event          pmc0          pmc1          pmc3
      1  30120       tick    7194570808    3804503293        659126
      2  30127       tick    5414863136    2418082409        170203
      3  30128       tick    6519550211    3365103167        704584
      4  30126       tick    5797691183    2668461950        190487
      5  30120       tick    7417726822    3266118982        100831
      6  30127       tick    6391099361    2859571911         73155
      7  30128       tick    6202063856    2804748224         64599
```

```
 8  30126       tick    6747513672    2966250007        123480
 9  30120       tick    7186768494    3215750336        101673
10  30127       tick    5952872137    2755767164         74605
11  30128       tick    5704938508    2595268292        188362
12  30126       tick    6461952578    2958763944        102871
13  30120       tick    6466461413    3018968913        208613
...
```

Para concluir, es preciso destacar que el nuevo diseño del módulo del kernel de PMCTrack no solo permite la monitorización de aplicaciones paralelas con `pmctrack` sino que brinda la posibilidad de visualizar en tiempo real distintas métricas de rendimiento de distintos hilos con PMCTrack-GUI, *frontend* gráfico creado en este Trabajo de Fin De Grado y descrito en el capítulo 4. En el capítulo 5 se muestran ejemplos de monitorización de aplicaciones paralelas con PMCTrack-GUI. Por otra parte, la nueva implementación del módulo del kernel supuso el punto de partida para construir la librería *libpmctrack* que se describe en el siguiente capítulo. Al instrumentar el código de una aplicación paralela con funciones de esta librería, el módulo del kernel mantiene un buffer de muestras independiente por cada hilo de la aplicación. En este escenario de uso no hay proceso monitor externo, sino que cada hilo puede consumir las muestras de los contadores que ha generado.

Capítulo 3

libpmctrack

Una parte transcendental del proyecto ha sido la implementación de una librería llamada **libpmctrack**, que provee a los programadores de la funcionalidad de monitorización que brinda PMCTrack. De esta forma, un programador puede obtener información de cómo se comporta el hardware mientras se está ejecutando su programa haciendo simplemente llamadas a la API de libpmctrack, e incluso hacer uso de dicha información para otras funciones más allá de la pura monitorización.

Este capítulo consta de las siguientes secciones. La sección 3.1 explica las razones que motivaron la creación de libpmctrack. En la sección 3.2 hablamos del funcionamiento de la librería y comentamos sus potencialidades de uso. También en esta sección se describe cómo usamos libpmctrack para refactorizar todo el código de la herramienta de línea de comandos `pmctrack`. Por último, en la sección 3.3 exponemos y describimos la API de la librería, tanto su interfaz básica como su otra interfaz más avanzada.

3.1. Motivación

La herramienta de línea de comandos `pmctrack` hace una gran labor para monitorizar el comportamiento global de un programa dado. Sin embargo, carece de la posibilidad de permitir la monitorización de fragmentos de código específicos. Así pues, el usuario dispone de una herramienta para poder monitorizar programas, pero el programador no tiene acceso a una API programática para acceder a la funcionalidad de PMCTrack desde el código de su programa.

La aproximación más cercana que podría tomar un desarrollador consistía en ejecutar el propio comando `pmctrack` desde su programa y capturar su salida para procesarla. Este proceso, como veremos, no siempre constituye la opción más conveniente.

En primer lugar, `pmctrack` está pensado para usarse desde una consola interactiva directamente con el usuario. A la hora de monitorizar código, las necesidades son diferentes a las que podemos tener desde la consola. Por ejemplo, no queremos especificar la localización un programa externo a monitorizar, si no las líneas de código que nos interesan. Del mismo modo, tampoco queremos recibir la salida por pantalla, si no que nos interesará tenerla accesible desde dentro de nuestro propio programa para trabajar con ella.

En segundo lugar, la existencia de una librería permitiría la desencapsulación de las tareas de comunicación e intercambio de información con el kernel; de las tareas que se refieren puramente al programa que se quiere monitorizar. Generando código mucho más adaptable a diferentes contextos y más fácil de utilizar.

En tercer lugar, para poder utilizar `pmctrack` necesitaremos tenerlo compilado, instalado y accesible desde el `PATH`. Esto genera una dependencia y dificultades de instalación a la hora de usar la suite PMCTrack, que, en realidad, carece de sentido propio.

Por todo esto, con la creación de libpmctrack pretendemos solucionar este problema proporcionando al programador una herramienta que permita obtener información sobre los eventos hardware de los PMCs desde su propio código. Para ello, el programador solo tiene que hacer llamadas a la librería directamente desde su código. Libpmctrack se encarga de la comunicación con el kernel y de ocultar al programador las particularidades de la arquitectura interna de PMCTrack.

3.2. Descripción libpmctrack

Libpmctrack es una librería escrita en C, al igual que el resto de los componentes de PMCTrack, y que encapsula las funcionalidades de monitorización que tiene PMCTrack en una API de funciones disponibles al programador. De este modo, libpmctrack proporciona al programador acceso directo a la información obtenida desde los PMCs, sin tener que preocuparse de la implementación de dichos contadores en cada arquitectura. Además, libpmctrack permite el uso de los contadores virtuales proveídos por los módulos de PMCTrack para cada arquitectura, lo cual es una gran ventaja frente a otras herramientas con fines similares como PAPI-C [16].

La funcionalidad más evidente que ofrece esta librería es hacer observaciones de rendimiento, ya sea de un programa que hayamos escrito; o bien sobre una o varias plataformas hardware para un cierto código proporcionado.
Además, existen otros usos potenciales que se pueden explorar con esta librería. Dado que hay muchos programas que se ejecutan dentro del contexto de otros contenedores software, tales como máquinas virtuales o *runtime systems*, dichos contenedores se podrían beneficiar ampliamente de disponer de información en tiempo de ejecución sobre los hilos que están ejecutándose sobre ellos. Esto, al fin y al cabo, es una generalización de la mejora que aporta PMCTrack al planificador de Linux –como comentamos en el primer capítulo de Introducción–, ahora generalizada a cualquier programa que se ejecute en el contexto de otros componentes software. Este potencial es ahora realizable gracias a libpmctrack.

La implementación interna de la librería difiere ligeramente del usado para la herramienta de línea de comandos PMCTrack. En este caso, no se trata de un proceso padre que se encarga de lanzar y monitorizar un proceso hijo; si no de un proceso que, mediante llamadas a la librería libpmctrack, obtiene sus propios datos de monitorización de los PMCs y es el programador quién debe decidir qué hacer con estos datos de retorno. Este modo de uso es el que permite que los programas sean capaces incluso de tomar decisiones en tiempo de ejecución, tal y como contábamos antes, según la información de rendimiento proporcionada por el hardware.

Para usar libpmctrack, el programador tiene que incluir en su código un fichero de cabecera de la librería, configurar qué monitorizar y encerrar el código que quiera monitorizar con llamadas a las funciones `pmctrack_start_count()` y `pmctrack_stop_count()`. La configuración para la monitorización se hace pasando un *string* con la configuración para cada PMC o contador hardware y, opcionalmente, los contadores virtuales que se quieren utilizar.

La libpmctrack también soporta la monitorización de programas *multithread*. Para permitir que fragmentos de código independientes sean monitorizados de manera simultánea, el módulo del kernel de PMCTrack mantiene un buffer de muestras independiente para cada uno de los hilos. Esta propiedad es especialmente interesante en entornos de programación paralela. Por ejemplo, el *runtime* de OpenMP o de Cilk puede realizar optimizaciones en tiempo de ejecución para equilibrar mejor la carga entre los distintos hilos de procesamiento [6].

3.3. Refactorización de `pmctrack` usando libpmctrack

Durante el transcurso del proyecto, y con la librería casi acabada, nos dimos cuenta de que había mucho código repetido entre ésta y la herramienta de comandos `pmctrack`. Fue entonces cuando pensamos que el código del programa `pmctrack` podía simplificarse considerablemente si se reescribía haciendo uso de funciones definidas en libpmctrack. Por lo tanto, llevamos a cabo una profunda refactorización del código de `pmctrack` usando libpmctrack.

En la refactorización, nos fijamos el objetivo de conseguir que `pmctrack` no se tuviese que comunicar directamente con el kernel en ningún momento, y se limitase a interactuar con el usuario, a la creación de los procesos y estructuras para monitorizar, y las llamadas a las funciones de libpmctrack que hicieran el trabajo de bajo nivel.

Gracias a esta refactorización, hemos conseguido reducir el código de `pmctrack` sustancialmente: inicialmente el código era de casi 900 líneas de código; ahora el código ha quedado reducido a algo más de 500 líneas, lo cual constituye una disminución del 40 %. Asímismo, el código ha quedado más desacoplado de la parte interna de PMCTrack, mejorando así notablemente su claridad y sencillez.

La refactorización también influyó positivamente en la librería, puesto que nos sirvió para verificar su funcionamiento y depurarla. Además, añadimos algunas funciones nuevas a la librería, que resultaron ser más relevantes tener en ésta en lugar de directamente desde el código del programa que la usase.

3.4. API

La API de libpmctrack está separada en dos interfaces. La primera viene definida en el fichero `pmctrack.h` y en ella se proveen las funciones de alto nivel que se necesitan para un uso estándar de monitorización de un programa dentro del propio código. La segunda viene definida en el fichero `pmctrack_internal.h` y contiene otra serie de funciones para acceso de más bajo nivel a las funcionalidades que provee el kernel modificado para PMCTrack. Es esta segunda interfaz la que usa el programa de línea de comandos.

Pasamos a desarrollar cada una de ellas individualmente.

3.4.1. pmctrack.h

Como hemos dicho anteriormente, el fichero `pmctrack.h` contiene la declaración de la interfaz básica para monitorizar rendimiento con PMCTrack desde código fuente. Este fichero define también un descriptor para usar en el paso de opciones a las funciones, así como las funciones de la API básica de libpmctrack. Todas estas funciones comienzan con el prefijo `pmctrack_*`. El código de este fichero cabecera puede verse en la figura 3.1. A continuación, explicamos el uso de esta interfaz.

El struct `pmctrack_desc_t` es un descriptor de los parámetros de configuración del usuario para PMCTrack y sirve como estructura de comunicación entre el usuario y la librería. Este descriptor no se debe modificar directamente por el usuario, será libpmctrack quién depositará allí la configuración de monitorización a realizar a partir de los paramétros introducidos por el usuario. De este modo se consigue de una forma elegante, sencilla y transparente al usuario en el paso de parámetros y de valores entre funciones de libpmctrack. Los diversos campos, por tanto, corresponden a valores de configuración y de resultados de monitorización obtenidos por libpmctrack.

Para usar esta librería, el usuario lo primero que tiene que hacer es declarar un `struct pmctrack_desc_t` y hacer que libpmctrack lo inicialice llamando a `pmctrack_init()` junto con un número máximo de muestras para definir un tamaño máximo al buffer intermedio que los va a ir almacenando. Si se superase dicho tamaño no habría ningún error en ejecución del programa pero sí se perderían algunos datos.

A continuación, se deben definir los *strings* con la definición de los contadores que se quieren usar y los eventos que se quiere que se cuenten, así como con los contadores virtuales si se quisiese usar alguno. Estos *strings* se han de pasar como argumento a la función `pmctrack_config_counters()`.

Figura 3.1: Interfaz básica libpmctrack

```
typedef struct {
        /* File descriptors */
        int fd_monitor;
        /* Buffer (pre-allocated) */
        pmc_sample_t* samples;
        unsigned int pmcmask;
        unsigned int kern_pmcmask;
        unsigned int nr_pmcs;
        unsigned int nr_virtual_counters;
        unsigned int virtual_mask;
        unsigned int nr_experiments; // Multiplexation...
        unsigned int ebs_on;
        unsigned int nr_samples;
        unsigned int max_nr_samples;
}pmctrack_desc_t;

/* Connect with the performance tool */
int pmctrack_init(pmctrack_desc_t* desc, unsigned max_nr_samples);
int pmctrack_destroy(pmctrack_desc_t* desc);
int pmctrack_clone_descriptor(pmctrack_desc_t* dest, pmctrack_desc_t*
    orig);
int pmctrack_config_counters(pmctrack_desc_t* desc, char* strcfg[], char*
     virtcfg, int mux_timeout_ms);
int pmctrack_start_counters(pmctrack_desc_t* desc);
int pmctrack_stop_counters(pmctrack_desc_t* desc);
void pmctrack_print_counts(pmctrack_desc_t* desc, FILE* outfile, int
    extended_output);
```

Ahora ya podemos rodear las partes de código que queramos monitorizar comenzando con `pmctrack_start_counters()` y acabando con `pmctrack_stop_counters()`. Los datos leídos de esta manera se almacenan en el campo *samples*. Para solicitar a libpmctrack que itere sobre ellos y los muestre debemos usar la función `pmctrack_print_counts()`, que nos los escribirá en el fichero que le pasemos como argumento. El valor *extended_ output* es un booleano con el cual indicamos si queremos la información básica o queremos también información más avanzada como el ID del experimento mostrándose o el core del cual se ha extraído la muestra.

Obsérvese que se puede llamar multiples veces a las funciones de comienzo y parada –start y stop– para monitorizar así varios fragmentos de código. Si bien, se debe hacer una llamada a `pmctrack_print_counts()` entre cada fragmento puesto que con la función `pmctrack_start_counters()` el contador de tamaño del buffer se inicializa a cero.

Finalmente, la función `pmctrack_destroy()` cerrará los ficheros que quedasen abiertos y liberará la memoria reservada para el buffer.

Figura 3.2: Esquema de uso de la interfaz básica de libpmctrack

```c
#include <pmctrack.h>

#define MAX_SAMPLES 20
#define TIMEOUT 100

int main(int argc, char *argv[])
{
    pmctrack_desc_t desc;
    char* strcfg[]={"pmc0,pmc3=0xc4",NULL};
    char* virtual_cfg=NULL;

    /* Initialize the thread descriptor */
    if (pmctrack_init(&desc,MAX_SAMPLES))
        exit(1);

    /* Configure counters */
    if (pmctrack_config_counters(&desc,strcfg,virtual_cfg,TIMEOUT))
        exit(1);

    /* Start counting */
    if (pmctrack_start_counters(&desc))
        exit(1);

    /***** Code to monitor here *****/

    /* Stop counting */
    if (pmctrack_stop_counters(&desc))
        exit(1);

    /* Display information */
    pmctrack_print_counts(&desc, stdout, 0);

    /* Free up memory */
    pmctrack_destroy(&desc);

    exit(EXIT_SUCCESS);
}
```

Esquema de uso libpmctrack

En la figura 3.2 se presenta un posible esquema, en código C, de cómo usar la librería siguiendo todos los pasos mencionados anteriormente. En este esquema, se ha inicializado el buffer con un tamaño máximo de 20 samples y se ha aplicado una frecuencia de muestreo de $100ms$.

La cadena de configuración de los contadores se ha definido en `strcfg` y depende fuertemente de la arquitectura y fabricante del procesador. En nuestro caso, hemos diseñado este esquema para un procesador Intel Core i7 con microarquitectura Ïvybridge", dicho procesador dispone de tres contadores fijos y cuatro configurables. En el esquema, a modo de ejemplo, se activa el contador fijo *pmc0*, el cual cuenta el número de instrucciones retiradas en el procesador; y se fija el contador configurable *pmc3* con el evento con código `0xc4`, el cual cuenta las instrucciones de salto retiradas en el procesador.

Finalmente, la salida de la información de monitorización obtenida se escribe por la salida estándar y se destruye el descriptor para libpmctrack.

3.4.2. pmctrack_internal.h

La interfaz *pmctrack_internal.h* corresponde a una interfaz más avanzada y de más bajo nivel de abstracción para el usuario. Esta interfaz permite el uso de funciones para configurar valores que usa directamente el módulo del kernel de PMCTrack.

Así pues, ahora tenemos el fichero `pmctrack_internal.h` que provee de nuevas funciones, todas ellas precedidas por la palabra `pmct_*`. Además, incluye al fichero `pmctrack.h` con lo cual basta con incluir la interfaz internal para tener toda la API de libpmctrack. El código de este fichero cabecera puede verse en la figura 3.3. Igual que con la interfaz anterior, explicamos brevemente cada función:

pmct_check_counter_config()
: Comprueba que el *string* de configuración para los PMCs y para EBS es correcto, y devuelve los valores que haya extraído así en: `nr_counters, counter_mask, ebs, nr_experiments`. En caso de que no se le pasase un *string* de configuración, esta función permite obtener dichos valores de configuración del fichero `/proc/pmc/properties`.

pmct_check_vcounter_config()
: Comprueba que el *string* de configuración para contadores virtuales es válido y devuelve los valores extraídos en: `virtual_mask y nr_virtual_counters`.

pmct_config_counters()
: Registra un *string* de configuración en la entrada /proc/pmc/config, la cual usará el kernel para saber cómo tiene que configurar la monitorización. Conviene haber comprobado antes el string de configuración con pmct_check_counter_config().

Figura 3.3: Interfaz avanzada libpmctrack

