DESARROLLO DE EXTENSIONES PARA HERRAMIENTA DE MONITORIZACIÓN DE RENDIMIENTO DE CÓDIGO ABIERTO

DEVELOPMENT OF EXTENSIONS FOR AN OPEN SOURCE PERFORMANCE MONITORING TOOL

JAIME SÁEZ DE BURUAGA BROUNS

FACULTAD DE INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO DE FIN DE GRADO
CURSO 2019-2020

DIRECTOR:
JUAN CARLOS SÁEZ ALCAIDE

Desarrollo de extensiones para herramienta de monitorización de rendimiento de código abierto

Memoria de Trabajo Fin de Grado

Jaime Sáez de Buruaga Brouns

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Septiembre 2020

Copyright © Jaime Sáez de Buruaga Brouns

Este documento está preparado para ser impreso a doble cara

Autorización de difusión y utilización

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

Título: Desarrollo de extensiones para herramienta de monitorización de rendimiento de código abierto.

Curso académico: 2019/2020.

Alumno: Jaime Sáez de Buruaga Brouns.

Jaime Sáez de Buruaga Brouns	Juan Carlos Sáez Alcaide

Agradecimientos

Agradezco al director de este trabajo las horas que ha dedicado a guiarme en todo este proceso y a resolver mis dudas. Su tutoría ha sido parte primordial en este proyecto. Muchas gracias, Juan Carlos.

Resumen

PMCtrack es una herramienta de código abierto para Linux que permite monitorizar el rendimiento de las aplicaciones haciendo uso de los contadores hardware del procesador (PMCs - Performance Monitoring Counters). Esta herramienta permite recabar valores de métricas relevantes sobre la ejecución de una aplicación, como el número de instrucciones por ciclo, o la tasa de fallos de predicción de saltos. Asimismo, PMCTrack proporciona información de monitorización hardware adicional no accesible mediante PMCs, como por ejemplo, el consumo de potencia o valores precisos del ancho de banda con memoria consumido por una aplicación. La flexibilidad de su API para recabar métricas de rendimiento desde distintos componentes del sistema operativo (SO), o el hecho de que está implementada en un módulo del kernel —lo cual permite extender la funcionalidad de la herramienta sin reiniciar el SO —, son algunas de las ventajas más relevantes de PMCTrack.

A pesar de sus ventajas, PMCTrack requiere actualmente un parche del kernel para funcionar, y gestiona los contadores hardware directamente, no haciendo uso del subsistema estándar de Linux para esta tarea (*perf events*). El objetivo principal del proyecto es dar los primeros pasos para permitir que PMCTrack pueda en un futuro llegar a funcionar en versiones de kernel Linux sin modificar. Para ello, en el proyecto se ha procedido a la creación de un *backend* de PMCTrack usando *perf events* —cuya API del kernel tiene escasa documentación—, y a la adaptación de los distintos componentes de espacio de kernel a nuevas versiones de Linux. Para llevar a cabo la evaluación del nuevo soporte añadido en PMCTrack, se ha realizado una validación exhaustiva del *backend* creado y se ha procedido a analizar la sobrecarga introducida en la lectura de contadores hardware, llevando a cabo una comparativa entre distintas herramientas y mecanismos disponibles.

Palabras clave: monitorización de rendimiento, contadores hardware, PMCtrack, Perf Events, kernel Linux, módulos del kernel.

Abstract

PMCtrack is an open-source **performance monitoring tool** for GNU/Linux that allows to monitor application performance using the processor hardware counters (PMCs – Performance Monitoring Tools). This tool allows to collect relevant metric values about the execution of an application, such as the number of instructions per cycle or the jump prediction failure rate. Also, PMCtrack provides additional hardware monitoring information not accesible through PMCs, such as power consumption or precise values of the bandwith with memory consumed by an appliation. The flexibility of its API to collect performance metrics from different components of the operatyng sistem (OS), or the fact that it is actually implemented in a kernel module —which allows to extend the functionality of the tool without restarting the OS —, are some of the most relevant advantages of PMCtrack.

Despite its advantages, PMCtrack currently requires a kernel patch to work, and manages the hardware counters directly, not making use of the standard Linux subsystem from this task (*perf events*). The main objective of the project is to take the first steps to allow PMCtrack to work on unmodified Linux kernel versions in the future. To achieve this goal, the project has proceeded to create a PMCtrack *backend* using *perf events* —whose kernel API has little documentation — and to adapt the different kernel space components to new versions of Linux. To carry out the evaluation of the new support added in PMCtrack, an exhaustive validation of the created *backend* has been carried out and the overload introduced in the reading of hardware meters has been analyzed, carrying out a comparison between different tools and mechanisms available.

Key words: performance monitoring, hardware events, scheduling metrics, perf API integration, Linux kernel, deadlock troubleshooting, interrupt context.

Índice

Capítulo 1

Introducción

PMCtrack es una herramienta de **monitorización de rendimiento** de código abierto para el sistema operativo GNU/Linux. Es una herramienta que fue específicamente diseñada para asistir a los desarrolladores de kernel en la implementación de algoritmos de planificación de procesos que exploten información proporcionada por los datos de los **contadores de monitorización de rendimiento** (PMC) para realizar optimizaciones **en tiempo de ejecución** [7].

A pesar de ser una herramienta orientada al sistema operativo, también permite recolectar valores de PMC desde **espacio de usuario** de distintas formas, necesarios para ayudar a los desarrolladores durante el proceso de diseño del planificador. Además de proporcionar la captura de métricas tales como el número de instrucciones por ciclo o la tasa de fallos de caché de último nivel, PMCtrack proporciona al planificador [1,2,3,4,5,6] y a distintos componentes de espacio de usuario de la herramienta otras **métricas de monitorización hardware** disponibles en procesadores modernos como el **espacio usado en el último nivel de caché** o el **consumo de energía**.

Para el acceso a los contadores hardware a bajo nivel, es necesario emplear código ensamblador por lo que cualquier herramienta que accede a ellos es dependiente de arquitectura. PMCtrack ofrece una ventaja sustancial sobre otras herramientas, que es el hecho de que gran parte de su funcionalidad está implementada dentro de un módulo cargable del kernel. Esto permite que casi cualquier extensión que desee realizarse en la herramienta pueda implementarse y activarse sin necesidad de reiniciar el sistema operativo. Sin embargo, PMCTrack posee limitaciones estructurales: (1) necesita un parche del kernel para funcionar, y (2) precisa soporte de bajo nivel para el acceso directo a los contadores hardware en las diversas arquitecturas de procesador y plataformas donde desee utilizarse. Para cada arquitectura se debe implementar un *backend* específico, es decir, un módulo software específico de arquitectura para acceder a bajo nivel a los contadores.

Otra de las herramientas disponibles en Linux para simplificar la gestión y el acceso a los contadores hardware es **Perf**. Se trata de una herramienta de *tracing* y monitorización de rendimiento integrada en el kernel Linux (desde la v2.6.31 en 2009) con control desde **espacio de usuario.** Gran parte de la funcionalidad de *perf* puede explotarse desde la línea de comandos, y la herramienta correspondiente nos permite, entre otras cosas, realizar un ***profiling* estadístico** de todo el sistema [8].

Perf es actualmente una de las herramientas más utilizadas para el análisis de rendimiento. Sin embargo, tiene dos limitaciones muy imponentes. La limitación teórica es la escasez de documentación disponible sobre la herramienta; por ejemplo, la mayoría de los eventos y alias de Perf no están documentados. La limitación práctica es más severa: es una herramienta implementada completamente dentro del kernel, lo que provoca que cualquier extensión que se quiera incorporar requiera la compilación del kernel Linux, y el correspondiente reinicio de la máquina para su activación y depuración. Perf contiene una API de bajo nivel que surgió para que diversas herramientas de monitorización del rendimiento pudieran utilizarla para recopilar información de los contadores. Además, Perf no fue diseñado específicamente para realizar monitorización desde el propio kernel, y la API que actualmente ofrece soporte para esto ha sufrido numerosas modificaciones en la última década.

El objetivo principal del proyecto es dotar a PMCtrack de algunas de las extensiones más solicitadas por la comunidad de usuarios, como proporcionar un *backend* de la herramienta basado en el sistema *perf events* del kernel Linux, y portar los distintos componentes de PMCtrack a nuevas versiones *longterm* del kernel (como la versión 5.4.y).

1. Motivación y objetivos del proyecto

El problema de usar herramientas como ésta, es que, como hemos mencionado anteriormente, son **dependientes de arquitectura**, y actualmente hay un sinfín de arquitecturas de procesador equipadas con contadores hardware, cada una con sus propios eventos y mecanismos de acceso desde el software de sistema. El objetivo a corto plazo de este proyecto es la completa **integración** de la funcionalidad de la API de *perf events* en lo que respecta al acceso a contadores hardware desde PMCTrack, encapsulando esta funcionalidad como **un nuevo backend de PMCTrack.** Con ello se persigue ofrecer el acceso indirecto (mediante llamadas de API) a los contadores hardware de cualquier arquitectura/plataforma soportada por *perf events* desde PMCTrack.