```
int pmct_check_counter_config(char* userpmccfg[],
                              unsigned int* nr_counters,
                              unsigned int* counter_mask,
                              unsigned int* ebs,
                              unsigned int* nr_experiments);
int pmct_check_vcounter_config(char* virtcfg, unsigned int*
    nr_virtual_counters, unsigned int* virtual_mask);
int pmct_config_counters(char* strcfg[]);
int pmct_config_virtual_counters(char* virtcfg);
int pmct_config_timeout(int msecs, int kernel_control);
int pmct_start_counting( void );
void pmct_print_header (FILE* fo, unsigned nr_experiments,
                        unsigned pmcmask,
                        unsigned virtual_mask,
                        int extended_output);
void pmct_print_sample (FILE* fo, unsigned nr_experiments,
                        unsigned pmcmask,
                        unsigned virtual_mask,
                        unsigned extended_output,
                        int nsample,
                        pmc_sample_t* sample);
int pmct_attach_process (pid_t pid);
int pmct_open_monitor_entry(void);
int pmct_read_samples (int fd, pmc_sample_t* samples, int max_samples);
```

pmct_config_virtual_counters()
Registra la configuración de los contadores virtuales pasada como párametro en la entrada `/proc/pmc/config`. Conviene haber comprobado antes el *string* de configuración de los contadores virtuales con `pmct_check_vcounter_config()`.

pmct_config_timeout()
Registra los valores de tiempo en la entrada `/proc/pmc/config`.

pmct_start_counting()
Avisa al kernel de que se quiere empezar la monitorización.

pmct_print_header()
Imprime una cabecera con las columnas correspondientes a la configuración pasada como parámetro.

pmct_print_sample()
Dado un *sample*, escribe en un descriptor de fichero las muestras obtenidas para cada PMC y contador virtual configurado.

pmct_attach_process()
Con esta función se establece la relación entre el proceso monitorizado, cuyo pid se debe pasar como argumento a esta función, y el proceso monitor.

pmct_open_monitor_entry()
: Abre fichero `/proc/pmc/monitor` para lectura. Esto se debe realizar si se desea obtener las muestras de los contadores hardware generadas por otro proceso. Por ejemplo, el programa `pmctrack` hace uso de esta función.

pmct_read_samples()
: Lee el máximo de muestras –o hasta la máxima capacidad del buffer si no se ha definido un máximo– desde el fichero pasado como argumento. Las muestras se devuelven en el parámetro *samples* y el número de muestras leídas es el valor de retorno de la función.

Capítulo 4

PMCTrack-GUI

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

Este capítulo se estructura como sigue. En la sección 4.1 se justifica la necesidad del desarrollo de PMCTrack-GUI. La sección 4.2 presenta las características principales de la aplicación PMCTrack-GUI y se realiza una breve comparativa de ésta con otras aplicaciones gráficas disponibles en el mercado. La sección 4.3 describe detalladamente cómo se usa la aplicación. Finalmente, la sección 4.4 explica detalles del diseño de la aplicación, las tecnologías usadas y sus principales componentes internos.

4.1. Motivación

Como se mencionó en el Capítulo 1, a pesar del gran potencial de la herramienta PMCTrack, ésta 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 toda esa cantidad de información, siendo necesario procesarla a posteriori.

Además, tal y como se explicó en el Capítulo 1 de esta memoria, el usuario no puede proporcionar al comando `pmctrack` el nombre del evento que desea contabilizar en un PMC ("Instrucciones retiradas" por ejemplo). En lugar de ello, debe proporcionar los códigos hexadecimales que se correspondan con el evento en cuestión, siendo estos códigos diferentes según la arquitectura y el modelo de procesador que estemos usando. Esto obliga al usuario a utilizar un manual de la arquitectura que esté usando para buscar los códigos hexadecimales del evento que quiera contabilizar.

Para solucionar estos dos problemas, es necesario el desarrollo de un *frontend* gráfico que facilite al usuario la tarea de monitorización de aplicaciones usando PMCs.

4.2. Ventajas y características

PMCTrack-GUI ha sido desarrollado para superar las limitaciones comentadas en la sección anterior, pero el resultado final es una herramienta que no sólo supera estas limitaciones sino que ofrece características que no soportan otras aplicaciones alternativas disponibles en el mercado.

A la fecha de escritura de estas líneas, PMCTrack-GUI es la única aplicación que permite visualizar gráficas de rendimiento de aplicaciones en tiempo real en múltiples arquitecturas usando PMCs. Actualmente distintos fabricantes de microprocesadores como Intel y ARM proporcionan herramientas gráficas compatibles únicamente con sus modelos de procesador. Ejemplos destacados de estas herramientas, son el *Intel Performance Counter Monitor* [9] y el *Streamline Performance Analyzer* de ARM [2].

PMCTrack-GUI, por el contrario, no solo ofrece soporte multiarquitectura sino que abstrae completamente al usuario de la arquitectura que esté utilizando. 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 con cada contador hardware. PMCTrack-GUI se encargará internamente de obtener los códigos hexadecimales asociados a los eventos y a la arquitectura en cuestión. No obstante, la aplicación cuenta con una configuración avanzada en la que se permite al usuario asignar eventos a contadores proporcionando los códigos hexadecimales asociados. Cabe destacar que PMCTrack-GUI también soporta la monitorización de aplicaciones multihilo, de forma que es posible visualizar gráficas en tiempo real de un determinado hilo de la aplicación que se esté monitorizando.

Para especificar las métricas de alto nivel que serán visualizadas posteriormente en gráficas en tiempo real, el usuario tan sólo debe escribir fórmulas de alto nivel 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 caché (LLC), la fórmula $(pmc3 * 1000)/pmc0$ define la métrica de rendimiento "*Número de fallos de LLC por cada 1K instrucciones*".

El código fuente de PMCTrack-GUI 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. Esto proporciona otra ventaja sobre la mayoría de aplicaciones alternativas, ya que estas no suelen ser libres y sus licencias de uso suelen ser de un alto coste económico.

Aunque ya se han comentado las principales características con las que cuenta PMCTrack-GUI, a continuación destacamos otras características relevantes:

- Permite la monitorización de máquinas remotas accesibles por SSH, desacoplando la GUI de la propia monitorización.
- Permite observar simultáneamente y en tiempo real dos o más gráficas de métricas de rendimiento, para el mismo o distintos hilos de ejecución de la aplicación monitorizada.

- Permite parar/reanudar la ejecución de la aplicación que se está monitorizando.
- Permite realizar una nueva configuración a partir de otra anterior cuya monitorización se está llevando a cabo.
- Permite personalizar el aspecto visual de las gráficas.
- Permite realizar capturas de las gráficas.

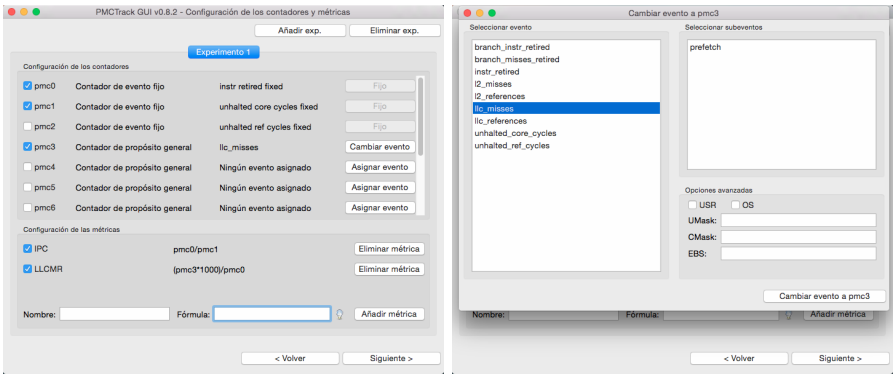
4.3. Modo de uso

Al ejecutar PMCTrack-GUI se inicia con el idioma que se encuentre configurado en el sistema operativo de la máquina que lo arranca. Actualmente se proporciona la traducción para los idiomas español e inglés únicamente, de modo que la aplicación se visualizará en español si la sesión del usuario del SO está configurada en español, o inglés en otro caso. No obstante, incluir soporte para otros idiomas no requeriría modificar el código fuente de PMCTrack-GUI sino que simplemente se ha de proporcionar un fichero, con formato reconocible por *gettext*, con la traducción para el idioma en cuestión.

En primer lugar, el usuario debe seleccionar la máquina que desea monitorizar, pudiéndose elegir la máquina donde se está ejecutando PMCTrack-GUI u otra máquina remota accesible por SSH. En cualquiera de los casos PMCTrack-GUI hará un chequeo para comprobar que está instalado el software necesario tanto en la máquina a monitorizar como en la máquina donde se está ejecutando la GUI. En caso de que falte algún componente software requerido se informará debidamente al usuario.

Si todas las dependencias software están resueltas y se establece correctamente la conexión con la máquina remota (si fuera necesario), se mostrará una nueva ventana donde el usuario podrá configurar los PMCs con los que cuenta la máquina a monitorizar, asignando eventos a dichos contadores. La figura 4.1(a) muestra una captura de dicha ventana. Para realizar la asignación de eventos, el usuario sólo tiene que hacer clic en el botón "Asignar evento" del contador que quiera configurar, y a continuación, hacer clic en el evento y subevento(s) que quiera asignar de entre todo el listado de eventos y subeventos disponibles que se mostrarán por pantalla. Adicionalmente, se le da la posiblidad a los usuarios avanzados de asignar eventos al contador proporcionando los códigos hexadecimales correspondientes a los eventos en cuestión, a través del cuadro de "Opciones avanzadas". En la imagen de la figura 4.1(b) se puede visualizar una ventana de configuración de un 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. Este cuadro 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 $((pmc0\hat{}2)/pmc1 * 1000) * pmc4$. En la figura 4.1(a) se puede observar la ventana de configuración de contadores y métricas donde hay añadidos otros ejemplos de fórmulas.



(a) (b)

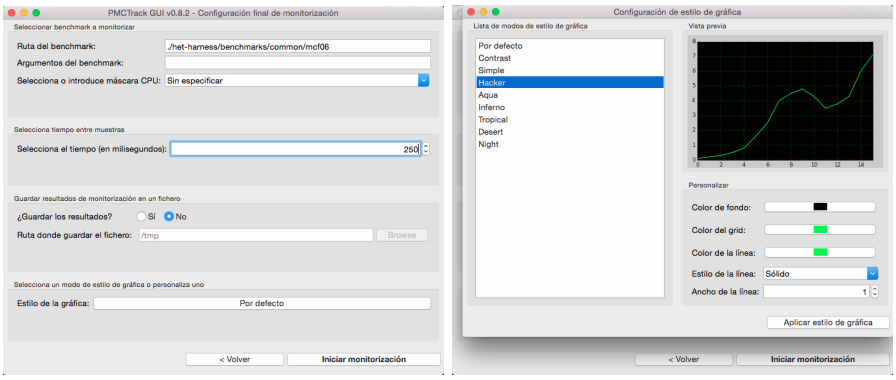
Figura 4.1: Ventanas de configuración de contadores y métricas

Cabe destacar que es posible crear más de un experimento, esto es, más de un conjunto de contadores y métricas. De este modo, es posible configurar un contador con un determinado evento y usarlo en una métrica y, en otro experimento, configurar el mismo contador con otro evento distinto y usarlo en otra métrica distinta.

Cuando el usuario haya terminado de configurar todos los experimentos que quiera y haya hecho clic en el botón "Siguiente" aparecerá una nueva ventana, en esta ventana se realizan las últimas configuraciones antes de iniciar la monitorización. Permite elegir la aplicación que se desea monitorizar, el tiempo entre cada muestra y la ruta del archivo donde guardar los resultados del comando PMCTrack generado (si es que el usuario quiere guardarlo). Adicionalmente, en esta ventana también se da la posibilidad al usuario de personalizar el aspecto visual de las gráficas que serán visualizadas posteriormente, pudiéndose elegir un modo ya configurado (modo por defecto, modo de contraste, modo hacker, modo aqua...) o personalizar uno. En la figura 4.2 se puede ver de forma gráfica lo comentado en este párrafo.

Una vez esté todo configurado y el usuario esté listo para iniciar la monitorización, hará clic en el botón "Iniciar monitorización" de la última ventana de configuración. Al hacer clic se abrirá una nueva ventana donde se visualizará la gráfica en tiempo real de la primera métrica del primer experimento configurado. En el caso de que la aplicación a monitorizar fuera multihilo se mostrará la gráfica del hilo *main*. En cualquier momento podemos realizar las siguientes acciones:

- *Mostrar otra gráfica distinta.* Seleccionando el experimento, métrica de ese experimento y PID del hilo (en el caso de que la aplicación sea multihilo). El usuario podrá elegir entre mostrar la gráfica en la ventana actual o en una nueva ventana independiente, esto proporciona una gran flexibilidad al poder visualizar simultáneamente tantas gráficas como desee el usuario.



(a) (b)

Figura 4.2: Fase final de la configuración

- *Mostrar gráfica acumulada o parcial.* Por defecto el usuario visualiza la última parte de la gráfica, es decir, la parte actual de la monitorización. Sin embargo, es posible cambiar el modo de la gráfica a gráfica acumulada, visualizando la gráfica desde que se inició la monitorización.

- *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.

- *Ocultar controles.* Para ver una gráfica lo mejor posible se le da la posibilidad al usuario de ocultar todos los controles de la ventana, de esta manera la gráfica ocupa todo el espacio de la ventana.

- *Parar/reanudar la aplicación.* El usuario puede parar la ejecución de la aplicación que se está monitorizando cuando lo desee, pudiéndola reanudar posteriormente. Esto puede servir para realizar capturas de gráficas en un punto escogido de la ejecución con más precisión.

En la figura 4.3 se puede ver un ejemplo gráfico de lo explicado anteriormente.

Cabe destacar que mientras se está realizando la monitorización, el usuario puede seguir desplazándose por las ventanas de configuración para preparar una nueva monitorización a partir de la ya existente. No hay ninguna restricción a la hora de cambiar parámetros de configuración, se permite incluso establecer otra máquina distinta accesible por SSH donde llevar a cabo la monitorización, y en ningún momento estos cambios afectarán a la monitorización que se está llevando a cabo. Cuando el usuario tenga lista la nueva configuración, hará clic en el botón “Cancelar monitorización”, en el caso de que

Figura 4.3: Monitorización de una aplicación de usuario en la que se están visualizando simultáneamente en tiempo real dos métricas

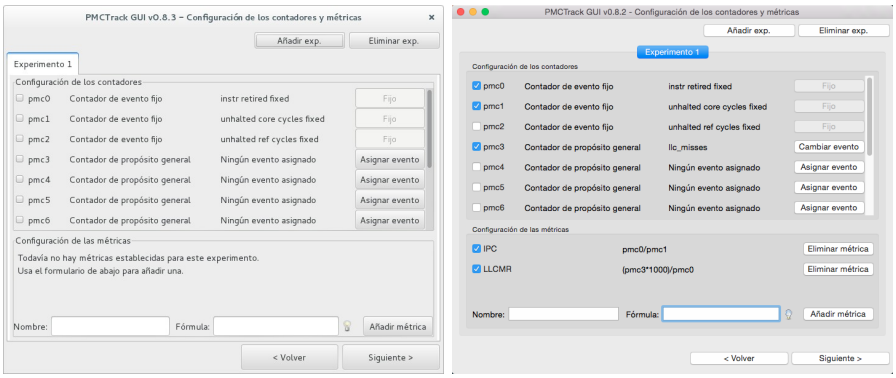
se esté llevando a cabo una monitorización, para matar el proceso de monitorización en la máquina en cuestión. Al hacerlo, se cerrarán todas las ventanas de monitorización, y el botón pasará a llamarse "Iniciar monitorización". Al volver a hacer a clic sobre ese botón se pondrá en marcha la nueva monitorización.

Cuando la aplicación que se está monitorizando termina, se le notifica al usuario mediante un cuadro de diálogo, pero en ningún caso se cierran las ventanas de monitorización. De este modo, el usuario puede seguir realizando las acciones que hemos visto anteriormente.

4.4. Diseño y detalles de implementación

La aplicación PMCTrack-GUI ha sido implementada en Python, eligiéndose este lenguaje principalmente por dos motivos. En primer lugar, por ser multiplataforma, de tal manera que su código puede ser ejecutado en cualquier sistema que soporte la instalación de un intérprete de Python. En segundo lugar, porque cuenta con una librería que permite la generación de gráficos de alta calidad y precisión a partir de datos contenidos en listas o arrays, la denominada librería *matplotlib*. El proceso de generación de gráficas con esta librería consume pocos recursos. Hemos observado que aún realizando la monitorización de una aplicación en la misma máquina que ejecuta la GUI, los resultados de monitorización apenas se ven afectados por la propia ejecución de PMCTrack-GUI para intervalos de muestreo moderados.

Para el desarrollo de las ventanas y diálogos con los que interactúa el usuario se ha utilizado la librería *wxPython*, un *wrapper* de Python de la librería *wxWidgets* escrita en C++. Al igual que Python, esta librería es multiplataforma, y adicionalmente, permite incluir dentro de las ventanas gráficos generados con la librería *matplotlib* antes mencionada. Gracias a que tanto la librería *wxPython* como el propio Python son mul-



(a) Interfaz GNOME (b) GUI nativa de MacOS X

Figura 4.4: PMCTrack-GUI ejecutándose en GNU/Linux y MacOS X

tiplataforma, PMCTrack-GUI también lo es. No obstante, a pesar de que teóricamente PMCTrack-GUI puede ejecutarse en cualquier sistema con un intérprete de Python, por motivos de implementación internos[1] la aplicación sólo es completamente funcional en GNU/Linux y Mac OS X. En la figura 4.4 se puede observar una vista de la aplicación en las dos plataformas soportadas.

PMCTrack-GUI ha sido diseñada para ser fácilmente extensible y mantenible. La justificación de esta afirmación es que, puesto que PMCTrack es una herramienta en continuo desarrollo, se ha dado soporte a PMCTrack-GUI para el uso de nuevas funcionalidades de la herramienta PMCTrack que fueron añadidas por desarrolladores externos simultáneamente al desarrollo de nuestro proyecto. Un ejemplo de esto es el soporte al muestreo basado en eventos (EBS) de la herramienta `pmctrack` de línea de comandos. Además, se ha otorgado a PMCTrack-GUI funcionalidades que no estaban inicialmente previstas, pero que dado el gran potencial que podían ofrecer a la aplicación, se decidieron incluir. Un ejemplo de ello es la posiblidad de monitorizar máquinas remotas accesibles por SSH.

En las siguientes secciones de este capítulo analizaremos los principales componentes internos con los que cuenta PMCTrack-GUI para proveer al sistema de ventanas de la aplicación los datos necesarios para poder llevar a cabo sus funciones.

[1] Esencialmente, el modo de monitorización por SSH de PMCTrack-GUI se basa en el uso de comandos que no están disponibles actualmente en Microsoft Windows.

4.4.1. Objetos de procesamiento

Esta parte consta de información sobre la máquina y los eventos hardware que puede monitorizar, así como de las clases Python que permiten su procesado y le facilitan dicha información al *sistema de ventanas* de la aplicación. Para desacoplar esta parte del resto de la aplicación, la información es servida al sistema de ventanas usando el patrón de diseño *Fachada*, implementado en la clase `FacadeXML`. En el apéndice D.1 se puede encontrar un diagrama UML completo del diseño que corresponde a toda esta parte.

Para obtener la información servida por la fachada, en primer lugar, el sistema de ventanas necesitará construir un objeto fachada y a través de ese objeto podrá obtener la información que quiera haciendo uso de las funciones que dicho objeto provee. La fachada es suficientemente inteligente para obtener la información del modelo que se está usando automáticamente y devolver sus eventos, sin que el sistema de ventanas tenga que especificar el modelo, si bien éste podría ser especificado si así se desease. Además, la fachada también detectará si estamos en una arquitectura asimétrica o heterogénea [13] en la cual las CPUs que podemos tener pueden disponer de eventos diferentes y también precisar de configuraciones diferentes.

Muchas veces, los valores de retorno serán objetos Python que encapsulan todos los datos necesarios. Por ejemplo, la fachada provee de la función `getAvailableEvents` que devuelve una lista de objetos `Event` los cuales contienen todos los campos para describir a un evento y ser usado desde la parte gráfica: nombre, descripción, código, flags y subeventos.

Internamente, cuando se le solicita información a la fachada, ésta crea las estructuras de datos necesarias, procesa los argumentos, hace diversas llamadas a las funciones necesarias para obtener la información deseada y devuelve la información procesada como valores de retorno. En particular, disponemos de una clase `Parser` que es la encargada de tratar con los archivos XML, siguiendo su formato de entrada bien definido por los ficheros DTD (*Document Type Definition*). En la figura 4.5 se muestra un pequeño esquema que ilustra el flujo de acceso a los objetos de procesamiento.

Los objetos de procesamiento se pueden dividir según su formato, en dos grupos: XML y texto plano.

1. XML: Dependientes de cada modelo de CPU, contienen la información de eventos e información de ese modelo de máquina en particular necesaria para la GUI,

2. Texto plano: tienen información más general de la máquina, proveída por el kernel Linux.

En los siguientes apartados, profundizaremos más detalladamente sobre qué información contienen y cómo están estructurados cada uno de estos elementos.

Figura 4.5: Flujo de acceso a cada uno de los objetos de procesamiento

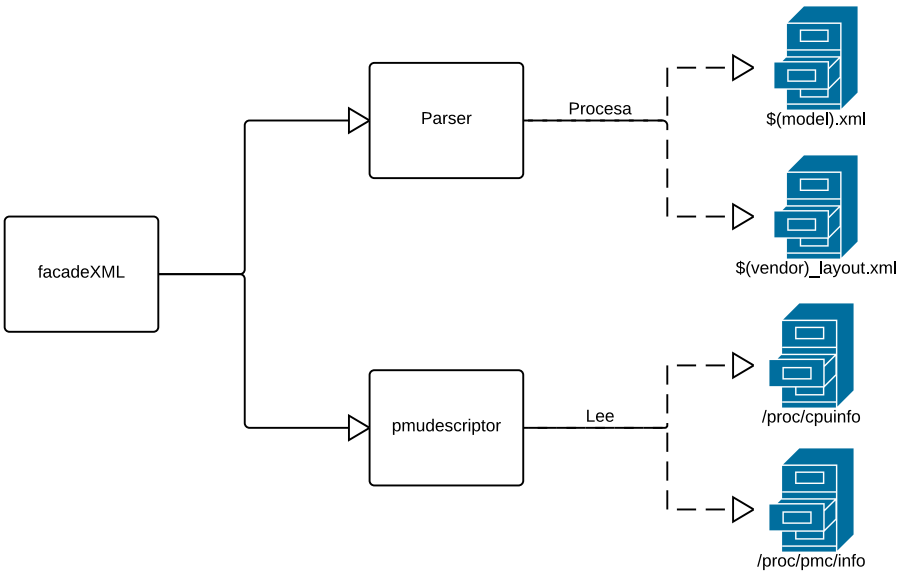


Figura 4.6: Fichero de definición DTD para los ficheros XML que definen el layout de los PMCs

```
<!ELEMENT layout (field*)>
<!ATTLIST layout
vendor (intel|amd|arm|undefined) #IMPLIED
family CDATA #IMPLIED
version CDATA #IMPLIED>

<!ELEMENT field (name,nbits,default)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT nbits (#PCDATA)>
<!ELEMENT default (#PCDATA)>
```

4.4.1.1. XML

Los ficheros XML son archivos escritos con la información organizada por etiquetas y permiten el almacenaje y procesado de la información de una manera sencilla y rápida. Además, son fáciles de leer y de editar manualmente. Por estas razones, hemos escogido almacenar toda la información sobre los eventos y los PMCs que necesitábamos para la interfaz en este formato.
Para mantener una definición formal del formato de XML que queremos leer como entrada y así también poder verificar que los datos del fichero XML son sintácticamente correctos, hemos creado un fichero DTD para cada tipo de XML que queremos usar.

En particular, los tipos de fichero XML que necesitamos son dos:

1. `{nombre_fabricante}_layout.xml`: Este fichero suele ser común a todos los modelos de un mismo fabricante y sirve para definir los campos configurables de cada contador y sus valores por defecto. Su definición se puede ver en el dtd de la figura 4.6.

2. `{nombre_modelo}.xml`: Este fichero contiene información relativa a los eventos, subeventos y contadores fijos de un modelo en particular. Aunque modelos del mismo fabricante tienen algunos eventos en común, sucede que muchos eventos cambian de modelo en modelo o de contadores fijos, de modo que se debe tener un fichero XML por cada modelo de CPU que queramos soportar en la interfaz gráfica. Su definición se puede ver en el DTD de la figura 4.7.

Estos ficheros XML han sido generados de dos maneras diferentes. Los de tipo layout para cada fabricante y para algunos de los eventos de cada modelo han sido escritos manualmente. Para otros modelos de procesador, sobre los que el grupo ArTeCS estaba investigando con la herramienta PMCTrack, ya existían unos ficheros CSV que describían los eventos y PMCs de esos modelos. Para reutilizar estos ficheros, escribimos un script Bash que genera automáticamente los XML equivalentes a esos ficheros CSV dados.

Figura 4.7: Fichero de definición DTD para los ficheros XML que definen los contadores fijos y los eventos de cada modelo

```
<!ELEMENT pmcs_and_events (pmcs?,events)>

<!ELEMENT pmcs (pmc*)>
<!ELEMENT pmc (pmc_name,pmc_type,pmc_number)>
<!ELEMENT pmc_name (#PCDATA)>
<!ELEMENT pmc_type (#PCDATA)>
<!ELEMENT pmc_number (#PCDATA)>

<!ELEMENT events (event+)>
<!ELEMENT event (name,descp?,code,flags?,subevents?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT code (#PCDATA)>
<!ELEMENT descp (#PCDATA)>
<!ELEMENT flags (flag*)>
<!ELEMENT subevents (subevent+)>

<!ELEMENT subevent (subevt_name,subevt_descp?,flags)>
<!ELEMENT subevt_name (#PCDATA)>
<!ELEMENT subevt_descp (#PCDATA)>