Como se ha mencionado anteriormente, PMCtrack no puede utilizarse en kernels Linux “vanilla” o sin modificar, sino que requiere aplicar un parche específico del kernel para funcionar correctamente. Esta limitación provoca que, en la práctica, los usuarios avanzados de PMCTrack deban poseer un amplio conocimiento del funcionamiento del kernel Linux. Si el parche correspondiente está disponible para la versión del kernel que se desea utilizar, un usuario con conocimientos de administración en Linux podría compilarlo y hacer funcionar PMCtrack. Si el parche no está disponible, solo un usuario con conocimientos medios de desarrollo en el kernel Linux será capaz de hacer

funcionar PMCTrack, ya que deberá portar el parche del kernel a la versión deseada. A pesar de que el parche del kernel de PMCTrack es simple y afecta a pocos ficheros y a código relativamente estable del kernel, el uso intensivo de la API de Linux desde el módulo del kernel de PMCtrack puede romper la compatibilidad entre distintas versiones del kernel, como discutiremos en las siguientes secciones. La existencia de incompatibilidades de este tipo requiere en ocasiones la modificación del módulo del kernel de PMCTrack para su correcto funcionamiento en versiones arbitrarias del kernel (no soportadas oficialmente para PMCTrack).

Con este proyecto se pretende que estas limitaciones puedan eliminarse a medio o largo plazo mediante la creación de un *backend* de PMCtrack basado en la API de perf. De este modo se hace uso del subsistema de monitorizacion de rendimiento del kernel Linux, que desde hace algunos años constituye el estándar de facto para gestionar el acceso a bajo nivel a los contadores hardware, y que tiene soporte para numerosas arquitecturas de procesador en la *mainline* de Linux.

Para la completa eliminación de las citadas limitaciones existen 3 barreras significativas: la primera es la integración de la API de perf como un backend más dentro de la herramienta PMCtrack; la segunda es el soporte para kernels nuevos que permitan el uso de tecnologías de *tracing* de eventos en el planificador de Linux como TracePoints y *kprobes* y; en último lugar, eliminar o reducir a su mínima expresión el parche necesario para la ejecución de PMCtrack en un kernel *vanilla*. PMCtrack necesita recibir notificaciones del planificador (mediante *callbacks*) para saber cuándo suceden eventos críticos –como la creación/terminación de procesos o los cambios de contexto--, y este es el principal problema para eliminar completamente la dependencia del parche del kernel.

El objetivo principal de este proyecto es eliminar las dos primeras barreras. La eliminación de la dependencia de PMCTrack de su parche del kernel es un objetivo a largo plazo demasiado ambicioso para abordar en este Trabajo de Fin de Grado. Al fin y al cabo, al comienzo de este proyecto todavía era necesario estudiar si con el API de *perf events* era realmente viable crear un *backend* para PMCtrack. La escasa documentación de perf y los numerosos interrogantes sobre las adaptaciones necesarias en PMCTrack, han obligado a plantear el proceso de eliminación del parche de PMCTrack como trabajo futuro.

2. Plan de trabajo

Para alcanzar los objetivos del Trabajo de Fin de Grado mencionados en la sección anterior, se ha dividido el desarrollo del proyecto en distintas etapas:

- Lectura de documentación de PMCtrack y análisis del código fuente de la herramienta.
- Instalación del parche del kernel de PMCtrack modificado para la versión 4.9.33_x86.
- Lectura de documentación de la API de perf.
- Estudio de ejemplos prácticos del uso de la API de perf en el kernel [18].
- Estudio del código del kernel Linux asociado con las llamadas de la API de perf-
- Realización de cambios necesarios en la implementación de PMCtrack para su integración con Perf.
- Ejecución de experimentos de validación de la funcionalidad incorporada en PMCtrack y análisis de la sobrecarga producida.
- Construcción de gráficas y análisis de resultados.
- Redacción de la memoria.

3. Estructura de la memoria

El resto del documento se estructura en los siguientes capítulos:

- El **capítulo 2** contiene una descripción del contexto histórico en el que surgió PMCtrack así como del funcionamiento de esta herramienta de monitorización del rendimiento.
- El **capítulo 3** presenta una descripción de la herramienta de monitorización de rendimiento perf, incidiendo de manera especial en el funcionamiento y uso de la API.
- El **capítulo 4** expone todo lo relacionado con la creación del nuevo *backend* de PMCtrack basado en la API de perf, centrándose en la necesidad de la integración, así como en los detalles de implementación y en las limitaciones detectadas.
- El **capítulo 5** define los cambios necesarios para la inclusión del soporte necesario en PMCTrack para funcionar en la versión 5.4 del kernel de Linux, detallando los cambios necesarios tanto en el parche del kernel como en el módulo cargable del kernel.
- El **capítulo 6** discute los resultados experimentales obtenidos, y discute tanto su validación (precisión de las métricas de rendimiento reportadas) como la sobrecarga asociada a la lectura de contadores.
- El **capítulo 7** presenta las conclusiones finales del Trabajo de Fin de Grado y discute el trabajo futuro.

Capítulo 2

Descripción PMCtrack

La mayor parte de los procesadores actuales cuentan con una serie de **contadores hardware** para monitorización (**PMCs**), que permiten a los usuarios obtener métricas relevantes sobre la ejecución de sus aplicaciones, como pueden ser los fallos en último nivel de caché (LLC misses) o el número de instrucciones por ciclo ejecutadas (IPC). Estas métricas resultan muy útiles para identificar distintos problemas en el software tales como la existencia de cuellos de botella en el acceso y uso de recursos [7,9].

A mediados de la década de los 2000, los fabricantes de procesadores proporcionaban muy escasa documentación sobre el uso de contadores hardware, lo que tenía como consecuencia una escasez de herramientas de monitorización de contadores hardware en GNU/Linux.

Entonces, en el Departamento de Arquitectura de Computadores y Automática (DACyA) de la Universidad Complutense de Madrid surgió la necesidad de crear una herramienta para proporcionar al planificador del kernel Linux acceso a los contadores hardware para realizar optimizaciones en tiempo de ejecución. Esto desenvocó en la creación de una herramienta propietaria que utilizaba nuevas rutinas en el sistema para acceder a los contadores hardware y proporcionarle los datos al planificador.

Esta herramienta surge con la intención de realizar mediciones de manera sencilla y pasar los resultados al planificador, ya que de las pocas herramientas que existían para realizar la lectura de los contadores hardware ninguna ofrecía los datos al planificador, sino que los cedía a herramientas de espacio de usuario.

En la misma época, empezaron a crearse muchas herramientas que realizaban esta tarea, tales como oProfile[10] o perfmon2[11], pero todas tenían algo en común: estaban orientadas a una monitorización externa por parte del usuario, en la que el usuario accedía, desde espacio de usuario, a los contadores para la monitorización del rendimiento de las aplicaciones.

Por el contrario, la herramienta desarrollada en el DACyA estaba orientada a ceder los datos al kernel con una API para hacerlo eficiente y no bloqueante,

Siguieron apareciendo herramientas para realizar esta tarea y cada una requería de un parche en el kernel. De alguna manera oProfile subió al primer puesto y se introdujo en

las fuentes del kernel de Linux. Había algunos aspectos de oProfile que no eran aceptados por la comunidad y el resto de herramientas seguían creando sus parches para poder seguir funcionando, pero eran incompatibles entre sí.

En este escenario, hacia el año 2008, distintas empresas y la comunidad de Linux decidieron crear un *framework* común que realizara el acceso a los contadores, y después crear una API de bajo nivel para poder adaptar todas las herramientas a dicha API. Este es el origen de *perf events*. Perf events tenía la intención de ser la única herramienta con acceso directo a los contadores hardware y, una vez ya fuera robusta, se crearía una API para que el resto de herramientas pudieran pasarse a Linux sin requerir parchear el kernel. En este momento, todas las herramientas de acceso a los contadores hardware seguían siendo orientadas a monitorización externa.

La principal diferencia de PMCtrack con estas herramientas es que el objetivo desde el primer momento era distinto: el resto de herramientas buscaban una monitorización externa en la que el usuarío recibiera los datos de los contadores, PMCtrack quería monitorizar desde el propio kernel para poder hacer optimizaciones en tiempo de ejecución.