<!ELEMENT flag (flag_name,flag_value)>
<!ELEMENT flag_name (#PCDATA)>
<!ELEMENT flag_value (#PCDATA)>
```

4.4.1.2. Texto plano

Además de los ficheros XML, existen dos ficheros de texto plano que resultan relevantes para nuestra interfaz y que son procesados por esta parte de ella. Procedemos a explicarlos brevemente a continuación:

- `/proc/pmc/info`: De este fichero se obtiene información relativa al número de contadores que ofrece la máquina, el nombre del modelo (o nombres de los modelos si se tratase de una arquitectura heterogénea) y alguna otra información que pudiera se necesaria en un futuro. Este fichero es proveído por el módulo común a todas las arquitecturas del kernel de PMCTrack.
- `/proc/cpuinfo`: De este fichero se obtiene el número de *cores* que hay en la máquina que se quiere monitorizar. Este fichero está disponible en cualquier versión del kernel Linux.

4.4.2. Objetos de configuración de usuario

Es un conjunto de objetos que almacenan toda la configuración del usuario. Son enviados de un *frame* de configuración a otro y cada uno de estos frames se encarga de almacenar en estos objetos la parte de configuración que le corresponde.

Cuando toda la configuración de usuario está almacenada, estos objetos son procesados por el componente *PMCExtract* que comentaremos en una sección posterior.

En el apéndice D.2 se puede encontrar un diagrama UML de este conjunto de objetos.

4.4.3. El componente PMCConnect

Es un objeto que, dada una configuración de conexión a una máquina (encapsulada en un objeto de configuración de usuario), provee métodos para obtener todo tipo de información de esa máquina. Permite comprobar si es posible establecer una conexión con la máquina a través de SSH, comprobar la existencia de un fichero, leer el contenido de ficheros y determinar si la máquina tiene un determinado paquete instalado.

Este objeto es usado por el sistema de ventanas para chequear las dependencias software tanto en la máquina donde se está ejecutando PMCTrack-GUI como en la máquina donde se llevará a cabo la monitorización.

Además, es usado por los *Objetos de procesamiento* para leer los archivos en texto plano.

Usando este objeto conseguimos una total independencia entre el resto de componentes de PMCTrack-GUI y la máquina donde se desea llevar a cabo la monitorización. Ni los *Objetos de procesamiento* ni el sistema de ventanas conocen la máquina en sí donde se va a llevar a cabo la monitorización. Gracias a esto, no hay que hacer distinciones en estos objetos dependiendo de si se realiza la monitorización en una máquina remota o en la propia máquina en la cual se está ejecutando PMCTrack-GUI. En vez de ello, cada vez que haya que obtener datos de la máquina, los componentes que requieran esos datos usarán este objeto.

4.4.4. El componente PMCExtract

PMCExtract es el objeto encargado de llevar a cabo la comunicación con la herramienta `pmctrack` de línea de comandos.

En su inicialización, procesa los objetos de configuración del usuario generando el comando `pmctrack` que lanza en la máquina que se haya configurado para monitorizar. Adicionalmente, crea un hilo encargado de extraer toda la información proveniente del comando `pmctrack`, guardándola de forma organizada en un diccionario de datos a medida que el comando `pmctrack` va generando nuevas muestras. Este diccionario de datos será utilizado por el sistema de ventanas para generar las gráficas en tiempo real. Si durante la obtención de muestras el comando `pmctrack` genera un error, el componente *PMCExtract* lo capturará y almacenará, de tal manera que el sistema de ventanas podrá recibir ese error y mostrárselo al usuario mediante un cuadro de diálogo.

PMCExtract cuenta con atributos a los que puede acceder el sistema de ventanas para saber en cualquier momento el estado de la aplicación que se está monitorizando, así como métodos que permiten enviar señales a dicha aplicación (señal de parada *SIGSTOP*, señal de reanudación *SIGCONT* y señal de matar proceso *SIGKILL*).

Capítulo 5

Casos de estudio

En este capítulo evaluamos las nuevas extensiones de PMCTrack realizadas en este proyecto, mediante tres casos de estudio. En el primer caso de estudio explotamos el potencial de PMCTrack-GUI y PMCTrack en su conjunto para recabar información de rendimiento mediante contadores hardware en distintas arquitecturas y diversos modelos de procesador. El segundo caso de estudio ilustra las capacidades de PMCTrack-GUI para monitorizar el rendimiento de hilos individuales de las aplicaciones paralelas. Finalmente, este capítulo concluye con un caso de estudio que pone a prueba la librería libpmtrack para estudiar la efectividad de distintas implementaciones alternativas de una estructura de datos.

5.1. Monitorización del rendimiento con PMCTrack-GUI

Para llevar a cabo nuestro estudio, seleccionamos un subconjunto de 10 benchmarks diversos de la *suite* SPEC CPU2006 [8] y los ejecutamos en múltiples plataformas con procesadores multicore de ARM, Intel y AMD. En particular, experimentamos con la placa de desarrollo Juno de ARM, que integra un procesador big.LITTLE [3]; y con tres servidores con procesadores x86 (Intel Atom, Intel Xeon "Haswell" y AMD Opteron "Magny-Cours"). La especificación detallada de las cuatro plataformas multicore exploradas se muestra en las tablas 5.1-5.4.

Procesador	Intel® Atom® N330 @ 1.6 GHz	
	Numero total de cores	2
	Topología	1 chips, 1 *die* por chip 2 cores por *die* compartiendo la L2 LLC (L2) de 1MB dividida entre cores
Memoria	16 GBytes (DDR2) Plataforma UMA	

Tabla 5.1: Características de la plataforma que integra un procesador Intel Atom.

Procesador	ARM® Big Little® Dual Cluster (ARM v8-A)	
	Numero total de cores	6 (divididos en dos *clusters*)
	Cluster Cortex-A57	2 cores "big" @ 1.10Ghz Cores comparten la L2 (2MB)
	Cluster Cortex-A53	4 cores "LITTLE" @ 850 Mhz Cores comparten la L2 (1MB)
Memoria	8 GBytes (DDR3) Plataforma UMA	

Tabla 5.2: Características de la plataforma que integra un procesador ARM big.LITTLE.

Procesador	4x AMD® Opteron® 6172 @ 2.1 GHz	
	Numero total de cores	48
	Topología	4 chips, 2 *dies* por chip 6 cores por *die* compartiendo la L3 LLC (L3) de 12 MB
Memoria	32 GBytes (DDR3) Plataforma NUMA	

Tabla 5.3: Características de la plataforma que integra un procesador AMD Opteron "Magnycours".

Procesador	Intel® Xeon® E3-1225 v3 @ 3.2GHz	
	Numero total de cores	4
	Topología	1 chip, 1 *die* por chip 4 cores por *die* compartiendo la L3 LLC (L3) de 8 MB
Memoria	32 GBytes (DDR3) Plataforma UMA	

Tabla 5.4: Características de la plataforma que integra un procesador Intel Xeon "Haswell".

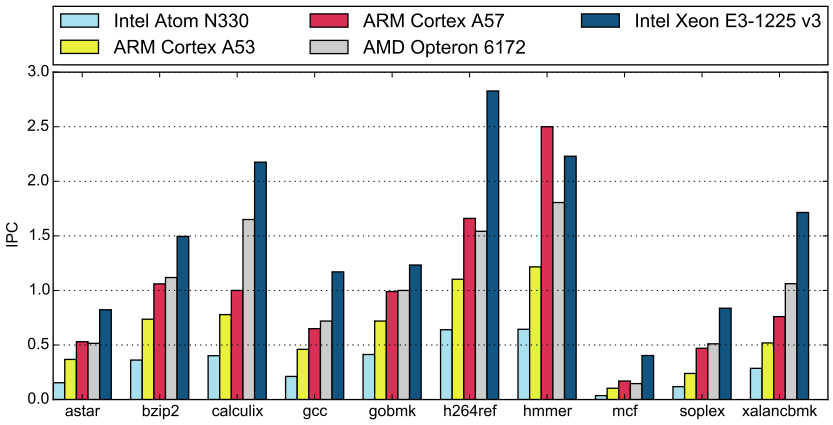


Figura 5.1: Número de instrucciones retiradas por ciclo (IPC) para los distintos *benchmarks*.

Las figuras 5.1, 5.2 y 5.3 muestran respectivamente el número de instrucciones por ciclo (IPC) medio, el número de fallos de último nivel de caché (LLC) y el número de fallos de predicción de saltos por cada mil instrucciones retiradas para los *benchmarks* seleccionados en los distintos tipos de core[1] usados en nuestro estudio. El IPC constituye una métrica global del rendimiento de una aplicación secuencial en una determinada plataforma, 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 de caché o fallos de prediccion de saltos, menor rendimiento experimentará la aplicación en cuestión tanto en procesadores con *pipeline* con ejecución en orden, como el Intel Atom, o con ejecución fuera de orden, como el AMD Opteron.

Para monitorizar las métricas de rendimiento consideradas, empleamos la herramienta PMCTrack-GUI desarrollada en este TFG. Más concretamente, hicimos uso del modo de monitorización remoto de esta herramienta (por SSH) para obtener los datos de las múltiples plataformas desde un PC de escritorio. PMCTrack-GUI no solo 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, después de la ejecución de cada benchmark en cada plataforma, PMCTrack-GUI 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 5.1, 5.2 y 5.3 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.

[1]Nótese que la plataforma de ARM integra un procesador asimétrico con 2 tipos de cores: Cortex-A57 ("big") y Cortex-A53 ("LITTLE"). Por lo tanto, para cada *benchmark* recabamos las métricas de rendimiento en cada tipo de core por separado.

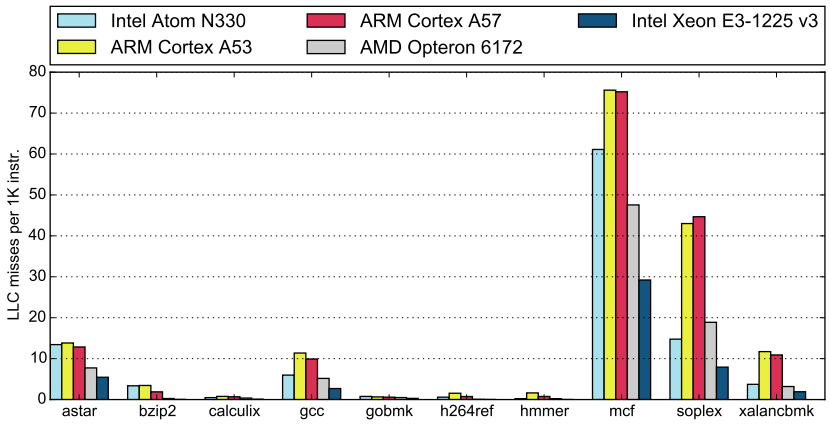


Figura 5.2: Número de fallos de último nivel de caché (LLC) por cada 1K instrucciones retiradas para los distintos *benchmarks*.

Los resultados revelan que los programas `astar`, `gcc`, `mcf`, `soplex` y `xalancbmk` son intensivos en memoria. Como se observa en la figura 5.2 estos benchmarks realizan un número elevado de accesos a memoria (fallos de último nivel de caché) por cada mil instrucciones en todas las plataformas exploradas. En particular, `mcf`, la aplicación con mayor tasa de fallos de caché, es también la que obtiene un menor número de instrucciones por ciclo, seguida de cerca por otras dos aplicaciones intensivas en memoria como `astar` y `soplex`.

Los datos obtenidos también revelan que otros benchmarks que podemos considerar *intensivos en CPU*, por su baja tasa de fallos de LLC, exhiben un IPC relativamente reducido, y comparable al de *benchmarks* intensivos en memoria. Este es el caso de las aplicaciones `bzip2` y `gobmk`. La figura 5.3 ilustra que estas aplicaciones sufren de numerosos fallos de predicción de saltos, lo cual puede derivar en frecuentes paradas del *pipeline* del procesador. Este efecto explica el bajo valor de IPC observado en todas las plataformas para estas aplicaciones.

Finalmente, cabe destacar que aquellas aplicaciones con mayores valores de IPC en todas las plataformas (`calculix`, `h264ref` y `hmmer`) tienen asociado, como cabía esperar, una baja tasa de fallos de caché de último nivel y un número reducido de fallos de predicción de saltos.

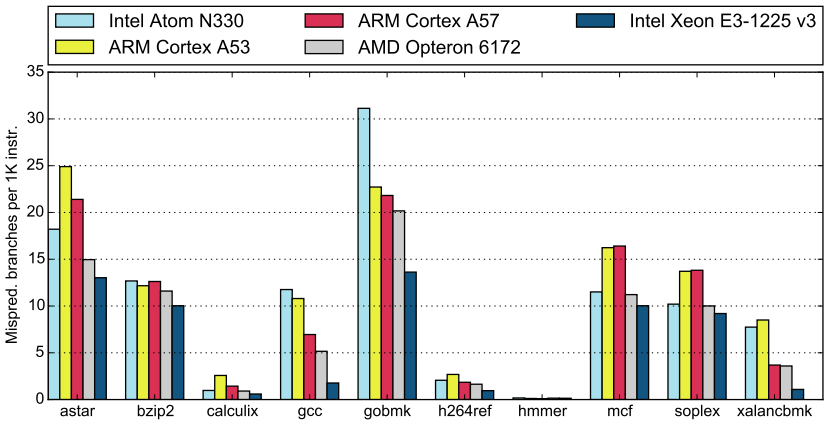
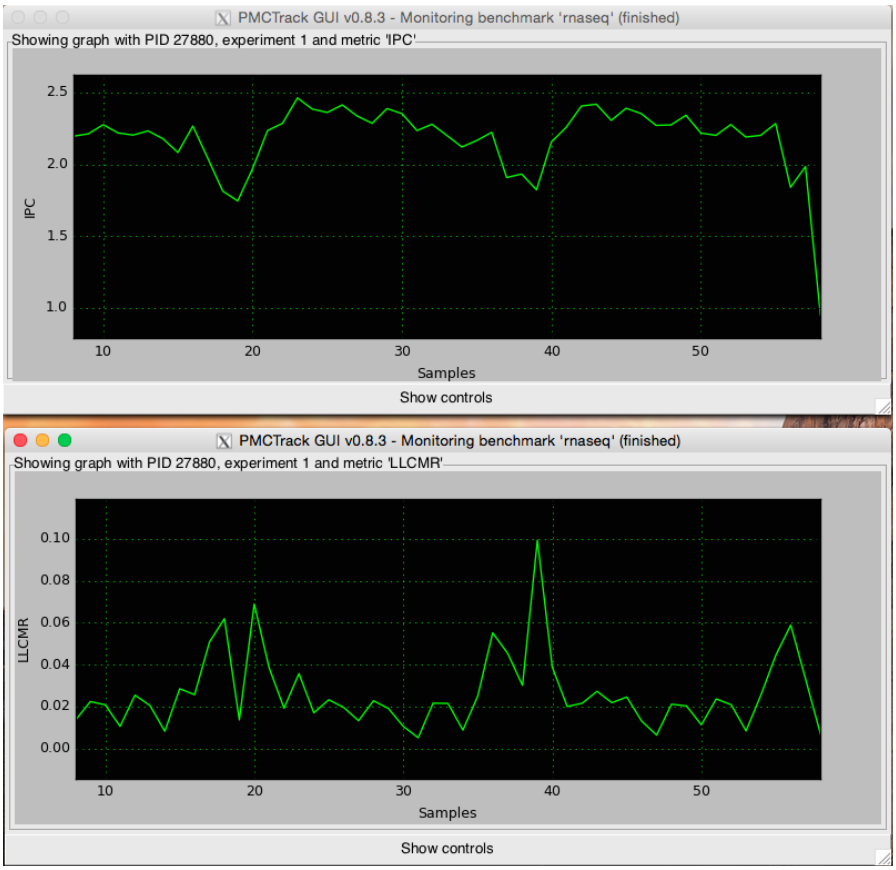


Figura 5.3: Número de fallos de predicción de saltos por cada 1K instrucciones retiradas para los distintos *benchmarks*.

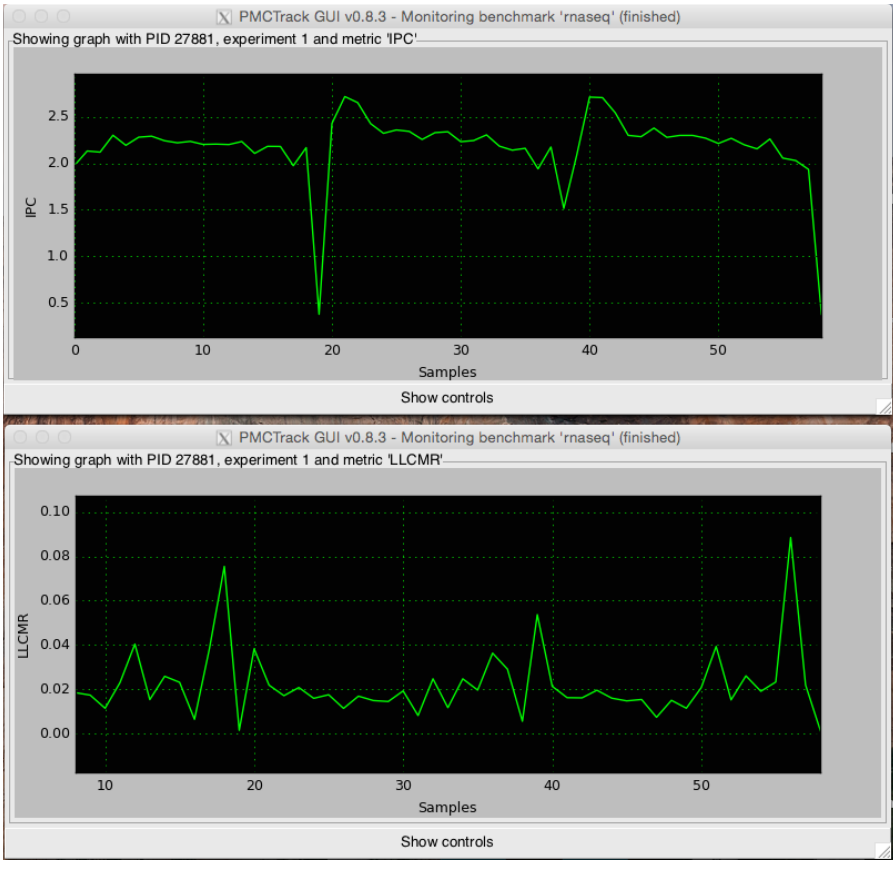
5.2. Análisis de aplicaciones multihilo con PMCTrack-GUI

En esta sección ponemos a prueba la aplicación PMCTrack-GUI desarrollada en este TFG para monitorizar el rendimiento de aplicaciones multihilo. En este caso de estudio nos interesa estudiar el comportamiento de los hilos de dos aplicaciones paralelas muy diferentes. La primera aplicación es `rnaseq`, un benchmark OpenMP de bioinformática. En esta aplicación OpenMP regular, los hilos realizan esencialmente el mismo cómputo sobre distintos datos. La segunda aplicación considerada es `ferret`, que pertenece a la suite de benchmarks PARSEC v1.3 [4]. Esta aplicación POSIX threads sigue el paradigma de programación paralela tipo *pipeline*. En este paradigma, la aplicación consta de distintos tipos de hilos; cada tipo de hilo se ocupa de realizar una fase de procesamiento y el resultado que genera sirve como de entrada para otro tipo de hilo que se encarga de realizar otra fase de procesamiento. En la aplicación `ferret` se espera que el paradigma tipo *pipeline* se refleje en distintos perfiles de rendimiento para cada tipo de hilo, debido al distinto tipo de procesamiento que realizan.

En nuestro estudio ejecutamos ambas aplicaciones con cuatro hilos en el sistema con el procesador Intel Xeon "Haswell" usado también en el caso de estudio previo, y cuyas especificaciones técnicas se detallan en la tabla 5.4. Usando PMCTrack-GUI empleando el modo SSH hemos monitorizado las dos aplicaciones paralelas, configurando la herramienta para que muestre las gráficas de rendimiento en tiempo real de dos métricas: instrucciones por ciclo del procesador (IPC) y fallos del último nivel de caché por cada mil instrucciones retiradas ($LLCMR$). Puesto que PMCTrack-GUI permite visualizar las gráficas de cada hilo individual de la aplicación, podemos comparar el comportamiento de estas dos aplicaciones analizando las gráficas de rendimiento de cada uno de sus respectivos hilos. Las figuras 5.4 y 5.5 muestran los resultados obtenidos.

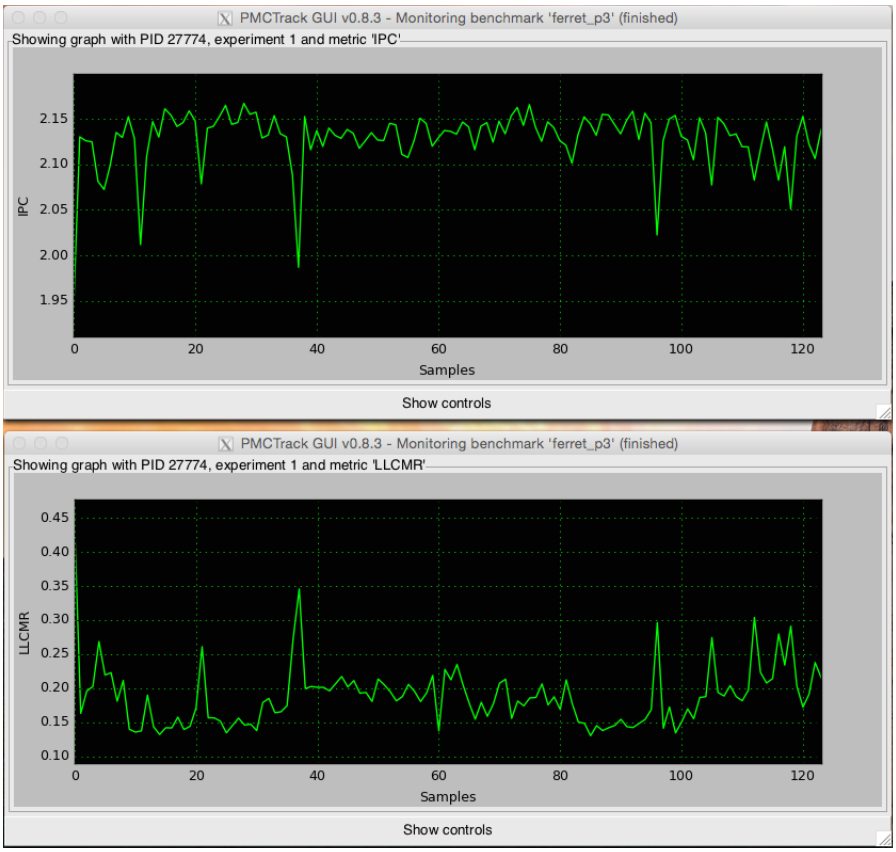


(a)

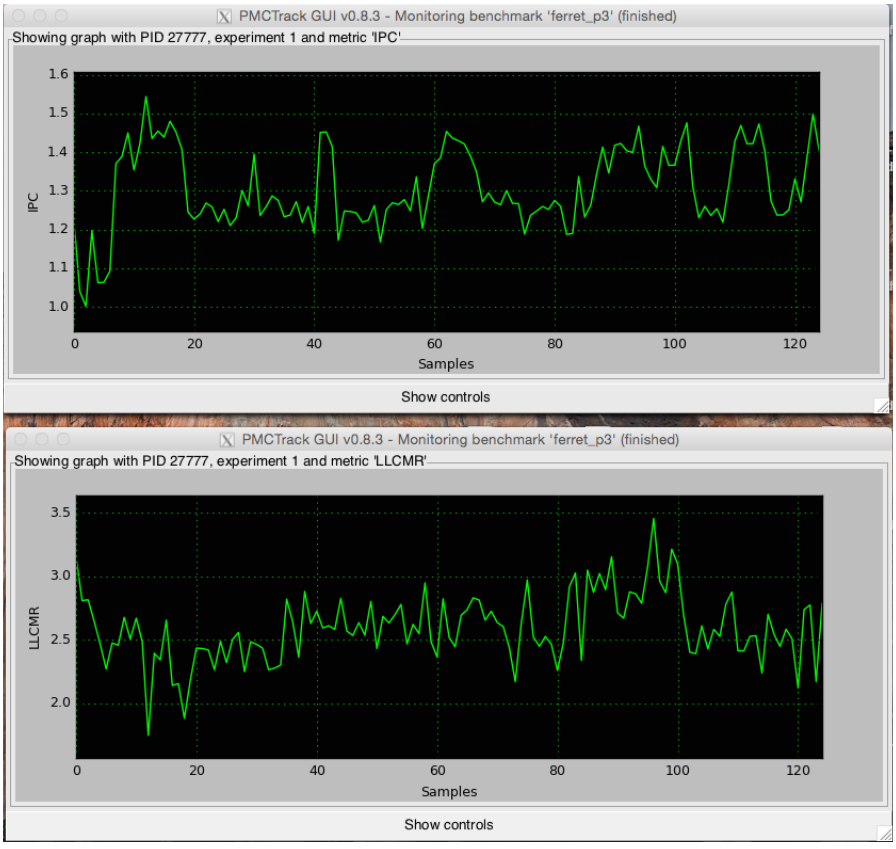


(b)

Figura 5.4: Gráficas de rendimiento asociadas a los hilos con PID 27880 (arriba) y 27881 (abajo) de la aplicación `rnaseq`.



(a)



(b)

Figura 5.5: Gráficas de rendimiento asociadas a los hilos con PID 27774 (arriba) y 27777 (abajo) de la aplicación `ferret`.

Centrándonos en la aplicación `rnaseq`, hemos escogido las gráficas de rendimiento de dos de sus hilos, pudiéndose visualizar en las figuras 5.4a y 5.4b. Comparando las gráficas de los dos hilos de la aplicación observamos que, a pesar de que no se tratan de las mismas gráficas, los resultados obtenidos son muy parecidos cuantitativamente hablando en cuanto a IPC y tasa de fallos de último nivel de caché. Además, podemos observar que ambos hilos atraviesan fases de ejecución muy similares. Estos resultados son perfectamente razonables ya que todos los hilos de la aplicación paralela realizan el mismo procesamiento con distintos datos.

Al igual que en la aplicación `rnaseq`, para la aplicación `ferret` también hemos escogido las gráficas de rendimiento de dos de sus hilos. Las gráficas de estos hilos se pueden visualizar en las figuras 5.5a y 5.5b. En este caso, los resultados revelan patrones de rendimiento muy diferentes en ambos hilos. El hilo con PID 27774 (figura 5.5a) ejecuta de media algo más de 2 instrucciones por ciclo, mientras que el hilo con PID 27777 (figura 5.5b) ejecuta entre 1 y 1,5 instrucciones por ciclo. El primer hilo por tanto está ejecutando entre un 50 y un 100 % de instrucciones por ciclo más que el segundo hilo. En el caso de la segunda métrica (LLCMR) la diferencia de resultados entre los dos hilos en aún más notoria, el hilo con PID 27774 apenas tiene fallos de caché de último nivel (de 0,13 a 0,35 fallos por cada 1000 instrucciones retiradas), mientras que el hilo con PID 27777 tiene entre 1,5 y 3 fallos. El segundo hilo por tanto tiene de media 9 veces más fallos de caché de último nivel que el primero. De estos resultados podemos concluir que en la ejecución del hilo 27777 se realizan muchos más accesos a memoria que en el hilo 27774. Estas diferencias sustanciales en el patrón de rendimiento de ambos hilos se deriva del hecho de que realizan procesamientos muy diferentes en el contexto de la aplicación paralela tipo *pipeline*.

5.3. Análisis de fragmentos de código con libpmctrack: comportamiento de la memoria caché en estructuras de datos con memoria dinámica

En esta sección empleamos la librería *libpmctrack* para analizar el comportamiento del hardware en distintos fragmentos de código.

Como bien sabemos, el acceso a memoria principal en los ordenadores no se hace directamente con peticiones a la RAM, si no que sigue una jerarquía de memorias caché ordenadas por niveles según su tamaño y latencia. Las memorias caché contienen bloques de memoria contiguos buscando explotar así la localidad espacial de los programas. Esto significa que es mucho más lento acceder a direcciones de memoria no contiguas que a direcciones contiguas solo debido a la memoria caché. Cuando en un programa reservamos memoria de forma estática, ya sea en la memoria global o en la pila del programa, se reservan bloques contiguos de memoria, y es por ello que podemos acceder a cada elemento mediante un índice (en realidad, el índice marca un desplazamiento respecto

al inicio del array). Sin embargo, cuando reservamos memoria de forma dinámica, es decir en la *heap* del proceso, los bloques de memoria son asignados según la conveniencia del sistema operativo y muchas veces estos bloques se encuentran en direcciones muy distanciadas dentro de la memoria.

En esta sección, queremos analizar cuánto podría afectar esta diferencia de tiempos de acceso para la implementación de estructuras de datos. Como ejemplo, usaremos dos posibles implementaciones de un montículo o cola de prioridad (en inglés *heap*) según hagan uso de memoria dinámica mediante el uso de punteros, o de memoria estática.

La primera de ellas es el conocido montículo de Fibonacci (*Fibonacci heap*), que tiene muy buenos costes amortizados, siendo sus operaciones de crear, insertar, obtener el mínimo y unir de coste constante en el caso peor; y solo siendo sus operaciones de eliminar de coste amortizado logarítmico. Para conseguir estos costes, el montículo se limita a hacer el mínimo trabajo posible cuando inserta elementos, manteniendo siempre un puntero al mínimo elemento, y se reestructura cuando tiene que hacer una operación de borrar el elemento mínimo.
Los montículos de Fibonacci hacen un uso intensivo de punteros. La estructura consiste en una lista dinámica de árboles de distinto grado donde cada nodo del montículo tiene cuatro punteros: al nodo padre, al nodo hijo, y a sus nodos hermanos izquierdo y derecho. La reestructuración de nodos que realiza en el caso de borrar va creciendo según la secuencia de Fibonacci (he de aquí su nombre).

La segunda de ellas es una implementación que solamente hace uso de memoria estática, la conocida como montículo de William o montículo binario (*Williams'heap*, *binary heap* o *bi-parental heap*). En esta implementación se simula el comportamiento de una estructura de árbol binario balanceado en un array de elementos. El árbol se simula estructurándolo de tal manera que cada nodo siempre tiene su padre en la posición $i/2$, su hijo izquierdo en la posición $2i$ y su hijo derecho en la posición $2i + 1$ del array.
La estructura debe mantener el invariante de que todos los nodos, excepto el nodo raíz, tienen una clave mayor o igual que la clave de su padre. De esta manera se garantiza que, a la hora de actualizar el árbol, solo un logaritmo de todos los elementos tendrá que ser recorrido. Por tanto, en este caso sus costes en el caso peor de insertar y de eliminar son logarítmicos, aunque el de obtener el mínimo elemento sigue siendo constante puesto que siempre estará en el nodo raíz.

Ahora vamos a comprobar si en la práctica se confirman los costes que predice la teoría. Para ello hemos implementado un sencillo benchmark que simplemente inserta y elimina un millón de números en un montículo. Estos números son generados al azar en un rango que va de -5000 a 5000. El benchmark, al terminar, nos dice cuánto tiempo ha transcurrido desde el comienzo de la ejecución con precisión de microsegundos.

5.3.1. Primer análisis

La ejecución se realiza en un sistema con un procesador Intel Core i7-3520M a 2.9GHz y 8GB de RAM. Después de ejecutar el benchmark una primera vez con cada montículo, obtenemos los siguientes tiempos: 232,859ms para el montículo binario y 1690,336ms para el montículo de Fibonacci. Es decir, el montículo de Fibonacci es siete veces más lento que el montículo binario.

Para ver si esta gran diferencia de tiempo es debido a la memoria caché, necesitamos alguna forma de monitorizar directamente cómo se está comportando. Ésta es una situación donde libpmctrack resulta tremendamente útil, ya que nos permite obtener información del hardware e incluso analizar diferentes fragmentos de código de forma aislada.

Ahora procedemos a instrumentar el código del benchmark anterior usando la librería libpmctrack. Más concretamente, inicializamos el descriptor `pmctrack_desc_t` con espacio para 15 muestras, fijamos el intervalo de muestreo a 250ms y configuramos los contadores hardware. Nos interesa que los contadores nos den información acerca del número de instrucciones retiradas, y los accesos y los fallos de la caché de último nivel (Nivel 3 en nuestra plataforma). Para ello, activamos el contador fijo *pmc0*, que cuenta el número de instrucciones, y asignamos a los contadores configurables *pmc3* y *pmc4* los eventos para contabilizar accesos y fallos en el último nivel de caché, respectivamente. Comenzaremos situando las llamadas a `pmctrack_start_counters()` y `pmctrack_stop_counters()` counters al principio y al final del código del benchmark.

Tabla 5.5: Resultados monitorización global montículo binario

nsample	pid	event	pmc0	pmc3	pmc4
1	7377	tick	561444260	3966742	339785
2	7377	self	549334613	4032294	151245

Los resultados obtenidos se encuentran en las tablas 5.5 y 5.6. Lo primero que podemos apreciar es que el montículo de Fibonacci tiene 13 muestras o *ticks* y el binario solo dos puesto que, como dijimos antes, el montículo binario es siete veces más rápido en este caso. Si sumamos los valores de todas las muestras, aproximadamente obtenemos los siguientes datos:

- 1150 millones de instrucciones el montículo binario y 5000 el de Fibonacci (casi cinco veces más instrucciones el montículo de Fibonacci). Probablemente debido a la mayor complejidad de las operaciones que realiza el montículo de Fibonacci.

- 7,5 millones de accesos a memoria caché en el binario y 25 millones en el de Fibonacci; por tanto, tres veces más accesos a caché por el montículo de Fibonacci. Como vemos, aunque el tamaño de los datos de entrada es el mismo –un millón de enteros–, el montículo de Fibonacci tiene que acceder muchas más veces a memoria caché de último nivel.

Tabla 5.6: Resultados monitorización global montículo Fibonacci

nsample	pid	event	pmc0	pmc3	pmc4
1	7558	tick	629554817	555265	339547
2	7558	tick	365243640	2111969	967323
3	7558	tick	363263873	2174515	943288
4	7558	tick	366034887	2190378	888714
5	7558	tick	361822568	2208562	914895
6	7558	tick	361998147	2201110	893159
7	7558	tick	358977538	2206190	879025
8	7558	tick	376751462	2220617	815036
9	7558	tick	371586535	2229520	828087
10	7558	tick	371069310	2196560	795315
11	7558	tick	376280116	2186783	740499
12	7558	tick	388247900	2161197	641935
13	7558	self	341342501	1586504	322759

- 0.4 millones de fallos a memoria caché de último nivel por el montículo binario y 10.5 millones por el montículo de Fibonacci. Aquí podemos comprobar como, efectivamente, la diferencia de fallos de acceso a la memoria caché es abismal, más de 20 veces más fallos en la memoria caché tiene el montículo de Fibonacci frente al montículo binario.

Resulta también de gran relevancia calcular la tasa de fallos de la memoria caché, esta sería de un 5,33 % en el caso del montículo binario, y de un 42 % en el caso del montículo de Fibonacci. De nuevo, una diferencia muy significativa.

Por tanto, todo apunta a que esta gran diferencia en el número de accesos a memoria caché y, sobre todo, en sus porcentajes de aciertos, hace muy superior en la práctica al montículo binario frente al montículo de Fibonacci, a pesar de que la teoría predijera lo contrario.

5.3.2. Segundo análisis

No obstante, podemos afinar aún más y obtener mayor información. Revisando el análisis anterior, observamos que la primera muestra del montículo de fibonacci no está equilibrada con el resto de muestras: ejecuta el doble de instrucciones que el resto, y, sin embargo, hace muchos menos accesos a memoria; lo que nos lleva a pensar que podría haber una fase inicial en las que el montículo invierta menos tiempo y menos acceso a recursos que el resto. Para comprobar esto, también mediante libpmctrack podemos monitorizar fragmentos de código aislados.

Para ello, situamos tres bloques `pmctrack_start_counters()`/`pmctrack_stop_counters()` en torno a tres partes clave de nuestro pequeño benchmark: uno para la inicialización del montículo, otro para la inserción del millón de números y un tercero para la eliminación de éstos. Estas tres partes se corresponderían con tres fases o etapas diferentes de operaciones con la estructura de datos. Además, puesto que las franjas de start y stop ahora son suficientemente pequeñas, también cambiamos la configuración para que no se haga captura de muestras por tiempo, fijando el timeout a 0, y tener de esta manera todos los datos de cada fase en una sola muestra.
Los resultados los podemos ver en las tablas 5.7 y 5.8.

Tabla 5.7: Resultados monitorización por fases montículo binario

stage	nsample	pid	event	pmc0	pmc3	pmc4
initialization	1	13464	self	833	10	7
insertion	1	13464	self	194528249	19764	17205
deletion	1	13464	self	916232599	7383241	467662

Tabla 5.8: Resultados monitorización por fases montículo Fibonacci

stage	nsample	pid	event	pmc0	pmc3	pmc4
initialization	1	13271	self	835	11	10
insertion	1	13271	self	377048052	15204	9962
deletion	1	13271	self	4655118486	37307044	14266564

Con estos nuevos datos, podemos ver claramente que la fase de inicialización y reserva inicial de memoria es prácticamente la misma para ambas estructuras en cuanto a número de instrucciones y acceso a memoria.

En la fase de inserción, la diferencia tampoco es demasiado grande, de hecho, aunque el montículo de Fibonacci realiza más instrucciones, efectivamente cumple su teoría en cuanto a que ejecuta menos accesos a memoria y éstos son más exitosos que el montículo binario.

La gran diferencia llega en la última fase, la fase de borrado de elementos, en ésta el número de instrucciones ejecutadas es mucho mayor para ambos montículos. Es en esta fase cuando el montículo de Fibonacci se dispara en cuanto a número de accesos a memoria y la tasa de fallos es tan alta como el 40 % mencionado anteriormente, eclipsando cualquier buen resultado que se podía haber obtenido en fases anteriores; mientras que el montículo binario, mantiene una buena tasa de fallos y un número de accesos relativamente mucho menor.

5.3.3. Conclusiones

Después de un primer análisis global y, posteriormente, un segundo análisis pormenorizando el benchmark en tres fases clave. Hemos comprobado, gracias a la librería libpmctrack, como la estructura de datos montículo o *heap* puede tener un rendimiento muy diferente en el borrado, según si la implementación hace uso de memoria dinámica o, por el contrario, hace uso de memoria estática.

En general, con este caso de estudio observamos que el uso de memoria dinámica para la implementación de estructuras de datos puede perjudicar en gran medida sus tiempos de ejecución, debido a la jerarquía de acceso a la memoria que existe en las computadoras actuales.

Debemos de aclarar, no obstante, que este análisis se ha realizado con tipos primitivos (enteros) como elementos contenidos en la estructura de datos. Los resultados tendrían que ser revisados si dichos elementos fuesen objetos o punteros a otras estructuras, lo cual estropearía la ventaja que tiene el montículo binario gracias a la localidad espacial de sus elementos.

Capítulo 6

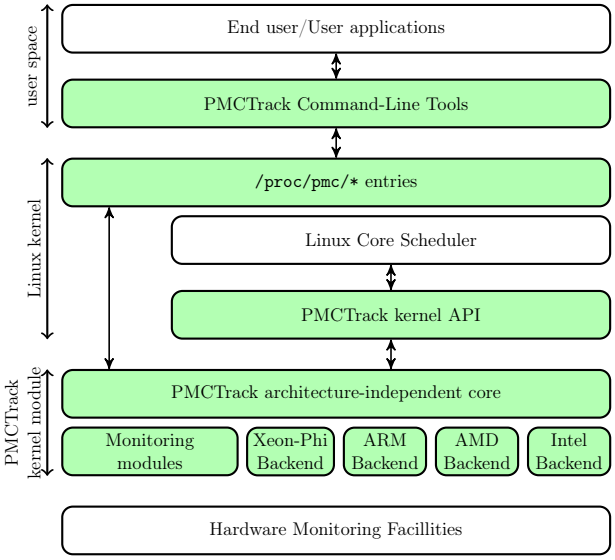
Conclusiones y trabajo futuro

6.1. Conclusiones

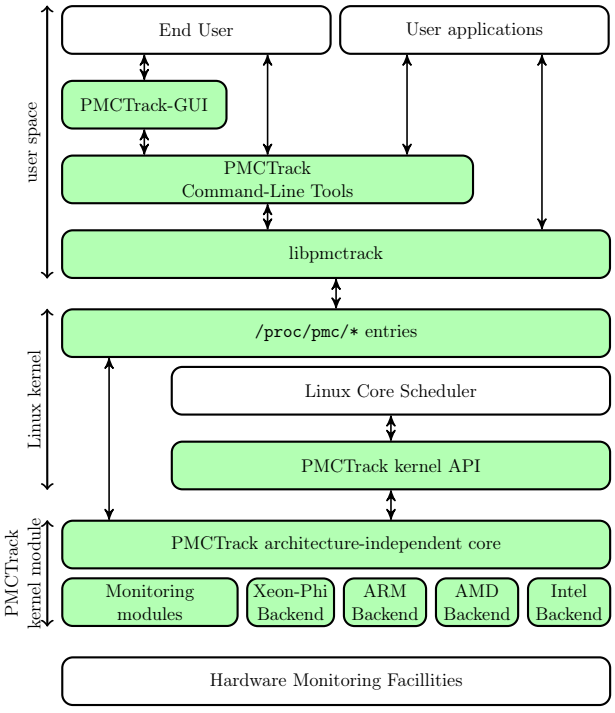
Después de la realización de este proyecto, PMCTrack cuenta con tres nuevas características: (1) soporte de monitorización para aplicaciones multihilo, (2) la librería libpmctrack –que permite la monitorización de fragmentos de código con los contadores hardware–, y (3) la aplicación PMCTrack-GUI –un *frontend* gráfico para PMCTrack que facilita su uso y permite obtener gráficas en tiempo real de métricas de alto nivel definidas por el usuario–. En los capítulos 2, 3 y 4, hemos explicado en detalle el funcionamiento y el proceso de diseño e implementación de cada una de estas características. En el capítulo 5, hemos puesto a prueba su funcionamiento y mostrado su gran utilidad mediante tres casos de estudio. Para finalizar, analizaremos ahora a fondo el impacto holístico de cada una de estas nuevas características.

En la figura 6.1 podemos ver dos diagramas. El primero es el que ya presentamos en el capítulo de Introducción en este mismo documento y corresponde a la arquitectura de PMCTrack antes de comenzar este TFG. El segundo muestra la nueva arquitectura de PMCTrack al finalizar nuestro proyecto. Podemos observar cómo los cambios han afectado a todos los niveles de PMCTrack: desde el nivel del kernel –al modificar y rediseñar el módulo para que permita recabar muestras de los contadores hardware de varios hilos de una misma aplicación–, pasando por el nivel del espacio de usuario con libpmctrack, hasta finalmente llegar al nivel más alto de aplicaciones gráficas que permiten al usuario final interactuar con la herramienta –con el desarrollo de PMCTrack-GUI–.

En primer lugar, el soporte multihilo o *multithreading* permite extender considerablemente la funcionalidad de PMCTrack. Podemos usar esta nueva característica para monitorizar el rendimiento de aplicaciones con más de un hilo de ejecución; o, también, podemos usarla para comparar la efectividad de varias implementaciones alternativas (secuenciales o paralelas) para un mismo problema. Esta capacidad era definitivamente necesaria para una herramienta de monitorización tan potente como PMCTrack. Ahora, por fin podemos afirmar que la herramienta es capaz de monitorizar cualquier tipo de programa que se ejecute en un microprocesador moderno con arquitectura ARM o x86.



(a) Antes del comienzo del TFG



(b) Después de la finalización del TFG

Figura 6.1: Arquitectura de PMCTrack

Además, con esta incorporación aprovechamos para cambiar el diseño que se usaba para guardar los datos de las muestras obtenidos con los contadores hardware. Tras este cambio, el diseño interno del módulo del kernel de PMCTrack ha mejorado sustancialmente en cuanto a claridad y robustez.

En segundo lugar, la nueva librería *libpmctrack* provee a los programadores de todas las capacidades de la monitorización mediante contadores hardware que ofrece PMCTrack. Con libpmctrack, un desarrollador puede evaluar y comparar las implementaciones de sus programas a partir de datos muy específicos del hardware, como puede ser los accesos a memoria caché o los fallos de predicción de saltos.
Asimismo, libpmctrack permite una monitorización de grano más fino que la que permite la herramienta de línea de comandos `pmctrack`: libpmctrack permite la monitorización individualizada de fragmentos de código seleccionados por el programador. Esto es especialmente útil para aislar partes del código cuyo análisis sea relevante por alguna razón como, por ejemplo, tratarse de un cuello de botella del programa.
Además, de nuevo la implementación de una nueva característica en el proyecto permitió la mejora del código existente en PMCTrack. Concretamente, libpmctrack sirvió para realizar una refactorización completa del programa `pmctrack`, dando lugar a un código mucho más sencillo y completamente desacoplado de la interfaz exportada por el módulo del kernel de PMCTrack.

En tercer lugar, la interfaz gráfica PMCTrack-GUI facilita en gran medida el uso de PMCTrack y, además, incorpora el soporte de gráficas que pueden resultar de gran interés para el usuario final.
El comando `pmctrack` requiere de la especificación de muchos detalles tremendamente tediosos y específicos de cada modelo de procesador como, por ejemplo: la identificación de la máquina, su número de contadores y tipo, el código hexadecimal de cada evento y su configuración, la creación de una métrica personalizada, como el *CPI* o el *cache rate*. En definitiva, el usuario es responsable de especificar todos los parámetros correctos y de bajo nivel para la monitorización mediante `pmctrack`. Aún siendo la temática de la monitorización por hardware ciertamente no apta para todo tipo de usuario, con PMCTrack-GUI esperamos haber resuelto, o, al menos, rebajado ese obstáculo y conseguir que un usuario, con ciertas nociones mínimas del funcionamiento del hardware en una computadora, 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 por hardware. PMCTrack-GUI también ha facilitado enormemente esta tarea, permitiendo no solo la creación de gráficas para la monitorización en tiempo real, sino también permitiendo la visualización de múltiples de ellas simultáneamente. Además esta herramienta ofrece la posibilidad de guardar los resultados obtenidos durante la monitorización y está equipada con opciones avanzadas para la personalización de las gráficas generadas (colores utilizados, anchura de línea, ...).

Finalmente, la herramienta además provee de soporte de monitorización remota de máquinas accesibles por SSH. Esta característica no fue planteada inicialmente, pero ha resultado de gran utilidad para nosotros durante el desarrollo del proyecto y creemos que, sin duda, lo será para una gran cantidad de usuarios de PMCTrack-GUI.

6.2. Valoración del TFG

Para la realización de este TFG hemos tenido que 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 libpmctrack; y otras en Python al más alto nivel de interfaz gráfica.

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, tanto en su parte en el espacio de usuario con `pmctrack` como en sus parte relacionada con la modificación del kernel y de los módulos para cada arquitectura.
- El lenguaje de programación Python.
- Los lenguajes de marcado XML y DTD, cómo realizar la lectura ficheros con este formato y cómo generarlos a partir de otros ficheros CSV.
- Diversas librerías externas de Python usadas en el desarrollo de la interfaz gráfica: wx, matplotlib,...

Para concluir, creemos que hemos hecho una importante aportación a la herramienta PMCTrack y esperamos que resulte de gran utilidad tanto dentro como fuera de nuestra universidad.

6.3. Trabajo futuro

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

- Soporte de PMCTrack en Android.
- Diseño de nuevos algoritmos de planificación en el kernel Linux que realicen optimizaciones en tiempo de ejecución en base en las medidas de los contadores que ofrece PMCTrack.

- Inclusión de una nueva opción en el comando `pmctrack` para permitir la monitorización de varias CPU al mismo tiempo y separar su salida.
- Soporte para otras arquitecturas hardware (más allá de x86 y ARM)
- Soporte para uso de contadores virtuales en PMCTrack-GUI.
- Capacidad para salvar configuraciones de contadores en ficheros para su carga posterior en PMCTrack-GUI.
- 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 PMCTrack-GUI ha solicitado mostrar gráficamente.
- Soporte de arquitecturas asimétricas o heterogéneas en PMCTrack-GUI.

Apéndice A

Introduction

A.1. PMCTrack: delivering hardware performance monitoring counter support

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 hints to programmers and computer architects. Notably, direct access to PMCs is typically restricted to code running at the OS privilege level. As such, a kernel-level tool, implemented in the OS itself or as a driver, is usually in charge of providing userspace tools with a high-level interface enabling to access performance counters [23, 10, 7].

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 multicore systems [11, 24, 15, 25, 12, 20, 18, 19]. While public-domain tools to access PMCs make it possible to monitor application performance from userspace, they do not provide an architecture-independent API empowering the OS itself to leverage PMC information to drive scheduling decisions. As a result, many researchers turned to architecture-specific *ad-hoc* code to access performance counters when implementing different scheduling schemes [11, 12, 20, 19]. This approach, however, leads the scheduler implementation to be tied to a certain architecture or processor model, and at the same time, forces developers to deal with (or even write themselves) the associated low-level routines to access PMCs on each architecture targeted by the scheduler. To avoid facing these issues, other researchers resorted to limited and simplistic userspace scheduling prototypes [24, 25, 18] that rely on existing userspace-oriented PMC tools.

In order to overcome these limitations, it was proposed the development of PMCTrack, a hardware-counter management tool for the Linux kernel. This tool was specifically designed to make it possible for the OS to use the counters for its internal tasks, such as process scheduling. This tool was originally developed as a project carried out by

students of the Computer Science School at the Complutense University of Madrid, back in 2012 [14]. Today, several members of the Architecture and Technology on Computing Systems Research Group (ArTeCS) of the Complutense University take on the development and support of PMCTrack. In fact, simultaneously to the development of our Degree Thesis, new features have been added to this tool beyond the extensions proposed in the DT.

PMCTrack's novelty lies in the *monitoring module* abstraction, an architecture-specific extension responsible for providing any OS scheduling algorithm that leverages PMC data with the performance metrics it requires to function. This abstraction makes it possible to implement architecture-independent OS scheduling algorithms. Specifically, ensuring that a thread scheduler works on a new processor model or architecture boils down to developing the associated platform-specific monitoring module in a loadable kernel module. More importantly, because PMCTrack offers an architecture-independent interface to easily configure events and gather PMC data, the monitoring module developer does not have to deal with the platform-specific low-level code to access PMCs on a given architecture, which greatly simplifies the implementation.

Despite being a tool specifically designed to aid the OS scheduler, PMCTrack is also equipped with a set of command line tools and userspace components. These userspace tools assist OS-scheduler designers during the entire development life cycle, by complementing the existing kernel-level debugging tools with PMC-related offline 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*.

A.2. Alternatives to PMCTrack

Several tools have been created for the Linux kernel in the last few years [7, 10, 17, 5, 22, 21], enabling to hide the diversity of the various hardware interfaces to end users and providing them with convenient access to PMCs from userspace. Overall, these tools can be divided into two broad categories. The first group encompasses tools such as OProfile [7], perfmon2 [10] or perf [17], which expose performance counters to the user via a reduced set of command-line tools. These tools do not require to modify the source code of the application being monitored; instead they act as external processes with the ability to receive PMC data of another application. The second group of tools provides the user with libraries to access counters from an application's source code, thus constituting a fine-grained interface to PMCs. The libpfm [10] and PAPI [5] libraries follow this approach.

The perf [17] tool, which relies on the Linux kernel's *Perf Events* [23] subsystem, is possibly the most comprehensive tool on the first category available at the time of this writing. Although perf began as a PMC tool supporting a wide range of processor architectures, it now empowers users with striking software tracing capabilities enabling them to keep track of a process' system calls or scheduler-related activity, or various

network/file-related operations executed on behalf of an application. Despite the potential of perf and the other aforementioned tools, neither of them implement a kernel-level architecture-agnostic mechanism enabling the OS scheduler to leverage PMC data for its internal decisions. PMCTrack has been specifically designed to fill this gap. As PMCTrack, perf has also the capability to expose non-PMC hardware-related data exposed by modern hardware to the user, such as the LLC occupancy.

Despite the potential of perf and other related tools, none of them implements a mechanism that provides a kernel level architecture-independent interface that allows the OS scheduler leverage information from the PMCs for its internal decisions. This is the main purpose of PMCTrack.

As PMCTrack, some performance monitoring tools require changes to the Linux kernel to provide the desired functionality [10, 1]. KerMon [1] relies on a separate scheduling class in the kernel to carry out low-level access to PMCs. To collect PMC-data for an application via KerMon, the application must be scheduled with the new scheduling class. This scheduling class is merely a clone of the fair (CFS) class so it does not exploit PMC values to make scheduling decisions. PMCTrack, on the other hand, makes it possible for virtually any scheduling class created in the kernel to gather performance metrics via an architecture-independent mechanism. To this end, the kernel requires minimal changes, since as we show in the next section, the vast majority of PMCTrack functionality is encapsulated in a loadable kernel module.

A.3. PMCTrack design

This section describes the internal architecture of PMCTrack, just as it was before our DT started. We also present the different use modes originally supported by the tool.

A.3.1. Architecture

Figure A.1 depicts PMCTrack's internal architecture. The tool consists of a set of user and kernel space components. At a high level, the end user and the applications interact with PMCTrack using the available command line tools. These components communicate with PMCTrack's kernel module by means of a set of Linux `/proc` entries 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 application's performance counter data to the user-land tools, the module implements a simple API to feed with per-thread monitoring data to any scheduling policy (class) that requires performance-counter information to function. Because both the core Linux Scheduler and scheduling classes are implemented entirely in the kernel, making

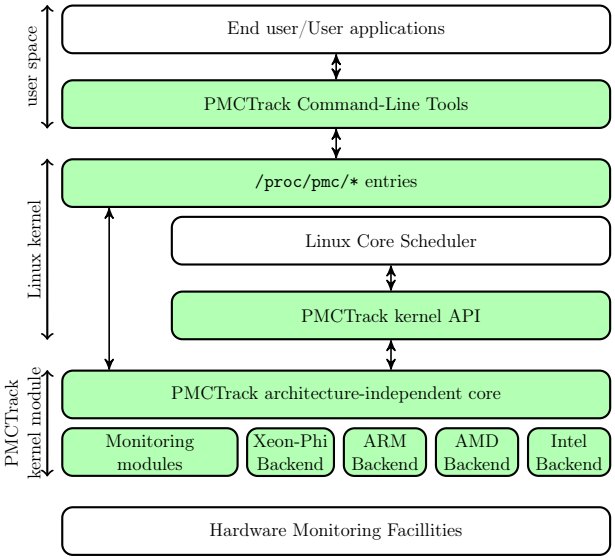


Figure A.1: Architecture of PMCTrack

PMCTrack's kernel module aware of these events and requests requires some minor modifications to the Linux kernel itself. These modifications, referred to as PMCTrack kernel API 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 following interface:

```
typedef struct pmc_ops{
  /* invoked when a new thread is created */
  void* (*pmcs_alloc_per_thread_data)(unsigned long,struct task_struct
      *);
  /* invoked when thread leaves the CPU */
  void (*pmcs_save_callback)(void*, int);
  /* invoked when thread enters the CPU */
  void (*pmcs_restore_callback)(void*, int);
  /* invoked every clock tick on a per-thread basis */
  void (*pmcs_tbs_tick)(void*, int);
  /* invoked when a process invokes exec() */
  void (*pmcs_exec_thread)(struct task_struct*);
  /* invoked when a thread exists the system */
  void (*pmcs_exit_thread)(struct task_struct*);
  /* invoked when a thread descriptor is freed up */
  void(*pmcs_free_per_thread_data)(struct task_struct*);
  /* invoked when the scheduler requests per-thread
        monitoring information  */
  int  (*pmcs_get_current_metric_value)(struct task_struct* task, int
      key, uint64_t* value);
} pmc_ops_t;
```

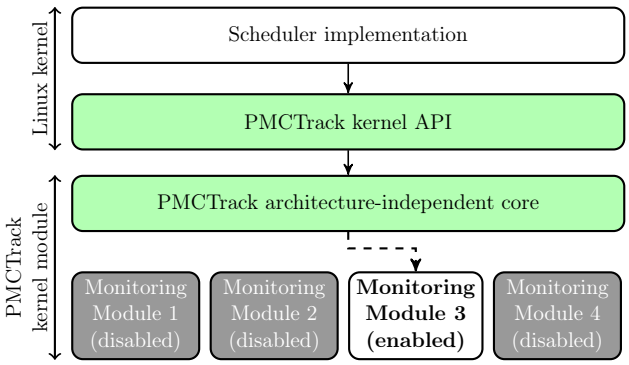


Figure A.2: PMCTrack monitoring modules

Most of these notifications get engaged only when PMCTrack's kernel module is loaded and the user (or the scheduler itself) is using the tool to monitor the performance of a specific application.

As shown in Figure A.1, PMCTrack's kernel module consists of various components. The architecture-independent core layer implements `pmc_ops_t` interface and interacts with PMCTrack command line tools via the Linux proc file system. The architecture-independent component relies on a Performance Monitoring Unit Backend (PMU BE) to carry out low-level access to performance counters, as well as for translating user-provided counter configuration strings into internal data structures for the platform in question. Currently there are backends compatible with most modern processors from Intel and AMD. Also, simultaneously to the development of our DT, the development of two extra backends was carried out, one compatible with ARM's Cortex processors of 32 and 64 bits, and another for Intel's Xeon Phi coprocessor. PMCTrack's kernel module also includes a set of platform-specific *monitoring modules*. The primary purpose of a monitoring module is to provide a scheduling algorithm implemented in the kernel with high-level performance metrics.

Augmenting a recent version of the Linux kernel (2.6.38 and above) to support PMCTrack entails including two new source files to the kernel tree (API for the module), and adding less than 20 extra lines of code. These changes can be easily applied to different kernel versions.

A.3.2. PMCTrack Usage Models

Before the beginning of our project, PMCTrack supported three usage models: scheduler mode, time-based sampling and event-based sampling.

A.3.2.1. Scheduler mode

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 microarchitectural properties. Turning on this mode for a particular thread from the scheduler's code boils down to activating the `prof_enabled` flag[1] in the thread's descriptor.

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 configure nor deals with performance counters directly. Instead, one of PMCTrack's *monitoring modules* is in charge of feeding the scheduling policy with the necessary high-level performance monitoring metrics, such as a task's instruction per cycle ratio or its last-level cache miss rate.

As shown in Figure A.2, PMCTrack may include several monitoring modules 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 several compatible monitoring modules are available, the system administrator may tell the system which one to use by writing in the `/proc/pmc/mmon_manager` file. In a similar vein, the PMC sampling period used by the monitoring module can be configured via the */proc* file system.

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, 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 to collect 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 interface very similar to `pmc_ops_t`. Specifically, it consists of several callback functions enabling 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.

Creating new monitoring modules is a rather simple task for several reasons. First, the programmer does not need to access performance-counter registers directly. Instead, PMCTrack's kernel module offers an architecture-independent API enabling the monitoring module to specify the counter configuration via strings, to receive performance

[1] This flag is added to Linux `task_struct` when applying PMCTrack's kernel patch.

counter samples periodically and to control event multiplexing. Second, because the module gets notified when a new thread is created or exits the system, the monitoring module may allocate per-thread data to simplify any kind of thread-specific processing. Third, because a monitoring module's code lives in PMCTrack's kernel module, fixing most bugs does not require rebooting the system; instead the kernel module can be unloaded/reloaded after applying the bug fix.

A.3.2.2. Time-based sampling (TBS)

This feature allows the user to gather an application's performance data from userspace at regular time intervals. The `pmctrack` command-line tool, inspired in Solaris's `cputrack` program, enables to use this feature. To illustrate how the tool works let us consider the following example:

```
$ pmctrack -T 1 -c pmc0,pmc3=0x2e,umask3=0x41 ./mcf06
nsample  event          pmc0          pmc3
      1   tick    1961001132        110634
      2   tick    1247853112          8323
      3   tick    1230836405          3859
      4   tick    1358134323        409386
      5   tick    1280630906       1199270
      6   tick    1231578609      15488307
...
```

In a system with a modern Intel processor, this command provides the user with the number of instructions retired and last-level cache (LLC) misses every second. Each sample is represented by a different row. The sampling period can be specified in seconds via the -T option; fractions of a second can be also specified (e.g, 0.3 for 300ms). If the user includes the -A switch in the command line, `pmctrack` displays the aggregate event counts for the application's entire execution instead. At the end of the line, we specify the command to run the associated application we wish to monitor (e.g: ./mcf06).

The -c option accepts as an argument a raw PMCTrack configuration string which follows the internal event configuration format recognized by PMCTrack kernel module. The raw format gives flexibility to experienced users enabling them to decide the event-to-counter mapping or the actual hex code that is written in the low-level PMC register fields exposed by the kernel module. As we have seen, the raw string `pmc0,pmc3=0x2e,umask3=0x41` would also enable to gather the aforementioned event counts on most modern Intel processors. On processors from the ARM Cortex Ax family, this event set can be represented with the `pmc1=0x8,pmc2=0x17` raw string. If the user does not know the hexadecimal codes that allow to associate an event with a PMC, he should consult those codes in the architecture's manual in question.

A striking feature of the time-based sampling usage model is the ability to retrieve virtually any run time metric gathered by the active monitoring module such as energy consumption, the application's LLC occupancy or any high-level metric determined by the module. To this end, the monitoring module must explicitly expose the metrics as *virtual counters* using PMCTrack's API. Virtual counter values can be retrieved using the -V option of the `pmctrack` program.

In case a specific processor model does not integrate enough PMCs to monitor a given set of events at once, the user can turn to PMCTrack's event-multiplexing feature. This boils down to specifying several event sets by including multiple instances of the -c switch in the command line. In this case, the various events sets will be collected in a round-robin fashion and a new *expid* field in the output will indicate the event set a particular sample belongs to.

Note that in the event the active monitoring module is currently using PMCs on behalf of a scheduling algorithm, the user is not allowed to specify a PMC configuration string via the `pmctrack`'s -c option. Nevertheless, for debugging purposes the pmctrack command may still be used in this case (without the -c switch), to retrieve the event counts associated with the configuration imposed by the monitoring module.

To support the time-based sampling feature, the PMCTrack's kernel module stores performance and virtual counter values in a per-application ring buffer. The command-line tool retrieves samples from the kernel buffer by reading from a /proc file that blocks the monitor process till new samples are generated or the application terminates.

A.3.2.3. Event-based sampling (EBS)

Event-based Sampling (EBS) constitutes a variant of time-based sampling wherein PMC values are gathered when a certain event count reaches a certain threshold (T). To support EBS, PMCTrack's kernel module exploits the interrupt-on-overflow feature present in most modern Performance Monitoring Units (PMUs). At a high level, when EBS is turned on, PMCTrack's kernel module initializes the associated performance counter to $-T$; when this counter overflows the PMUs generates an interrupt, upon which the kernel module reads all the counters.

To use the EBS feature from userspace, the *ebs* flag must be specified in `pmctrack`'s command line at the end of the configuration counters string. In doing so, a threshold value may be also specified as in the following example:

```
$ pmctrack -c pmc0,pmc3=0x2e,umask3=0x41,ebs0=500000000 ./mcf06
nsample  event          pmc0          pmc3
      1    ebs     500000087         10677
      2    ebs     500000002         22336
      3    ebs     500000004         17131
      4    ebs     500000007         12995
      5    ebs     500000014          9348
      6    ebs     500000010          5804
...
```

The *pmc3* column displays the number of LLC misses every 500 million retired instructions. Note, however, that values in the *pmc0* column do not reflect exactly the target instruction count. This has to do with the fact that, in modern processors, the PMU interrupt is not served right after the counter overflows. Instead, due to the out-of-order and speculative execution, several dozen instructions or more may be executed within the period elapsed from counter overflow until the application is actually interrupted. These inaccuracies do not pose a big problem as long as coarse instruction windows are used.

A.4. Project goals

The PMCTrack tool provides great features and empowers the OS scheduler to take advantage of hardware counters to make optimizations at run time. However, PMCTrack still has important limitations that cast a shadow over its great possibilities, especially when it comes to monitoring application performance from userspace.

First, its use turns out a bit complicated for inexperienced users, since they have to have access to technical manuals of different architectures to look up the appropriate hex codes for the events they want to monitor with the various PMCs.

Second, getting the specific events count on the PMCs on a timely basis does not provide the user the big picture on the evolution over time of the values for the different counters, let alone high level metrics made up by the combination of two or more PMC values. This valuable information for the user can be provided by means of charts. Nevertheless, this entails to go through various steps: (1) processing the data provided by the `pmctrack` command to get the high level metrics values over time, and (2) using some helper utility, such as *Gnuplot* to generate the final charts. Finally, another significant limitation of the tool is the fact that the `pmctrack` command (and the kernel module itself on which it relies) does not allow monitoring multithreaded applications from userspace.

To address these and other related problems, this project pursues the following goals:

1. Provide support for monitoring multithreaded applications from userspace with PMCTrack.

2. Design and implement a graphical frontend for PMCTrack (referred to as PMCTrack-GUI) that enables real-time visualization of high-level performance metrics defined by the user.

3. Create *libpmctrack*, a library that makes it possible to monitor code fragments of application programs by means of PMCs.

A.5. Work plan

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

1. Divide work into individual tasks for the various members of the work group
2. Reading documentation about programming languages and other technologies used to carry out the development (Python, wxPython, matplotlib, ...). This step also entails getting familiar with PMCTrack internal architecture.
3. Making PMCTrack-GUI mockups to study different design alternatives and help reaching an agreement on the final design.
4. Augmenting PMCTrack with support for monitoring multithreaded applications from userspace.

5. Implementation of PMCTrack-GUI.
6. Design and implementation of *libpmctrack* and refactoring the code of the `pmctrack` command-line tool.
7. Developing the various case studies to test the different extensions incorporated into PMCTrack.

It is worth noting that the order of these steps is merely illustrative, since such steps were completed in a strictly sequential way. Specifically, the development of the various components of PMCTrack required crafting a separate schedule, since we implemented different components simultaneously. In addition, changes in PMCTrack were made during our project due to maintenance issues (PMCTrack is a tool in continuous development), and new features not initially planned were implemented at the end, such as the inclusion of an SSH mode for PMCTrack-GUI.

Apéndice B

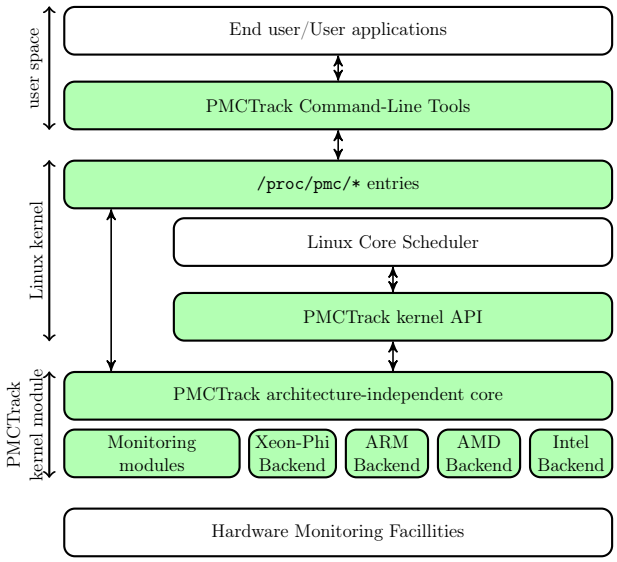
Conclusions and future work

B.1. Conclusions

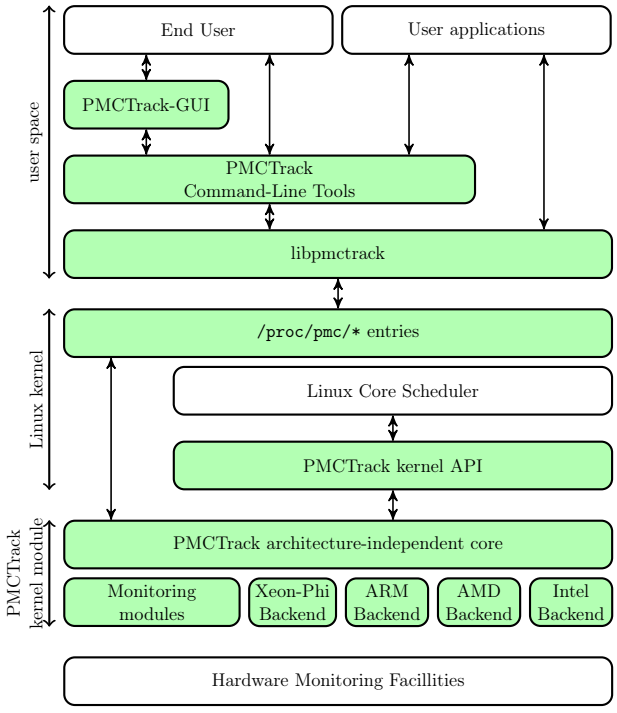
After the completion of this project, PMCTrack has got three new features: (1) support for multithreaded applications, (2) the libpmctrack library –which allows to analyze particular fragments of code through PMCs–, and (3) the frontend interface PMCTrack-GUI –a Graphical User Interface which eases its use and produces real-time graphs of user-defined high level metrics–. In chapters 2, 3 and 4, we have discussed in detail the inner working, the design process and the implementation for each of these features. In chapter 5, we have tried to illustrate their capabilities and effectiveness by analyzing some case studies. Lastly, we will study how has been impact of our project, as a whole, on PMCTrack.

In the figure B.1 we can appreciate two diagrams representing the before and after situation of PMCTrack's architecture. We can see how the changes have affected to every level of PMCTrack: from the kernel level –modifying and redesigning the kernel module to allow get samples from several threads using a shared buffer–, through the userspace level with libpmctrack, to the uppermost level with PMCTrack-GUI –where we have worked in the graphical interface level–.

First, multithreading support has extended significantly the PMCTrack functionality. We can use this new feature to profile multithreaded applications; or, also, we can use it to see the outcome of paralleling algorithms, comparing the performance of sequential implementations versus concurrent implementations. This feature was definitely needed to have, because it makes PMCTrack capable to monitor any type of program executing on a modern x86 or ARM microprocessor.
In addition, with this new feature we were able to redesign the process of saving the samples data obtained from the PMCs. Thank to this change, the internal design of the PMCTrack's kernel module has improved substantially in clarity and sturdiness.



(a) Before the start of this project



(b) After the completion of this project

Figure B.1: Architecture of PMCTrack before and after this project

Second, the new library *libpmctrack* gives to the programmers all the possibilities of profiling provided by PMCTrack. Now, a developer can use libpmctrack to evaluate hers different programs implementations, using very hardware-specific data as Last Level Cache Accesses or Mispredicted Branch Instructions.
The tool `pmctrack` sometimes can get too general data of the program and the developer can be rather more interested in specific parts, for example looking for a bottle neck in hers program. Our new library, libpmctrack, provides to the programmer of a easy way for analyzing particular fragments of code separately.
Again the implementation of a new feature introduced an improvement in the preexisting code of PMCTrack. Specifically, we completely refactored the code of `pmctrack` command-line tool, drawing upon libpmctrack for the communications between it and the kernel module. This leads a much simpler code and entirely decoupled from the interface exported by the PMCTrack's kernel module.

Third, the Graphical User Interface (GUI) PMCTrack GUI allows a straightforward use of the tool PMCTrack and, also, support the generation of real-time custom graphs. The command-line tool `pmctrack` requires of the specification of many details, for example: the number of PMCs and type, the hexadecimal code for each event, the creation of custom metrics like CPI or cache rate. Also, all these details are also highly dependent on the model of the machine wanted to be monitored. PMCTrack-GUI accomplishes the task of easing this process, it abstracts to the user of all these particular considerations which depends on different models and gets the user directly to hers main goal: profile applications through the profiling monitoring counters. Nevertheless, this topic is still restricted to users with a minimum knowledge of the computer internal architecture and its way of working.
Additionally, often the user wants to get graphs to see the profiling outcomes. PMCTrack-GUI fills this gap generating directly real-time graphs based on the metrics configuration made by the user. Furthermore, PMCTrack-GUI allows to widely customize the displaying options of the graphs, supports the visualization of multiples graphs simultaneously and gives an easy way to save the results achieved during the profiling process.
Finally, this graphical tool provides of the ability to monitor remote machines through SSH. This feature was not in the initial plans, but it turned out to be very useful for us upon the development of the project and we think that many PMCTrack-GUI users will find it useful too.

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, and some other times in Python 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.
- The structure and code of Linux kernel.
- The internal architecture of PMCTrack, both the userspace part, upon `pmctrack`, and the kernel space part, upon the kernel modifications and kernel modules.
- The Python programming language.
- The Markup languages: XML and DTD, how to read files of these formats and how to generate them from other DTD files.
- Several external Python libraries used on the development of PMCTrack-GUI: wx, matplotlib,...

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:

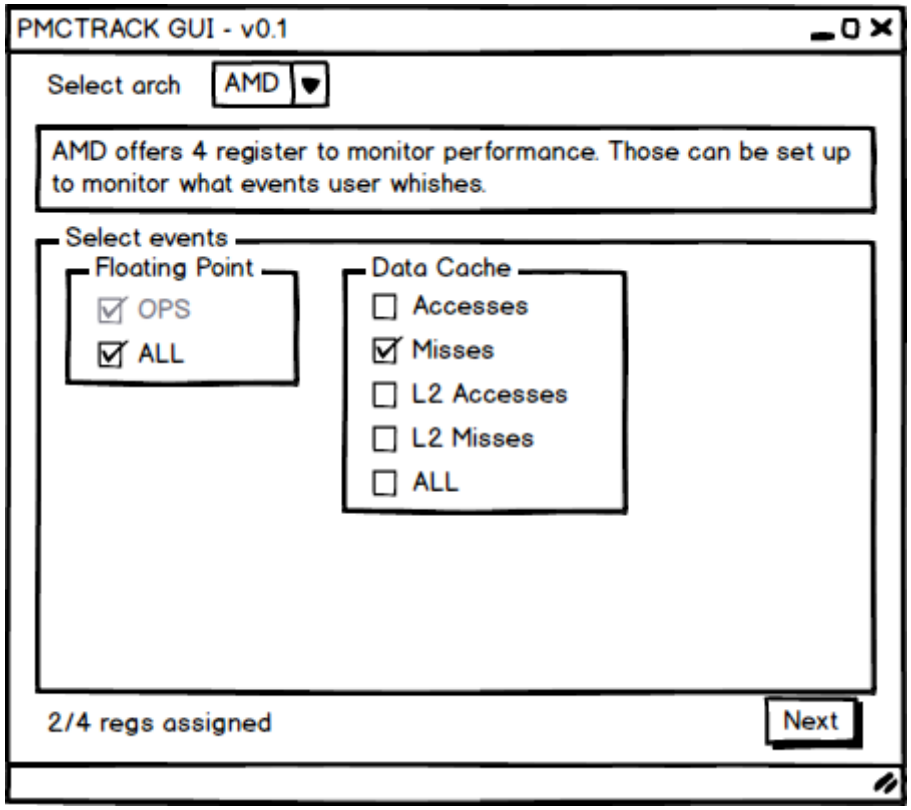
- To add support to PMCTrack for Android.
- To design of new scheduling algorithms in the Linux kernel that perform optimizations in run time based on the data obtained from the PMCs through PMCTrack.
- To extend the command `pmctrack` with a time-based sampling per-CPU mode.
- To add support for additional CPU architectures.
- To add support for virtual counters in PMCTrack-GUI.
- To allow saving and loading configurations for PMCTrack-GUI.
- To allow saving the output for the custom metrics defined by the user in PMCTrack-GUI. Right now, it saves the standard output of `pmctrack` only.
- To support asymmetric architectures in PMCTrack-GUI.

Apéndice C

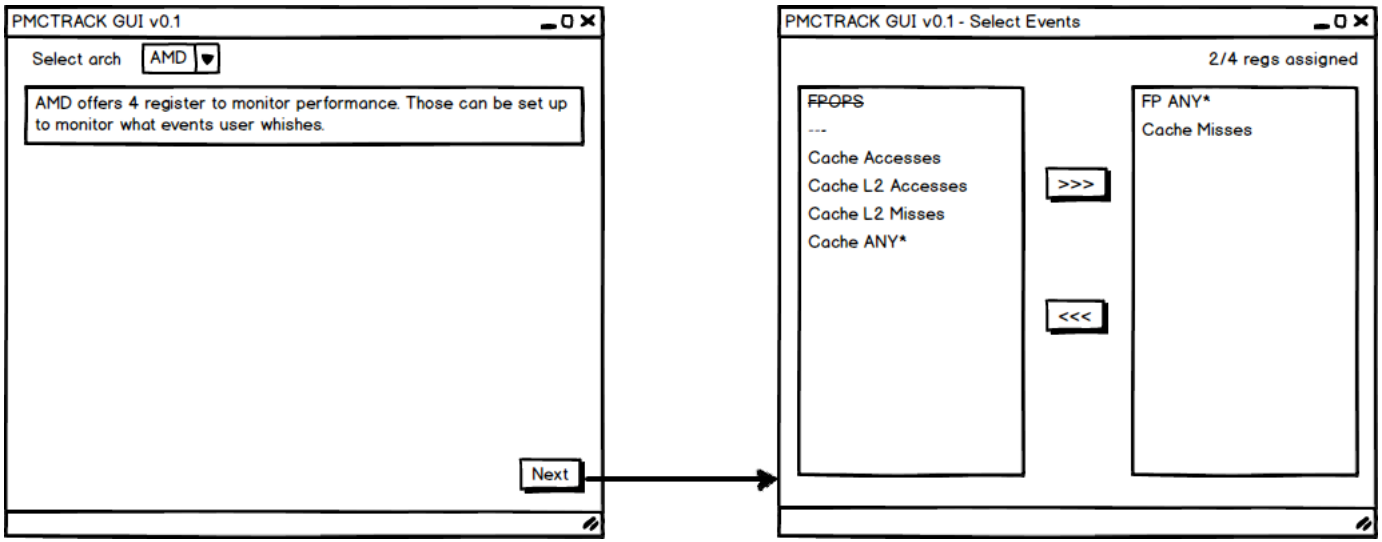
Bocetos para la interfaz gráfica PMCTrack-GUI

En este apéndice incluimos algunos de los bocetos o *mockups* que realizamos para decidir cómo debía ser el diseño de la interfaz gráfica de PMCTrack-GUI.

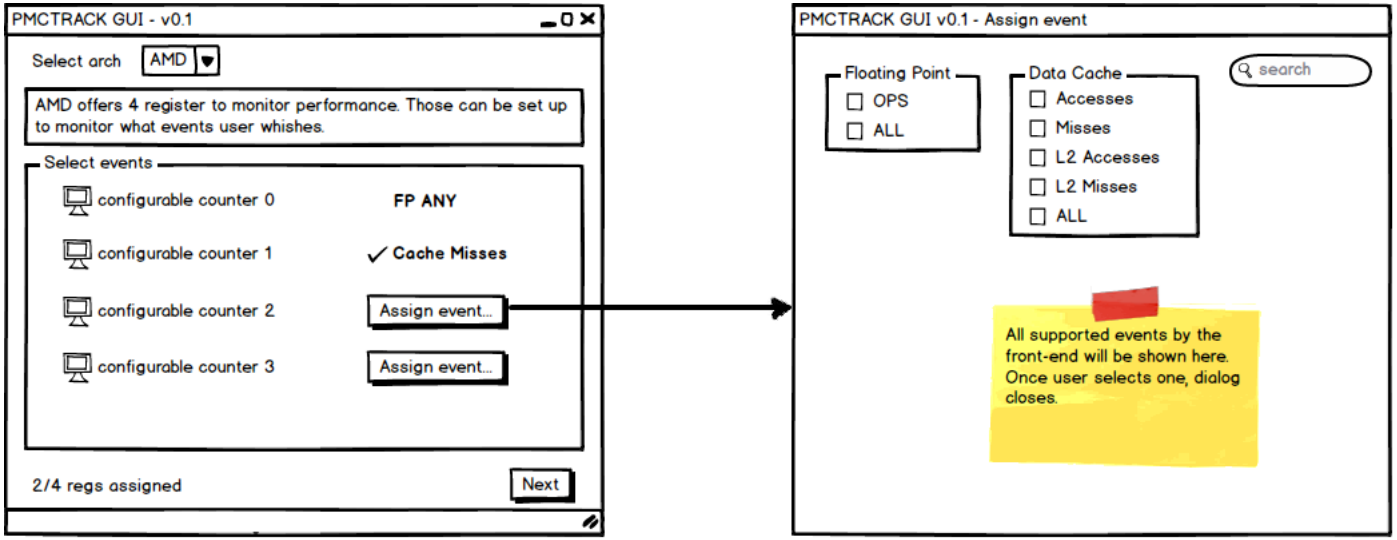
C.1. Boceto temprano de la ventana principal



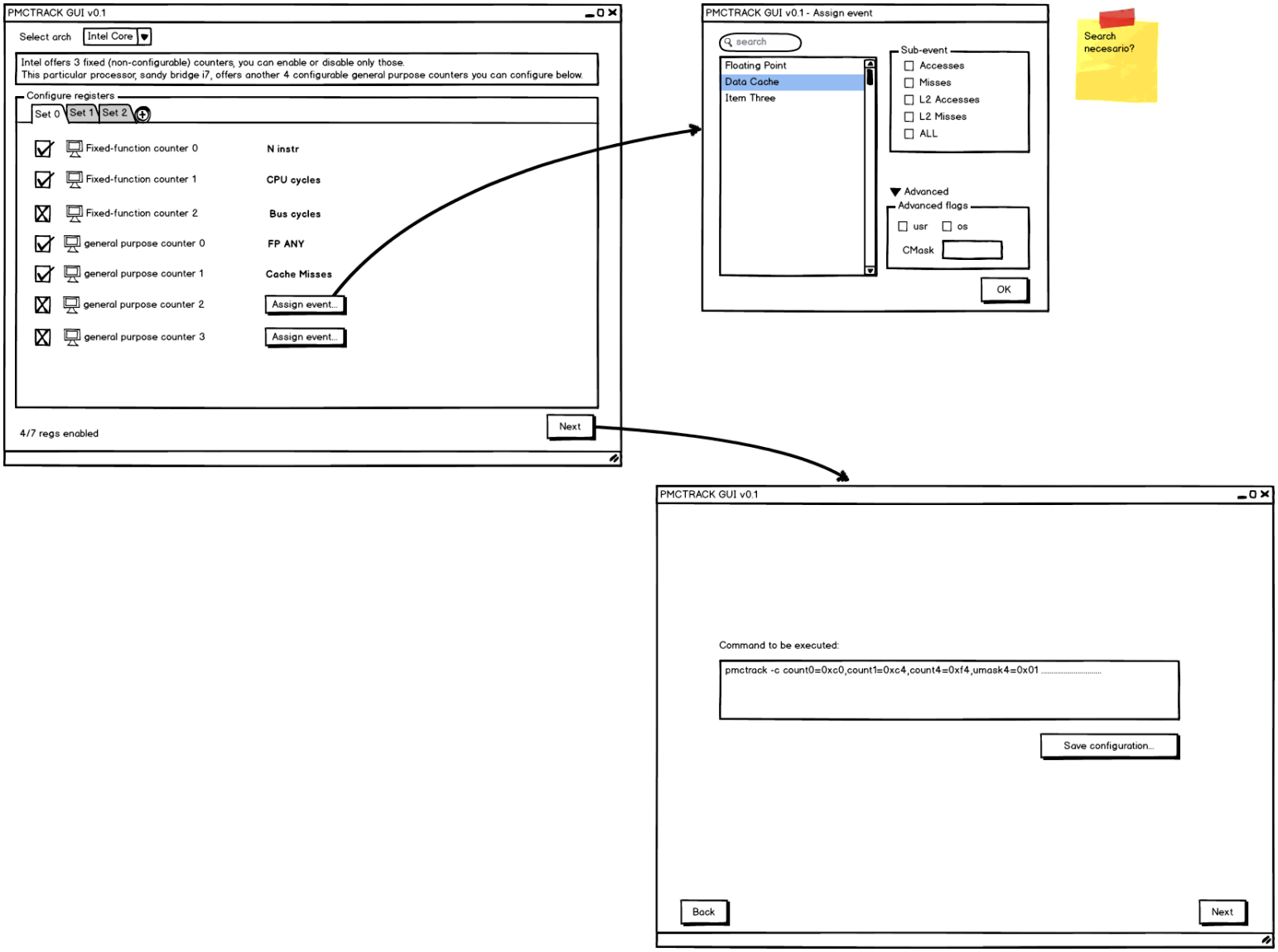
C.2. Segunda iteración de bocetos



C.3. Boceto más avanzado



C.4. Última iteración de bocetos

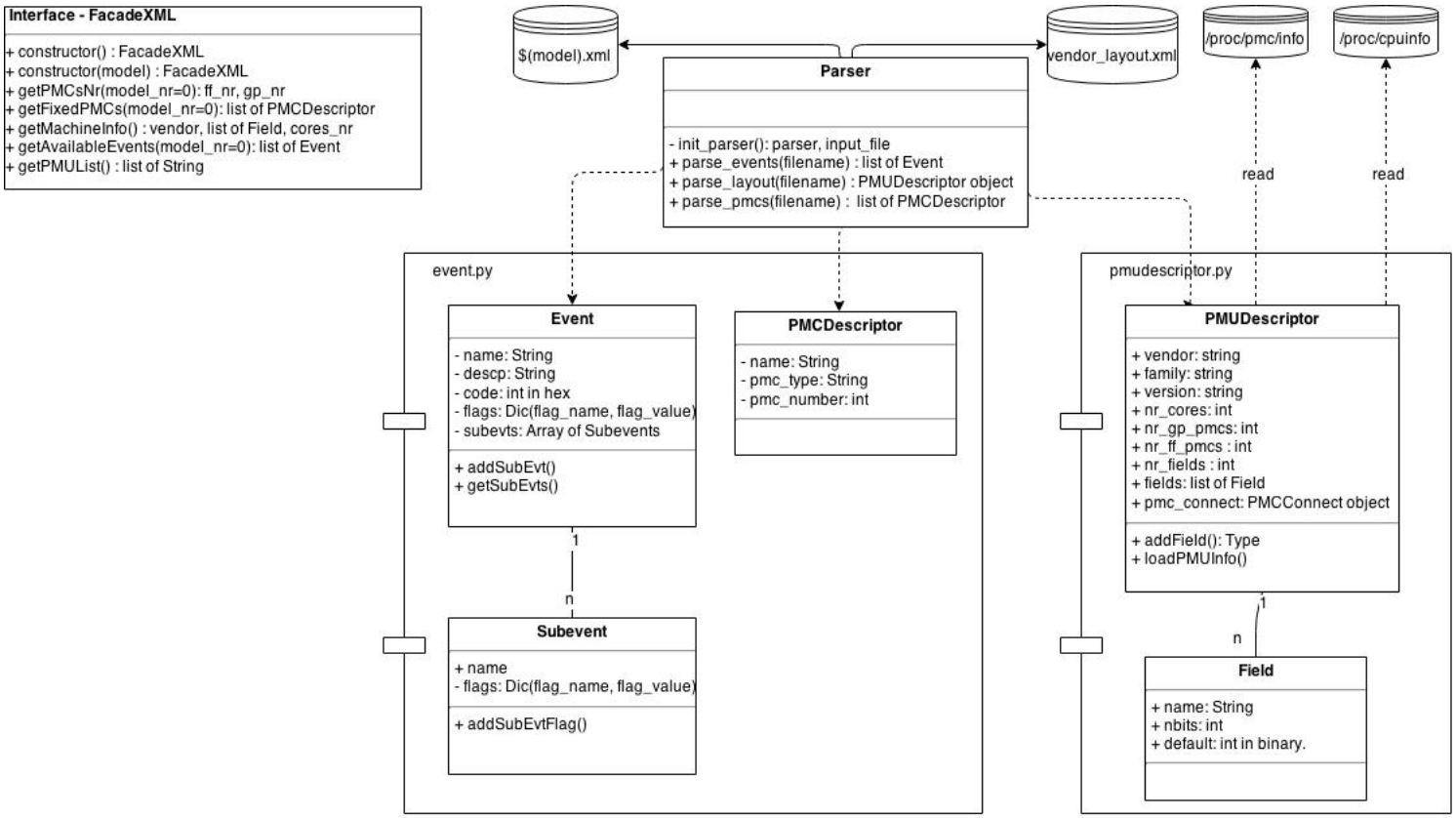


Apéndice D

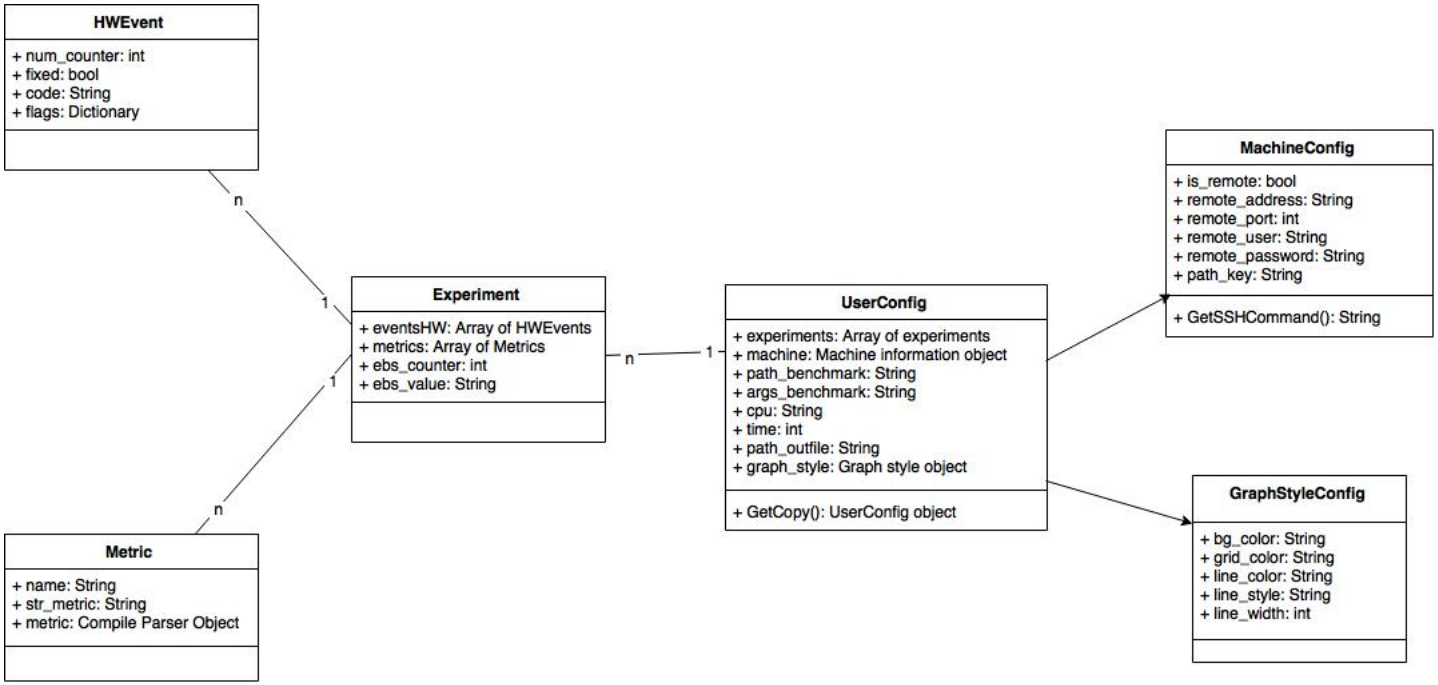
Diagramas del diseño de PMCTrack-GUI

En este apéndice incluimos los diagramas que hemos usado durante el diseño de nuestra aplicación PMCTrack-GUI, y que sin duda, servirán al lector para un mayor entendimiento de la aplicación.

D.1. Diagrama UML: Objetos de procesamiento



D.2. Diagrama UML: Objetos de configuración de usuario



Apéndice E

Dependencias software de PMCTrack-GUI

En el presente apéndice se adjunta el listado de dependencias software de PMCTrack-GUI para que se puedan utilizar todas sus funcionalidades. Adicionalmente, se incluye una pequeña guía de instalación de todas estas dependencias para los sistemas operativos Debian/Ubuntu y MacOS X.

PMCTrack-GUI cuenta con un total de cuatro dependencias software que enumeramos a continuación:

- Python v2.7
- Matplotlib
- Comando sshpass
- WxPython v2.8 o superior.

E.1. Instalación de las dependencias software en MacOS X

Para la instalación de los requisitos software de PMCTrack-GUI en el sistema operativo MacOS X, basta con ejecutar los siguientes comandos en una terminal.