En 2010, cuando los principales fabricantes de procesadores como Intel y ARM ya llevaban invertidos muchos recursos, y la herramienta *perf events* era robusta, decidieron introducir la API a bajo nivel. Esta API en primer momento generó mucho rechazo, ya que requería de un diseño estructural muy específico para su uso, de manera que si la herramienta que quería adaptarse a la API estaba bien diseñada se podía, más o menos, adaptar para su correcta integración; pero si esta tenía demasiado parche del kernel incompatible con el diseño de *perf* no era posible. Esto pasó con perfmon2, que dejó de mantenerse. Sin embargo, oProfile se rediseñó.

En el año 2012 se produce el primer anuncio público de PMCtrack, cuando unos estudiantes de la Facultad de Informática, como parte de su proyecto de fin de carrera, implementaron una versión poco robusta de los componentes de espacio de usuario que le faltaban a la herramienta creada en 2007 en el DACyA. No es hasta el año 2015 cuando aparece la primera versión pública, y se libera el código fuente de PMCtrack.

En la actualidad, el API de perf, es difícil de utilizar para su uso interno desde el kernel. Tiene funciones muy versátiles que permiten ser invocadas desde muchos sitios, pero una mala invocación produce sobre bloqueos críticos en la máquina. Necesitan de una invocación sumamente cuidadosa, previamente habiendo desgranado el código del kernel Linux para saber; por ejemplo, que una invocación a la función *perf_read* desde una CPU distinta a la de lectura va a producir un bloqueo y desencadenar *inter-processor interrupts*; no se pueden desactivar las interrupciones en las invocaciones a las funciones de la API de bajo nivel de perf.

En definitiva, el diseño de PMCtrack fue totalmente al revés que el resto de estas herramientas: primero se pensó en ceder los datos de los contadores a componentes del sistema operativo, y después al usuario.

A pesar de las limitaciones de la API de *perf events* anteriormente citadas, esta API es hoy día suficientemente versátil como para poder plantearse la integración. Así surge este Trabajo de Fin de Grado.

PMCtrack, en la actualidad, es una herramienta de código abierto de monitorización de rendimiento orientada al sistema operativo GNU / Linux. A diferencia de otras herramientas de monitorización, las funciones de PMCTrack y la API en el kernel permiten al programador del sistema operativo acceder a los datos de PMC por hilo de una manera independiente de la arquitectura [12].

La arquitectura actual de PMCtrack es la siguiente:

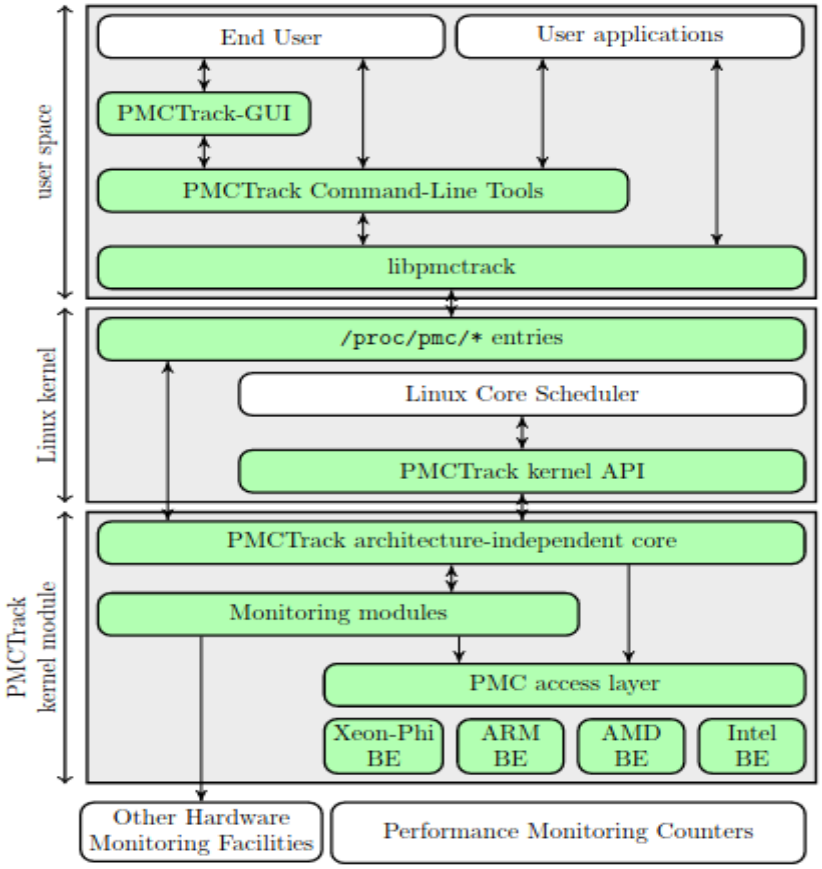


FIGURA 1
Arquitectura de PMCtrack.

Lo que se quiere conseguir con este proyecto es la integración de la API de perf como un *backend* más por debajo de la capa *PMC access layer,* pensando en un futuro en el cuál se pueda ejecutar PMCtrack en un kernel *vanilla* sin el necesidad de parchear el kernel como hasta ahora. Hasta este momento, para el correcto funcionamiento de PMCtrack es necesario parchear el kernel, ya que necesita información de eventos que surgen en el planificador.

Usando PMCtrack desde la línea de comandos

Una de las formas de acceder a la funcionalidad de PMCtrack desde espacio de usuario es empleando el comando `pmctrack`, cuyo uso describimos brevemente en esta sección.

En esta sección proporcionamos una breve descripción del uso de dicha herramienta. En primer lugar, se debe indicar a PMCtrack los **tipos de eventos** que se quieren medir. Esta es una diferencia importante de PMCtrack con respecto a perf: los eventos de PMCtrack se encuentran en un CSV en espacio de usuario, pudiendo ser modificados en cualquier momento; sin embargo en *perf* gran parte de ellos están hardcodeados en el kernel, pudiendo también usar cualquier evento a través de la notación *raw*.

Eventos muy utilizados son por ejemplo los **fallos en último nivel de caché** o el número de **ciclos** o **instrucciones**. También se debe indicar el **modo de ejecución**, que puede ser basado en eventos (EBS), basado en tiempo (TBS) o modo planificador. El modo de **muestreo por tiempo** (TBS – *Time-Based Sample*) permite al usuario recopilar datos de rendimiento de una aplicación desde **espacio de usuario** a intervalos de tiempo regulares. Para ilustrar el funcionamiento consideraremos el siguiente ejemplo [13].

$ pmctrack -T 0.1 -c instr,cycles,llc_misses,0xc0 ./benchmarks/common/lbm06
[Event-to-counter mappings]
pmc0=instr
pmc1=cycles
pmc3=llc_misses
pmc4=0xc0
[Event counts]

nsample	*pid*	*event*	*pmc0*	*pmc1*	*pmc3*
1	*7145*	*tick*	*69528627*	*74240619*	*8105*
2	*7145*	*tick*	*185300393*	*117217402*	*278254*
3	*7145*	*tick*	*413761175*	*217654045*	*5313725*
4	*7145*	*tick*	*424237375*	*217668271*	*3334511*
5	*7145*	*tick*	*423341399*	*217468854*	*3327588*

6 7145 tick 424025201 217680656 3333617
7 7145 tick 423220688 217681899 3326940
8 7145 tick 423463423 217675081 3329947
9 7145 tick 422729601 217686908 3322685
10 7145 tick 422942565 217607936 3325537
11 7145 tick 422203571 217683128 3318032
12 7145 tick 422221847 217679730 3320282
...

Este comando ejecuta el modo de muestre por tiempo (TBS) cada 0.1 segundos, parámetro pasado a la opción *-T*. A través de la opción *-c* se indican los eventos que se quieren medir. Se debe pasar una cadena de caracteres, con el nombre de los eventos separados por comas. En nuestro caso, *instr,cycles,llc_misses* indica que se quieren medir el número de instrucciones por ciclo, de ciclos y de fallos en último nivel de caché. Por último, se debe indicar qué algoritmo se quiere ejecutar, y nosotros hemos elegido el benchmark LBM de SPEC CPU 2006.

En la salida, cada columna indica una métrica. La primera, indica el número de muestra. En segundo lugar, se imprime el PID del proceso que ejecuta la lectura. Para continuar, se muestra el tipo de evento: en nuestro caso es un *tick* del timer, puesto que el modo es TBS y se ejecutan mediciones cada 0.1 segundos. Ahora, se imprime una columna por cada evento medido: en la columna *pmc0* se imprime el número de instrucciones, en la *pmc1* el número de ciclos y en la *pmc3* el número de fallos en último nivel de caché.

Una característica destacable de **pmctrack** es su capacidad de obtener los valores de otros contadores denominados virtuales. Esto permite obtener la contabilización de los eventos de PMC 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.