```
$ sudo port upgrade outdated
$ sudo port install py27-matplotlib py27-numpy py27-scipy py27-ipython
    py27-wxpython-2.8 sshpass
$ sudo port install
$ mkdir  ~/.matplotlib
$ cp /opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/
    python2.7/site-packages/matplotlib/mpl-data/matplotlibrc  ~/.
    matplotlib
$ sudo port select --set python python27
$ sudo port select --set ipython ipython27
$ which python
```

E.2. Instalación de las dependencias software en Debian/Ubuntu

Para la instalación de los requisitos software de PMCTrack-GUI en los sistemas operativos basados en Debian, basta con ejecutar los siguientes comandos en una terminal.

```
$ sudo apt-get update
$ sudo apt-get install python-wxgtk2.8 python-matplotlib sshpass
```

Apéndice F

Contribuciones de cada participante

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

F.1. Contribución de Jorge Casas Hernán

La mayoría de mi contribución a este proyecto ha estado dirigida a la aplicación PMCTrack-GUI, de la cual he implementado desde el sistema de ventanas con el cual interactúa el usuario hasta la generación de las gráficas en tiempo real con el uso de la librería *matplotlib*, pasando por el diseño e implementación de componentes internos de la aplicación. No obstante, también he colaborado en la realización de la extensión de PMCTrack para el soporte de aplicaciones multihilo desde el espacio de usuario, extensión que hemos desarrollado de forma conjunta mi compañero de proyecto y yo. En el resto de líneas de esta sección relato de manera detallada mis contribuciones al proyecto por orden cronológico.

La primera tarea productiva que realicé en este proyecto fue el estudio de documentación acerca del lenguaje Python, lenguaje que elegimos para el desarrollo de PMCTrack-GUI, por los motivos explicados en la sección 4.4. Esta tarea de estudio de documentación comprendió no solo el propio lenguaje de programación básico, si no también las librerías *wxpython* y *matplotlib* que serían usadas ampliamente en la implementación de la aplicación. A lo largo de esta tarea realicé pequeñas aproximaciones a funcionalidades concretas con las que tenía que contar PMCTrack-GUI, para familiarizarme con el lenguaje.

Una vez entendido el funcionamiento básico de Python y sus librerías, comencé a desarrollar la estructura básica de las ventanas de configuración del usuario, basadas en los bocetos que habíamos realizado previamente, usando para ello una herramienta llamada *wxglade*. A medida que implementaba cada una de estas ventanas iba conociendo más en profundidad la librería WX y el lenguaje Python en general. Esto me permitió reimplementar las ventanas realizando una serie de optimizaciones que mejoraron ampliamente el aspecto final de la aplicación.

Simultáneamente a la finalización del desarrollo del aspecto visual de las ventanas, mi compañero de proyecto terminó el desarrollo principal de los Objetos de procesamiento, de modo que comencé a adaptar la estructura básica de cada ventana para que sus componentes gráficos se generaran en función de los datos proveídos por los Objetos de procesamiento. Junto a esta tarea también comencé a desarrollar los manejadores de los eventos producidos por el usuario al interactuar con las ventanas.

Posteriormente realicé el modelo de objetos de configuración de usuario, de forma que las ventanas usaran estos objetos para almacenar la configuración que iba realizando el usuario. Finalizada esta tarea ya estaba en condiciones de iniciar el desarrollo de la generación de las gráficas de rendimiento en tiempo real. No obstante, llegados a este punto mi compañero del proyecto y yo comenzamos a desarrollar la extensión de PMCTrack para el soporte de aplicaciones multihilo.

Esta extensión, que fue desarrollada de forma conjunta entre los dos, comenzó con un estudio detallado sobre la implementación de la herramienta PMCTrack, dando lugar a un conjunto de reuniones donde discutiríamos acerca del nuevo diseño adecuado de la herramienta para esta nueva característica. Posteriormente, mi compañero y yo realizamos diversas sesiones de trabajo conjuntas hasta la implementación completa de la extensión. Esta parte del proyecto finalizó con un conjunto de pruebas usando benchmarks multihilo de la suite PARSEC que verificó el funcionamiento estable de la extensión.

Simultáneamente al desarrollo de esta extensión, proseguí diseñando e implementando la parte de generación de gráficas de PMCTrack-GUI. Para ello creé el componente interno PMCExtract, encargado de la obtención de datos de la herramienta de línea de comandos `pmctrack`; y del canvas de la librería *matplotlib* que, usando esos datos proporcionados por PMCExtract, se encargaría de generar las distintas gráficas solicitadas por el usuario. Esta tarea duró más tiempo del esperado debido a la multitud de problemas de rendimiento y *bugs* con los que contaba inicialmente la aplicación.

Posteriormente me centré en la realización del modo SSH para la monitorización de máquinas remotas. Para modificar lo menos posible el código ya escrito, desarrollé el componente interno PMCConnect, encargado de realizar y devolver las lecturas de datos en la máquina local o remota solicitadas por los distintos componentes de la aplicación. Esto supuso realizar pequeñas modificaciones en estos componentes para que hicieran uso de PMCConnect, incluso en los Objetos de procesamiento realizados por mi compañero de proyecto, aunque esto no supuso ningún problema gracias al buen diseño de este componente.

La siguiente etapa consistió en el desarrollo del resto de funcionalidades de PMCTrack-GUI: realización del soporte multilenguaje usando el módulo *gettext* de Python, traduciendo la aplicación al inglés y español (mi compañero me ayudó en las traducciones al inglés); personalización del aspecto visual de las gráficas de rendimiento por parte del usuario; posibilidad de parar y reanudar el benchmark que se está monitorizando; etcétera.

Finalmente, pusimos a prueba la aplicación PMCTrack-GUI mediante la realización de casos de estudio (véanse las secciones 5.1 y 5.2) que nos ayudaron a detectar numerosos *bugs* que fui corrigiendo hasta llegar a una versión estable de la aplicación.

F.2. Contribución de Abel Serrano Juste

De entre los dos miembros del grupo, yo he sido el encargado de especializarme en la parte de más bajo nivel del proyecto. En particular, esto comprendía, entre otros, la lectura de documentación sobre contadores hardware, el estudio profundo de la herramienta PMCTrack, el desarrollo y refactorización del código con libpmctrack, el soporte de los eventos de bajo nivel para la interfaz gráfica,...

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

1. Estudio de documentación
2. Participación en el diseño de PMCTrack-GUI
3. Implementación de los objetos de procesamiento para PMCTrack-GUI
4. Implementación conjunta del soporte *multithreading* en PMCTrack
5. Diseño e implementación de libpmtrack. Refactorización de `pmctrack`
6. Realización de los benchmarks para el caso de estudio de libpmctrack

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.

F.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, del año 2012, dónde se introducía por primera vez la herramienta PMCTrack. A continuación, leí más documentación acerca de herramientas que cumplían funcionalidades parecidas a las de PMCTrack, como OProfile y perf. Adicionalmente, forma parte de esta primera fase de documentación, la lectura de los diversos manuales para desarrolladores de los distintos fabricantes de microprocesadores: AMD, Intel y ARM. En estos manuales, se presenta información muy detallada y particular de cómo usar los PMC para cada fabricante, junto con los códigos y la descripción de cada uno de los eventos que éstos pueden monitorizar.

En lo que podríamos llamar una segunda fase del estudio de documentación, me dediqué al aprendizaje del lenguaje de programación Python. Para ello, me leí un tutorial de Python 2.7 y realicé un pequeño curso online gratuito facilitado por Google. Esta fase se corresponde con la implementación de los objetos de procesamiento para PMCTrack-GUI, apartado F.2.3 de este apéndice.

Por último, la última fase de mi estudio de documentación se corresponde con las contribuciones de más bajo nivel: "Implementación conjunta del soporte *multithreading*" y "Diseño e implementación de libpmtrack", apartados F.2.4 y F.2.5 respectivamente.
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`.

F.2.2. Participación en el diseño de PMCTrack-GUI

En la etapa de realización de bocetos de la interfaz para PMCTrack-GUI, realicé varios de los bocetos sobre los que discutimos para llegar al diseño final de la interfaz. Dichos bocetos los realicé mediante la herramienta *Balsamiq Mockups*, cuyo uso había aprendido en la asignatura "Diseño de Sistemas Interactivos" en el primer semestre de este curso. Los bocetos fueron al principio de baja fidelidad, y, fueron aumentando su nivel de fidelidad y detalle según fuimos discutiendo e iterando en el proceso de diseño.
Por supuesto, también participé en las reuniones que tuvimos para el debate de estos bocetos.

F.2.3. Implementación de los objetos de procesamiento para PMCTrack-GUI

Inicialmente para el desarrollo de PMCTrack-GUI hicimos una división del trabajo. Yo me encargaría de la parte que más tiene que ver con las cuestiones a bajo nivel, tales como la encapsulación de la información relevante de la máquina en un objeto Python o la modelización de los eventos hardware que pueden contar los PMCs; mientras, mi compañero de proyecto se encargaría de la implementación de los componentes gráficos con wxPython según los diseños realizados en la fase anterior.

Para realizar esta tarea, comencé haciendo un diagrama UML inicial. Debatimos e iteramos repetidas veces sobre ese diagrama. El diseño final al que llegamos consiste, principalmente, en una interfaz fachada para comunicarse con el resto de la interfaz gráfica y de varias clases para encapsular los eventos y la información de la máquina que se quiere monitorizar.
Además, necesitaba guardar de forma persistente los datos relativos a los eventos de cada modelo y a la configuración de los contadores de cada fabricante, de modo que elegí entre varias opciones la de usar ficheros XML por su sencillez de editar y su claridad en su lectura.

Así pues, mi contribución en esta parte del proyecto fue la implementación en Python de todo lo expuesto en el párrafo anterior, junto con el *parser* para los ficheros XML. Además de esto, también realicé los ficheros DTD que servirían para verificar la corrección de los ficheros XML, escribí algunos ficheros XML a mano y programé el script en Bash encargado de generar el resto de archivos XML a partir de sus correspondientes archivos CSV.

F.2.4. Implementación conjunta del soporte *multithreading*

Para la implementación del soporte multithreading en el kernel de PMCTrack, fue necesario primero un cuidadoso estudio de cómo funcionaba la herramienta anteriormente. Después de este estudio, tuvimos una reunión en la que discutimos varias posibilidades hasta dar con un diseño adecuado para conseguir una buena solución al problema.

Con este diseño ya planteado, mi compañero Jorge y yo realizamos diversas jornadas de trabajo conjuntas para la implementación completa de esta parte del proyecto. Posteriormente, realizamos un conjunto de pruebas mediante la herramienta `pmctrack`, usando benchmarks multihilo de la suite PARSEC y contrastamos sus resultados con otras fuentes para verificar si eran correctas. Posteriormente estos son los benchmarks que usaríamos para el caso de estudio del soporte multihilo en PMCTrack, comentados detalladamente en la sección 5.2 de este documento.

F.2.5. Diseño e implementación de libpmtrack. Refactorización de `pmctrack`

Para el diseño de la librería libpmctrack tuvimos varias reuniones conjuntas en las que decidimos las funciones que constituirían la interfaz básica y la interfaz avanzada. Con las interfaces ya definidas, yo fuí el encargado de implementar todas estas funciones en lenguaje de programación C.

Con la implementación de la librería casi terminada, nos dimos cuenta que había mucho código repetido y que se podría hacer que `pmctrack` hiciera uso de la librería en lugar de acceder directamente a PMCTrack. De modo que corrió a mi cargo también la refactorización del código de la herramienta `pmctrack` para que hiciese solamente llamadas a la librería en lugar de comunicarse directamente con el kernel.

F.2.6. Realización de los benchmarks para el caso de estudio de libpmctrack

Para el caso de estudio de libpmctrack –comentado en la sección 5.3 de esta memoria– adapté unas implementaciones mías en C++ de las estructuras de datos *binary heap* y *Fibonacci heap*.

Finalmente, también programé los benchmarks en C++ que usamos en dicho caso de estudio para los diversos análisis.

Bibliografía

[1] D. Antao, L. Tanica, A. Ilic, F. Pratas, P. Tomas, and L. Sousa. Monitoring performance and power for application characterization with the cache-aware roofline model. In *PPAM, Part I. LNCS, vol 8384*, pages 747–760, 2013.

[2] ARM. Streamline performance analyzer. `http://ds.arm.com/ds-5/optimize/`.

[3] ARM. Benefits of the big.LITTLE Architecture. `http://www.arm.com/files/downloads/Benefits_of_the_big.LITTLE_architecture.pdf`, 2015. Accessed: 2015-01-10.

[4] C. Bienia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. of PACT'08*, October 2008.

[5] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, Aug. 2000.

[6] Q. Chen and M. Guo. Adaptive workload-aware task scheduling for single-isa asymmetric multicore architectures. *ACM Trans. Archit. Code Optim.*, 11(1):8:1–8:25, Feb. 2014.

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

[8] J. L. Henning. Performance counters and development of spec cpu2006. *SIGARCH Comput. Archit. News*, 35(1):118–121, Mar. 2007.

[9] Intel. Intel performance counter monitor. `https://software.intel.com/en-us/articles/intel-performance-counter-monitor`.

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

[11] 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.

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

[13] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proc. of ISCA '04*, pages 64–75, 2004.

[14] G. Martínez Fernández, S. Sánchez Gordo, and S. Dronda Merino. Interfaz de uso de contadores hardware multiplataforma, 2012.

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

[16] Papi. Papi-c overview. `http://icl.cs.utk.edu/projects/papi/wiki/PAPIC:Overview`. Accessed: 2015-01-30.

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

[18] 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.

[19] 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)*, To appear.

[20] 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.

[21] L. Tanica, A. Ilic, P. Tomas, and L. Sousa. Schedmon: a performance and energy monitoring tool for modern multi-cores. In *Euro-Par Workshops, Part II. LNCS, vol 8806*, pages 230–241, 2014.

[22] J. Treibig, G. Hager, and G. Wellein. Likwid: a lightweight performance-oriented tool suite for x86 multicore environments. In *Proc. of ICPPW*, pages 207–216, 2010.

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

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

[25] 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.