El modo de **muestreo basado en eventos** (EBS – *Event-Based Sample*) constituye una variante del anterior en el que los valores de los PMCs se recaban cuando el número de ocurrencias de un cierto evento alcanza un cierto umbral *U*. Para soportar EBS, PMCtrack exporta la característica de **interrupción por desbordamiento** presente en la mayoría de las Unidades de Monitorización de Rendimiento (PMUs) de los procesadores actuales. Esta característica produce que, en plataformas x86, las mediciones en PMCtrack sucedan en un contexto de una interrupción no enmascarable, surgiendo una limitación de integración ya que las llamadas de lectura de la API de perf son, en general, bloqueantes. Este problerma ha sido correctamente resuelto y se detalla en el **capítulo 4** [13].

Cuando se activa EBS, 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 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**, pudiendo establecerse el valor del umbral en dicho string, como en el siguiente ejemplo [18].

Ejemplo de ejecución EBS [13]:

```
$ 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
...
```

En el ejemplo, la columna *pmc3* muestra el número de fallos de caché por cada 500 millones de instrucciones retiradas. Sin embargo, los valores de la columna *pmc0* no reflejan con exactitud el número especificado en el flag *ebs*. Esto se debe a que la interrupción PMU en los procesadores modernos con ejecución fuera de orden y superescalar no se sirve inmediatamente después del desbordamiento del contador.

Capítulo 3

Perf

Existe una herramienta interna del kernel de Linux llamada **perf,** que implementa una interfaz de aplicación sencilla para recuperar información detallada sobre eventos que ocurren en el sistema operativo. Estos eventos pueden servir de ayuda para resolver funciones avanzadas de rendimiento de solución de problemas [8].

Las preguntas que suele responder *perf* sobre el estado completo del rendimiento del sistema operativo son las siguientes [14].

- ¿Qué rutas de código están causando errores en la caché de segundo nivel?
- ¿Están las CPUs paradas por culpa del sistema de paginación?
- ¿Qué rutas de código están asignando memoria y cuánta requieren?
- ¿Qué está desencadenando retransmisiones TCP?
- ¿Se está invocando a cierta función del kernel y con qué frecuencia?
- ¿Por qué motivos se bloquean los hilos que se ejecutan en el sistema?

Como se ha explicado en el capítulo anterior, *perf* aparece en el año 2009 como resultado del intento de unificación de las herramientas de acceso a los contadores hardware bajo un mismo *framework* o subsistema del kernel; creando después una API de bajo nivel para usar a su libre albedrío. La API de bajo nivel llegó mucho más tarde que *perf*, lo que provocó que fuera muy limitada.

Esta API ha ido evolucionando hasta un punto en el cual es suficientemente madura para poder usarla en PMCtrack. La documentación es lamentable, y ha sido necesario un estudio concienzudo del kernel de Linux para poder ver el contexto de invocación de las funciones, así como el modo de uso.

En la actualidad, *perf events* es una herramienta muy potente que ha sido desarrollada por los principales fabricantes de procesadores como Intel o ARM. Actualmente, permite capturar métricas y obtener trazas muy exhaustivas. Por ejemplo, permite hacer lo siguiente:

- Estadísticas/recuento: incrementa un contador entero en un evento.
- Muestreo: recolecta detalles como el **contador de programa** o la **pila** de un subconjunto de eventos.
- Trazado: recaba numerosos detalles de cada uno de los eventos.

La funcionalidad de los comandos perf se puede dividir en los siguientes grupos: [14]

- Listado de eventos.
- Recuento de eventos disponibles en la plataforma: Recaba estadísticas de los contadores hardware del procesador (de cache de nivel 1, de ciclos, etc.) dados un comando, un PID, etc.
- Perfilado: Muestra un perfilado de funciones de la CPU dados un comando, un PID, etc.
- Trazado: Realiza un trazado sobre el sistema.
- Mezcla. Realiza funciones varias como el trazado de llamadas al sistema por proceso o muestrear las pilas a 2 niveles de profundidad.
- Especial: Graba eventos de caché e informa de eventos de línea de caché de grabaciones anteriores.
- Informe: Listado, volcado y muestra de todo tipo de informes.

Perf permite capturar estadísticas de una serie de eventos, y emplea una interfaz unificada para diferentes fuentes de eventos y puntos de instrumentalización del kernel. La siguiente figura ilustra las fuentes del evento:

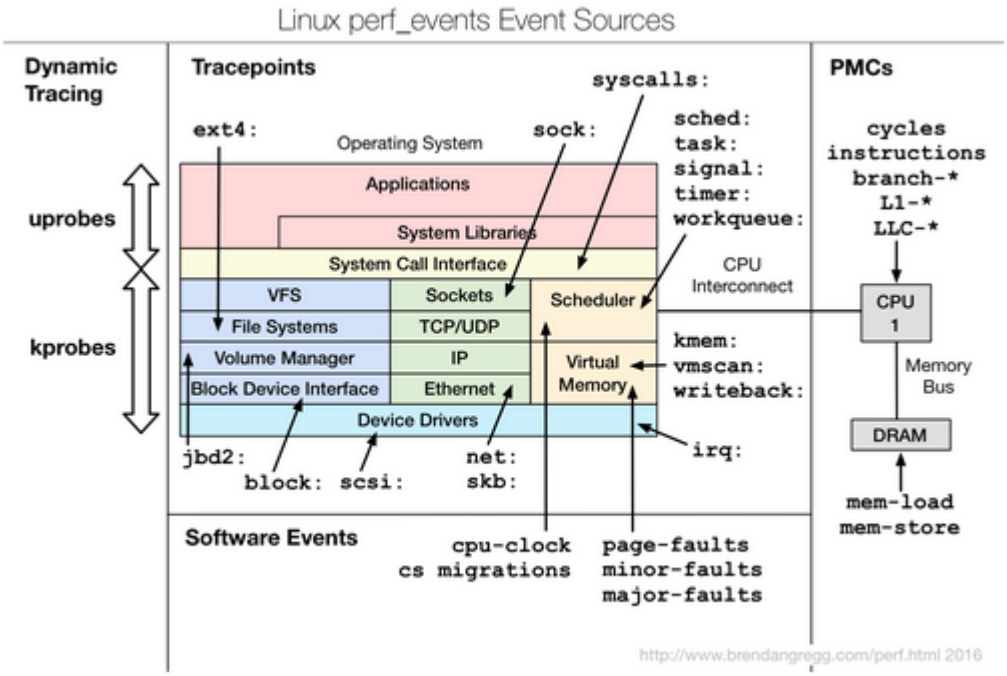


FIGURA 2
Fuente de eventos perf_event de Linux.

Los tipos de eventos son [14]:

- Eventos Hardware. Contadores de monitorización de rendimiento de CPU.
- Eventos Software. Eventos de bajo nivel basados en contadores software del kernel (migraciones CPU, errores del major/minor; …).
- Eventos de Kernel Tracepoints. Puntos de instrumentación estáticos a nivel de kernel.
- Puntos de traza definidos por el usuario. Puntos de traza estáticos para programas y aplicaciones a nivel de usuario.
- Trazado dinámico. Puede crear eventos dinámicos en cualquier ubicación usando kprobes o uprobes.
- Perfilado por tiempo. Se pueden capturar *snapshots* con una frecuencia arbistraria.

Los eventos se miden a través de el comando `perf stat`. A continuación se muestra un ejemplo de los *counting events* que se pueden medir con perf [8,15].

```
# Estadísticas de contador de CPU para el comando especificado:
perf stat command

# Estadísticas de contador de CPU para el PID especificado, hastas
Ctrl-C:
perf stat -p PID

# Estadísticas de CPU varias, por todo el sistema, por 10 segundos:
perf stat -e cycles,instructions,cache-references,cache-misses,bus-
cycles -a sleep 10

# Estadísticas varias de caché de nivel 1 para el comando
especificado:
perf stat -e L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores
command

# Estadísticas de datos de TLB de CPU para el comando especificado:
perf stat -e dTLB-loads,dTLB-load-misses,dTLB-prefetch-misses command

# Estadísticas del último nivel de caché CPU para el comando
especificado:
perf stat -e LLC-loads,LLC-load-misses,LLC-stores,LLC-prefetches
command

# PMCs: ciclos de recuento y puestos frontend a través de
especificaciones sin procesar:
perf stat -e cycles -e cpu/event=0x0e,umask=0x01,inv,cmask=0x01/ -a
sleep 5

# Contador de llamadas al sistema por tipo para el PID especificado,
hasta Ctrl-C:
perf stat -e 'syscalls:sys_enter_*' -p PID

# Contador de eventos del planificador para el PID especificado, por 5
segundos:
perf stat -e 'sched:*' -p PID
```

```
# Count ext4 events for the entire system, for 10 seconds:
perf stat -e 'ext4:*' -a sleep 10

# Contador de eventos de E/S del dispositivo de bloque para todo el
sistema durante 10 segundos:
perf stat -e 'block:*' -a sleep 10

# Contador de eventos de vmscan , imprimiendo un informe cada segundo:
perf stat -e 'vmscan:*' -a -I 1000
```

FIGURA 3

Ejemplo de medición de eventos Linux perf.

Perf es un sistema basado exclusivamente en eventos y no en contadores, por lo que las mediciones que puede llevar a cabo están limitadas a la lista de tipos de eventos simbólicos disponibles para la plataforma. Cada versión del kernel de Linux contiene una lista de eventos asociada que se puede consultar a través del comando ***perf list***. Para la versión 5.3 del kernel, los eventos perf definidos son de varios tipos [8,9]:

- Eventos de herramienta.
- Eventos de kernel PMU.
- Eventos de caché.
- Eventos de frontend.
- Eventos de memoria.
- Eventos relacionados con el pipeline.
- Eventos de **uncore.**
- Eventos de memoria virtual.
- Otros eventos.

Esto significa que la funcionalidad de *perf* está muy restringida a factores externos como la plataforma hardware usada y la versión del kernel en uso.

Perf, al igual que PMCtrack, tiene parte de su implementación específica de arquitectura, pero al ser una herramienta integrada en el kernel Linux tiene en las mismas fuentes soporte para un gran número de arquitecturas en, por ejemplo para x86, el directorio */arch/x86/events/perf.c*.

Una limitación, que no es propia de perf sino al integrarse en PMCtrack, es que utiliza mecanismos bloqueantes en su API del kernel, lo cual complica su utilización desde PMCtrack. Esta limitación se resolverá y detallará en el capítulo 3: **Backend perf-PMCtrack**.

El aspecto más relevante del API de perf para este Trabajo de Fin de Grado es la parte de medición de contadores de monitorización de rendimiento de PMCs a través de los eventos hardware.

Nueva API de perf

La API de perf ha sido desarrollada años más tarde que perf y han llevado flujos de desarrollo distintos, por lo que ha sido muy limitada hasta hace no mucho. Además, la documentación es muy deficiente, y ha sido necesario un estudio exhaustivo del kernel de Linux [16] para conocer el funcionamiento de la API. Además, los propios desarrolladores no parecen inclinados a actualizar la documentación [17].

Existe una estructura *perf_event*, definida en el fichero de las fuentes del kernel */include/linux/perf_event.h*, que representa un evento de rendimiento hardware. En primer lugar, contiene un campo *union* con los posibles eventos que puede representar la estructura.

Éste puede ser un evento hardware, representado por los siguientes campos:

```
struct {
     u64             config;
     u64             last_tag;
     unsigned_long   config_base;
     unsigned_long   event_base;
     int             event_base_rdpmc;
     int             idx;
     int             last_cpu;
     int             flags;

     struct hw_perf_event_extra extra_reg;
     struct hw_perf_event_extra branch_reg;
};
```

En segundo lugar, el evento puede ser de software, en cuyo caso viene representado por un **struct hrtimer**. Para continuar, contiene un campo **task_struct,** asociado con el proceso tarea en caso de que el evento sea un evento por tarea (*per task event*). A continuación, contiene una serie de *flags* de estado usados para trackear el estado de *perf*.

De la inicialización de los contadores de *perf* se encarga la función de creación de contadores de la API **struct perf_event *perf_event_create_kernel_counter(struct perf_event_attr *attr, int cpu, struct task_struct *task, perf_overflow_handler_t overflow_handler, void *context).** Esta función se encarga de crear y devolver un evento perf asociado con unos parámetros. Esta invocación requiere de una estructura **perf_event_attr**, que contiene una serie de atributos necesarios como el evento a monitorizar (ya sea a través de la configuración en crudo o a través de los eventos definidos en las fuentes del kernel para perf) o la *callback* de desbordamiento, en caso de modo EBS.

Para seguir, se deben activar los contadores *perf* a través de una función de la API de *perf* llamada **void perf_event_enable(struct perf_event *event)**. Esta función

La lectura de un evento hardware a través de la API de perf es relativamente simple. En primer lugar, se invoca a una función llamada **u64 perf_event_read_value (struct perf_event *event, u64 *enabled, u64 *running)** (anteriormente mencionada), cuya función es leer y devolver el valor del contador asociado al evento **event**. Esta función es de naturaleza **bloqueante**, así que no debe ser invocada en un contexto de interrupción. En el siguiente capítulo se hace una descripción detallada de los escenarios posibles de interbloqueo que pueden suceder de la invocación a dicha función y los mecanismos de sincronización utilizados para evitar estas situaciones. Para la lectura del evento, esta función invoca a otra llamada **u64 perf_event_read(struct perf_event *event)**, cuya función es actualizar el valor del contador en la estructura **event** asociada con el evento, siempre que el evento esté activo y ejecutándose en la CPU. La lecctura de los contadores en *perf* es muy eficiente, ya que no para los contadores, lee el valor, los resetea y los vuelve a poner a correr, sino que guarda el valor anterior del contador y en la lectura hace un *diff* con el valor anterior, siendo el valor actual al diferencia entre ambpos. Esta función, para devolver el valor de la lectura, invoca a otra llamada **u64 perf_event_count(struct perf_event *event)**, que devuelve el valor del contador, almacenado en **event→count**.

En último lugar, se pueden liberar los contadores *perf* a través de la invocación a la función **int perf_event_release_kernel(struct perf_event * event)**.

Capítulo 4

Backend perf-PMCtrack

En este capítulo se describe el proceso de integración del *backend* de *perf* en PMCtrack, empezando por explicar la **necesidad** de dicha integración, siguiendo por los detalles de implementación y finalizando con una discusión sobre las **limitaciones** de la integración.

El *backend* se ha implementado sobre x86 como prueba de concepto, pero de forma trivial se puede adaptar a otras arquitecturas reutilizando el código de otros *backends* de PMCtrack.

1. Necesidad de creación

La necesidad de creación de un *backend* de perf surge del objetivo a largo plazo del proyecto: la ejecución completa de PMCtrack para cualquier arquitectura sin necesidad de hacer customizaciones en el kernel. Para ello, el primer paso es la creación de un nuevo *backend*, accediendo a los contadores a través de la API de bajo nivel que ofrece perf.

PMCtrack es una herramienta muy flexible apra interactuar con los contadores hardware del kernel, pero requiere soporte específico para la arquitectura del procesador de la plataforma. Es por eso mismo que se ha hecho un *wrapper* de la API de *perf* en última instancia para que pueda ser utilizada por la interfaz de PMCtrack para realizar mediciones en cualquier arqutiectura soportada por *perf*, y sin que PMCtrack acceda a los registros de los contadores hardware de forma directa.

PMCtrack tiene una característica especial y es que se carga como un módulo del kernel. Si, en un futuro, es posible eliminar la necesidad de parchear el kernel, podría ser usado en cualquier arquitectura de cualquier procesador soportado por *perf* en un entorno de producción real en el que una customización del kernel es un proceso crítico, mientras que la carga de un módulo del kernel es algo trivial.

2. Detalles de implementación

Todos los cambios realizados en el proyecto se han implementado empleando compilación condicional bajo el **símbolo de preprocesado** *CONFIG_PMC_PERF*, para que el resto de arquitecturas del proyecto siga funcionando correctamente.

El backend de PMCtrack-perf requiere de cambios en las siguientes estructuras:

- **pmon_prof_t**: estructura de datos por-hilo mantenida por el módulo de kernel de PMCtrack. Esta estructura es la encargada de contener los datos recuperados por PMCtrack.
- **hw_event**: estructura encargada de representar un evento hardware.

2.1 *Backend* de perf y acceso a los contadores

En primer lugar, y comenzando por los aspectos más sencillos de la implementación, se va a documentar el cambio estructural requerido por la implementación del *backend* PMCtrack-perf.

Es necesario efectuar la creación de un directorio **perf** en la ruta *src/modules/pmcs/include/pmc*, en el cual se han de alojar los ficheros que se encargan de declarar las funciones de acceso a lectura de los PMCs: **pmu_const.h**, que tan solo contiene tres macros, y **hw_events.h**. Este último módulo, *src/modules/pmcs/include/pmc/hw_events.h*, contiene la estructura **hw_event**, a la cual ha sido necesario añadir una serie de campos para el correcto funcionamiento de los modos TBS y EBS a través del *backend* de *perf*. El primer campo es un **struct perf_event**, que representa al evento *perf* y donde se va a almacenar toda la información devuelta por dicha API. El segundo campo consiste en un **task_struct**, contenedor de información relacionada con el proceso encargado de realizar las lecturas de los contadores. Para seguir, se añade un parámetro **reset_value**, que indica el parámetro umbral para el modo EBS; o dicho de otra forma, el tiempo periódico en el que se deben realizar las lecturas. A continuación se ha añadido un parámetro **last_read_value**, el cual representa el último valor del PMC leído. Para terminar, ha sido necesario añadir un parámetro **old_value**, que representa al anterior valor. Esto tiene la explicación en que *perf* no resetea el valor de los contadores en al lectura, si no que guarda el valor anterior del contador y hace un *diff* en le lectura, disminuyendo la carga operacional. En dicho módulo se implementan una serie de funciones encargadas del acceso a los PMCs y que se documentan a coninuación.

En la estructura antes mencionada **pmon_prof_t**, es necesario añadir un campo asociado a una tarea diferida para poder realizar la función de medición de rendimiento a través de la API perf en un contexto seguro para código bloqueante, ya que las tareas

diferidas se ejecutan en un contexto de un kernel *thread*. Este campo se denota **struct work_struct *read_counters_task**.

A continuación se va a documentar el proceso de cambio que han sufrido las funciones de bajo nivel de acceso a los PMCs con respecto a la versión sin el backend de perf. En primer lugar, vamos a documentar el fichero **hw_events.h** descrito anteriormente, contenedor de funciones de acceso a los PMCs. Para empezar, es necesaria la re-implementación de la función **__read_count_hw_event**, la cual se encarga de la invocación a la función de lectura de contadores de la API de perf (**perf_event_read_value**) y actualiza los valores contenidos en el evento **hw_event** entrante por parámetro, de manera que se actualiza el valor **hw_event→old_value** al valor devuelto por la función de lectura de la API, y se actualiza el valor de **hw_event→last_read_value** al antiguo valor de **hw_event→old_value**. En último lugar, es necesario implementar un *getter* y un *setter* para el valor de parámetro umbral de EBS, **hw_event→reset_value**. Todas estas funciones son invocadas desde el módulo de PMCtrack indepediente de arquitectura **mchw_core.c**.

```
struct hw_event{
     struct perf_event *event;
     struct task_struct *task;
     uint64_t reset_value;
     uint64_t last_read_value;
     uint64_t old_value;
};
```

Para seguir, se va a modificar la función encargada de transformar un **array de configuración de contadores** independiente de plataforma **pmc_usrcfg_t** en una estructura de bajo nivel que contenga los datos necesarios para la configuración de contadores hardware. En primer lugar, ha sido necesario añadir un parámetro **struct task_struct *task** a los parámetros entrantes de la función, que en un futuro será usado para inicializar el task_struct dentro de la estructura contenedora de los eventos de hardware de bajo nivel (**low_level_exp**). Para ello, recupera la configuración de los eventos hardware que han de ser activados a través de la sentencia *(pmc_cfg[i].cfg_umask << 8 | pmc_cfg[i].cfg_evtsel)*. Esta sentencia se encarga de leer la máscara de configuración **cfg_umask** que contiene un 1 en el índice asociado con el PMC de la métrica que queremos leer, y hacer una operación or lógica con la configuración de los eventos **cfg_evtsel**, de manera que devolverá la máscara en *raw* del evento que queremos medir, para ser pasada como parámetro a **init_counter**. Esta función es encargada de inicializar el evento hardware y de invocar la inicialización del contador *perf*.

La función **init_counter**, implementada en el módulo **pmu_config_perf.c**, es la función encargada de la inicialización de los eventos hardware dados una CPU en ejecución, un **task_struct** con información acerca del proceso de lectura, un **pmc_usrcfg_t** con la información de configuración de los contadores y una máscara de configuración **config_mask**, indicando la configuración en formato *raw* del evento *perf*. Esta función se encarga de generar una estructura contenedora de varios atributos requeridos para la inicialización. Dichos atributos son el tipo de configuración de contador (cruda, en nuestro caso, *PERF_TYPE_RAW*), si es necesario el modo de conteo del núcleo, la máscara de configuración cruda **config_mask**, y, si el modo de configuración indicado por **pmc_usrcfg_t→cfg_ebs_mode** es EBS, se asigna **event_overflow_callback** como la *callback* de desbordamiento. Esta llamada, ejecutada en contexto NMI con IRQ deshabilitado, es la encargada de rellenar la información de los contadores cuando, en modo EBS, se desborda el contador. Esta función recupera el **cpu** a través de la llamada a *smp_processor_id()*, el **task_struct** a través de la macro *current* y la estructura **pmon_prof_t** a través del **task_struct** prof=task→pmc. Se comprueba que las variables recuperadas no sean nulas y se recupera el **core_experiment_t**, ubicado en *prof→pmcs_config*. A continuación, se recupera la máscara de configuración que indica qué PMCs se han desbordado mediante la instrucción *(0x1 << core_exp→log_to_phys[core_exp→exp_idx])*. En último lugar, se invoca a la función **do_count_on_overflow**, de la cuál hablaremos a continuación.

```
void event_overflow_callback(struct perf_event *event,
#if LINUX_VERSION_CODE < KERNEL_VERSION(3, 2, 0)
                         int nmi,
#endif
                         struct perf_sample_data *data,
                         struct pt_regs *regs)
{
     uint64_t overflow_mask;
     unsigned int this_cpu=smp_processor_id();
     /* Retrieve task_struct from current macro */
     struct task_struct *p=current;
     pmon_prof_t* prof=p→pmc;
     core_experiment_t* core_exp=NULL;

     if(!prof || !prof→this_tsk→prof_enabled|| !
prof→pmcs_config)
          return;

     /* Retrieve core_experiment from pmon_prof_t */
     core_exp=prof→pmcs_config;
```

```
overflow_mask=(0x1«core_exp→log_to_phys[core_exp→ebs_idx])
;

     do_count_on_overflow(regs, overflow_mask);
}
```

Existe una función llamada **free_experiment_set** en el módulo **mc_experiments.h** que se encarga de liberar el alojamiento asociado al conjunto de experimentos PMC entrante para que, cuando se trate de un evento *perf*, recorra los contadores del array de eventos de bajo nivel **low_level_exp** y libere uno a uno los contadores *perf* a través de la invocación a **perf_event_release_kernel**.

Por último, vamos a hablar de dos funciones implementadas para la habilitación y desactivación de los contadores perf. Estas están implementadas en el módulo **pmu_config_perf.c**, y son **void perf_enable_counters(core_experiment_t *exp)** y **void perf_disable_counters(core_experiment_t *exp)**, cuya única función es recorrer todos los elementos del *buffer* **exp** e invocar a la función **perf_event_enable** o **perf_event_disable** con el parámetro **exp→array[i].event.event**, el cuál se refiere directamente a la estructura **perf_event**.

2.2 Modificaciones estructurales en código independiente de arquitectura

Ahora ha llegado el momento de explicar los cambios efectuados sobre el núcleo central de PMCtrack, el módulo **mchw_core.c**. Este módulo implementa todas las funciones relacionadas con el núcleo de PMCtrack independiente de arquitectura. Los cambios en este fichero han sido críticos, ya que no son parte del *backend* de perf, sino que constituyen aspectos independientes de arquitectura de PMCtrack.

En primer lugar, las *callbacks* han sido más sencillas ya que perf ya se ocupa del mantenimiento de los registros.

La primera función de la que vamos a hablar es **do_count_on_overflow**, la función de manejamiento de interrupciones de desbordamiento de los PMCs, encargada de leer los PMCs e insertar una muestra del PMC al *buffer* de muestras en modo EBS cuando se ha producido una interrupción. En primer lugar, adquiere el cerrojo residente en **pmon_prof_t→lock** e inicializa la estructura **pmc_sample_t**, asociada con la muestra del PMC, con una serie de parámetros: el tipo de muestra es *PMC_EBS_SAMPLE* indicando una muestra EBS, se rellena el tipo de núcleo, la máscara de PMC y el número de contadores con la información residente en **core_experiment_t**, se asigna el

número de contadores virtuales y la máscara a 0 (en modo EBS los contadores son fijos) y se asigna el pid a *current→pid*. A continuación, la función invoca a **do_count_mc_experiment_buffer**, función encargada de leer los contadores PMCs e introducir los valores devueltos en un *buffer* de muestras, en este caso el *buffer* contenido dentro de la estructura de muestra **pmc_sample_t.pmc_counts**. Por último, la función inserta el valor recuperado en el *buffer* de la hebra en ejecución y libera el cerrojo.

Continuando con el módulo independiente de arqutiectura, vamos a documentar la función de tarea diferida **deferred_read_function**, encargada de realizar la lectura de contadores en modo TBS como una tarea diferida para que no haya problemas de interbloqueos, ya que la función de la API de perf de lectura de contadores es de naturaleza bloqueante. En primer lugar, esta función se encarga de recuperar la estructura **pmon_prof_t** a través de la sentencia *cointainer_of(work, pmon_prot_t, read_counters_task*), el **task_struct** asociado con el proceso de lectura a través del atributo *prof→this_tsk* y la **cpu** en ejecución a través de la sentencia *smp_processor_id()*. A continuación, se asigna como evento un *tick* del timer (*PMC_TIMER_TICK_EVT*) y, en último lugar y siempre que el campo **prof→pmcs_config** no sea nulo, la función de tarea diferida invoca a la función **sample_counters_user_tbs** con los parámetros devueltos, encargada de recuperar una muestra de los contadores.

Como continuación, vamos a documentar la función **sample_counters_user_tbs**, encargada de leer los contadores PMCs y virtuales e insertar la muestra del PMC en el *buffer* de muestras a su debido tiempo (al final de cada intervalo de tiempo). Para el backend PMCtrack-perf ha sido necesario realizar una serie de cambios que vamos a detallar. Cesta función se compila de manera condicional asociado al símbolo de preprocesado *CONFIG_PMC_PERF*, para que se compile distinto cuando la arquitectura es perf. En esta función solo son interesantes los eventos asociados con un tick del timer o con el modo de monitorización de PMCtrack, por lo que si el evento **event** entrante como parámetro no es del tipo *PMC_TIMER_TICK_EVT* o *PMC_SELF_EVT*, se descartará la muestra. A continuación, se invoca a la función **do_count_mc_experiment**, la cual devuelve la lectura de los contadores de rendimiento asociados con la configuración PMC pasada como parámetro, y se actualizan los contadores en la hebra indicada por la estructura **prof**. Para seguir, se inicializa la muestra según el evento hardware entrante: si éste es un *tick* la muestra es *PMC_TICK_SAMPLE*, si por el contrario es *self_sample*, la muestra pasa a ser *PMC_SELF_SAMPLE*, indicando el modo de automonitorización. Para finalizar, se asignan las variables de la muestra (tipo de núcleo, máscara de PMCs, número de contadores virtuales, número de contadores, PID, etc.). Por último, se copian y limpian todas las muestras alojadas en la estructura **prof** y se invoca al módulo de monitorización a través de la función **mm_on_new_sample**. Para que funcione el modo

EBS, ha sido necesario eliminar la función **sample_counters_user_tbs_task** asociada al flag de compilación *CONFIG_PMC_PERF*, ya que dicha función es bloqueante.

Callbacks

Ahora es el momento de hablar de las funciones de *callback* residentes en el módulo **mchw_core.c**. Estas funciones son usadas por PMCtrack para dar respuesta a eventos que le llegan desde el planificador, ya que PMCtrack necesita de esta información. Estas son son **mod_save_callback_gen** y **mod_restore_callback_gen**. La reimplementación consiste en el vaciado de las funciones para el flag *CONFIG_PMC_PERF*, de manera que la funcionalidad es la misma: si el puntero **prof** o su atributo **prof→this_tsk→ prof_enabled** son nulos, retorna. Si no, invoca a **mm_on_switch_out(prof).** También hay otra función que se ha reimplementado, la función **static void mod_tbs_on_tick(void* v_prof, int cpu)**, que efectúa las mismas comprobaciones que las anteriores y encaso de éxito invoca a **mm_on_tick(prof,cpu)**, encargada de manejar los ticks enviados desde el planificador.

Para seguir, es necesario explicar los cambios realizados sobre la función invocada en la creación de un proceso, **mod_alloc_per_thread_data**, encargada del alojamiento de espacio. Se añade un fragmento de código para ejecutar la **asociación** de la función de la **tarea diferida** con el **parámetro** de **tarea diferida** añadido en la estructura **pmon_prof_t**.

A continuación vamos a hablar de la función **tbs_mode_fire_timer**, asociada con el temporizador de kernel usado para TBS. Ahora, además de su funcionalidad anterior, se encarga de **encolar la tarea diferida** en la cola de trabajos diferidos del kernel a través de la sentencia *schedule_work_on* y pasando como parámetro *prof→read_counters_task*. Para ello, es necesario un mecanismo de sincronización sobre el **task_struct** asociado con el evento. Esta tarea diferida invoca a la función de lectura de contadores de *perf* **perf_event_read_value**, una función muy peculiar. Esta función no tiene naturaleza bloqueante, pero es una función programada con muchos escenarios en mente. Un ejemplo es uno en el que se invoca desde una llamada al sistema y haya que leer los contadores desde otra CPU, lo que conllevaría a una *inter-processor interrupt*, con la finalidad de funcionar correctamente sin importar el lugar de invocación. En PMCtrack se invocaría desde el vencimiento de un timer de PMCtrack, este timer podría estar ejecutándose en una CPU distinta del proceso a monitorizar y esto provocaría una *non-maskarable interrupt* (NMI), que PMCtrack no soporta. Para solucionar este problema, se ha estudiado toda la rama de código y hemos observado que en el hipotético caso de que dicha función se ejecute en la misma CPU del proceso a monitorizar, no va a bloquear ni a hacer *inter-processor interrupt*, ya que está en la misma CPU. Para ello se han implementado dos mecanismos: en modo TBS, se ha programado la tarea diferida para que se ejecute en la misma CPU donde está el

proceso, a través de la sentencia *schedule_work_on*. En modo EBS, se ejecuta en el cotnexto de una NMI cuando devuelve el contador en la misma CPU. Aún así, existe un escenario en el cuál el proceso a monitorizar termina al tiempo en el que el planificador asigna a la tarea diferida (con la función de lectura de perf), y el task_struct a ha sido liberado. Este problema se conoce como **problema de la existencia del kernel**. Para que esto no suceda, se implementa este mecanismo de contador de referencia sober el task_struct del proceso a monitorizar, a través de las sentencias *get_task_struct* y *put_task_struct*.

Para finalizar con el módulo central, es requerido eliminar la implementación de la función encargada de detectar qué PMC ha desbordado al gestionar el handler de EBS, **update_overflow_status_non_ebs_pmcs**, ya que esta tarea está ahora a cargo de perf.

2.3 Cambios estructurales en organización de ficheros de proyecto

Para comenzar esta última sección, se empieza por describir uno de los cambios principales en la estructura del proyecto como es la creación de un **directorio** "perf" en las fuentes del módulo del kernel de PMCtrack, cuyo objetivo es el alojamiento del Makefile asociado con perf. Este directorio se encuentra en la ruta absoluta *src/modules/pmc/perf*, y en este se compilará el módulo de kernel asociado con el *backend* de perf, **mchw_perf.ko**, que deberá ser instanciado posteriormente para su ejecución. Para implementar el *Makefile* para que compilara PMCtrack con el *backend* de perf, basta con hacer *copy-paste* del Makefile de **core2**, y hacer una serie de cambios tales como que el nombre de módulo pasa a ser **mchw_perf** y los objetos a compilar son los siguientes: **mchw_core.o**, **mc_experiments.o**, **pmu_config_perf.o**, **cbuffer.o**, **monitoring_mod.o** y **syswide.o**. Para terminar con el Makefile, se añade el flag **CONFIG_PMC_PERF** a los flags extra de compilación.

Por último, se debe añadir al *script* genérico de compilación de PMCtrack **pmctrack-manager** el parámetro **perf** para que acepte *perf* como uno de los módulos compatibles para el equipo. De esta manera, cada ejecución de *pmctrack-manager* [*clean*|*build*] afectará al módulo perf.

3. Limitaciones de integración

La primera limitación y posiblemente la más difícil de afrontar ha sido la escasa documentación que ha sido posible encontrar de la API de bajo nivel de perf. Esto es consistente con el proceso de creación de *perf events*, discutido en los capítulos 2 y 3; es una API que llegó mucho más tarde que la herramienta, que ha ido por detrás y que realmente no ha sido muy utilizada, puesto que el cambio estructural de diseño que era

necesario de implementar en las herramientas de medición de contadores hardware en ese momento era más difícil de afrontar que una implementación con un diseño nuevo, lo que provocó el abandono de herramientas como *perfmon2*.

A pesar de esto, a través de la realización de un estudio exhaustivo de las fuentes del kernel de Linux [16], mostrando especial detalle en la parte asociada con las llamadas de la API de perf, ha sido posible la comprensión del funcionamiento de la API. Este estudio concluyó que la función de lectura de contadores hardware de perf **perf_event_read_value** puede ser bloqueante, como se ha explicado en la sección anterior.

Existe un modo en PMCtrack basado en eventos en lugar de en tiempo (*Event-Based Sample*), anteriormente mencionado, en el que los valores de los PMCs se recogen cuando el número de ocurrencias de un cierto evento alcanza un cierto umbral *U*. Esta función se invoca desde la API de perf en un contexto de interrupción en la que se pueden suceder algún tipo de interrupción. Esto requiere que todas las funciones en el flujo de código sean **no bloqueantes**, para que no se de el caso que sucede una interrupción dentro del código crítico de **perf_event_read_value**, se salte al código de la interrupción, y a la vuelta a la sección crítica tenemos un **problema de interbloqueo**.

Capítulo 5

Migración PMCtrack a kernel v5.4

Uno de los pilares de este proyecto ha sido la creación del soporte necesario en PMCtrack para su correcto funcionamiento en nuevas versiones. Para la compilación y carga satisfactoria del módulo del kernel de PMCtrack en el núcleo es necesario aplicar un parche sobre la versión del kernel en uso.

El principal cometido de este parche es reescribir una serie de ficheros de las fuentes del kernel (algunos como *arch/arm64/kernel/perf_event.c* o *arch/x86/events/core.c*) con el objetivo de modelar las fuentes del kernel para que soporten la compilación e integración de PMCtrack. Los principales pasos a tomar son:

- Inicialización de una interfaz para el módulo de kernel PMCtrack
- Funciones de registro de módulos
- Declaración de funciones de la API del kernel PMCtrack
- Modificación del Makefile del kernel
- Modificación del módulo **fork** del kernel
- Inclusión de notificaciones en el planificador de Linux para comunicar eventos críticos a PMCtrack, como la creación de libs, los cambios de contexto, etc.

Desde la versión 2.6 (inicio PMCtrack) hasta nuestros días no ha habido problemas de actualización del parche debido a su simplicidad, solo es necesario registrar *callbacks* en puntos específicos del planificador, que siempre tiene las mismas funciones: *tick*, *fork*, etc. El problema reside en la actualización del módulo de PMCtrack, ya que el API del kernel cambia sustancialmente. Los problemas sucedieron con la versión 4.15, donde cambió el API relacionada con los timers, crítico en PMCtrack.

Para la explicación de la primera parte, se han estudiado los cambios de este parche con la compilación condicional asociada al flag *LINUX_VERSION_CODE >= KERNEL_VERSION(4,15,0).*

En cuanto a la explicación de la segunda parte, se van a explicar los cambios tomados en el nuevo parche creado *pmctrack_linux-5.4.35_x86.patch*.

1. Adaptación módulo kernel

Ha sido necesaria la adaptación del módulo del kernel de PMCtrack para que compile contra dicho kernel, siendo tres pilares fundamentales los que han cambiado en la API del kernel: la interfaz de memoria compartida, los temporizadores del kernel (v4.15.0) y las señales (v5.4.0).

Los cambios afectan principalmente al fichero **mchw_core.c**, y se ha implementado empleando **compilación condicional** asociada a la versión del kernel. De esta manera, algunas funciones se implementan de una manera para versiones superiores a al 4.14, y de otras para las versiones inferiores al 4.14. Así, en la compilación se elige una u otra. En este apartado se van a desgranar dichas funciones.

La primera función; y quizás la más obvia de ser dependiente de versión de kernel, es la función asociada con el temporizador del kernel usado para TBS **tbs_mode_fire_timer**. Para versiones anteriores a la 4.15, el parámetro de entrada a la función es un **unsigned long**, que posteriormente se castearía de manera directa a una estructura **pmon_prof_t** a través de la instrucción *pmon_prof_t* prof = (pmon_prof_t*) data*. En el caso de versiones posteriores a la 4.14, el parámetro entrante es una estructura **struct timer_list * t**, de la cuál se puede recuperar la estructura **pmon_prof_t** a través de la instrucción *pmon_prof_t * prof = container_of(t, pmon_prof_t, timer)*. Esto se debe al cambio de la API de los timers surgido en al versión 4.15.

El segundo caso es la función **mod_alloc_per_thread_data**, invocada en la creación (*fork*) de un proceso, ya que requiere de la inicialización del temporizador. En este caso, si la versión del kernel es inferior a 4.15, la inicialización del temporizador se hace a través de la función **init_timer(&prof→timer)** y la asociación de función de temporizador a través de la instrucción *prof→timer.function = tbs_mode_fire_timer*. Si, en el otro caso, la versión es superior a 4.14, tan solo es necesario la invocación a la función **timer_setup**, pasándo por parámetro el temporizador **&prof→ timer** y la función de temporizador **tbs_mode_fire_timer**.

El tercer caso es la función **mmap_nopage**, cuyo caso es especial. Aquí hay una tripe compilación condicional: si la versión del kernel es mayor o igual a la 5.4.0, la función se declara **static vm_fault_t mmap_nopage (struct vm_fault *vmf)**, y inicializa una variable **struct vm_area_struct *vma** a través de la instrucción *vma=vmf→vma.* Si la versión es mayor o igual a la 4.14.0, la función se declara como **static int mmap_nopage (struct vm_fault *vmf)**, y recupera la variable **struct vm_area_struct** igual que en la anterior. Por último, si la versión es cualquier otra, declara la función como **static int mmap_nopage (struct vm_area_struct *vma, struct vm_fault *vmf)**,

entrando la variable **struct vm_area_struct** por parámetro en lugar de inicializarla. A partir de ahí, el cuerpo de la función es el mismo.

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 4, 0)
static vm_fault_t mmap_nopage( struct vm_fault *vmf)
{
     struct vm_area_struct *vma=vmf→vma;
#elif LINUX_VERSION_CODE >= KERNEL_VERSION(4, 14, 0)
static int mmap_nopage( struct vm_fault *vmf)
     struct vm_area_struct *vma=vmf→vma;
#else
static int mmap_nopage(struct vm_area_struct *vma, struct
vm_fault *vmf)
#endif
```

Como se ha mencionado anteriormente, no solo han cambiado los timers sino también el API de gestión de señales en el kernel. Como resultado, el último caso es la función **static void send_signal (int sig_num, struct task_struct* p)**. Aquí es dependiente de la versión del kernel la estructura encargada de almacenar la información de las señales del kernel. Para versiones posteriores o la 5.4.0, esta se hace en una estructura llamada **struct kernel_siginfo**, y su alojamiento en memoria a través de la sentencia *memset(&info, 0, sizeof(struct kernel_siginfo))*. Para otras versiones, se hace en una estructura llamada **struct siginfo**, y su alojamiento a través de *memset(&info, 0, sizeof(struct siginfo))*. El resto del cuerpo de la función es el mismo.

2. Adaptación parche

En este apartado vamos a comentar los cambios realizados sobre el parche del kernel de PMCtrack para el kernel 5.4. Para poner en antecedentes, el último parche realizado hasta ahora era para la versión 4.14. A partir de esta, han surgido una serie de cambios en el kernel Linux que han obligado a modificar una serie de instrucciones relacionadas con las señales, la gestión de la memoria compartida y la API de los timers.

El parche es muy simple, ya que su única funcionalidad es registrar *callbacks* en puntos específicos del planificador de Linux (cuando se recibe un *tick*, cuando se crea un proceso…).

El primer cambio es la inicialización del driver de PMCtrack del módulo perf en las arquitecturas ARM, a través de la función **static int __init armv8_pmu_driver_init()** en **arch/arm64/kernel/perf_event.c**.

El fichero con cambios más sensibles es el fichero relacionado con el núcleo del planificador del kernel y las llamadas al sistema relacionadas, **kernel/sched/core.c**. En versiones anteriores a la 4.15, se importa la librería <*linux/pmctrack.h*>, mientras que en versiones superiores a la 5.4 esto se hace en el fichero **kernel/sched.h**. Para versiones superiores a la 5.4, en la función **static struct rq *finish_task_switch(struct task_struct *prev)**, se debe invocar a las funciones *vtime_task_switch(prev)*, *perf_event_task_sched_in(prev, current)* y *finish_task(prev)*, mientras que para las anteriores a la 4.15 es suficiente con invocar a *smp_mb_after_unlock_lock()*. Para continuar, en versiones anteriores a la 4.15, en la función **static void __sched_ notrace __schedule(bool preempt)** se debe invocar a la función *finish_lock_switch()*. Esto no es necesario para versiones posteriores a la 5.4, puesto que en estas en la función **finish_task_switch** se ha invocado a *kcov_finish_switch(current)*.

Capítulo 6

Evaluación experimental

En este capítulo se discuten los resultados experimentales empleados para validar el correcto funcionamiento del nuevo *backend* de PMCtrack que hace uso del API de perf, y evaluar la sobrecarga introducida en la lectura de los contadores.

Denotaremos perf cuando hablemos de la herramienta de espacio de usuario *perf*, PMCtrack-perf para referirnos al *backend* de dicha API en la herramienta de monitorización PMCtrack y PMCtrack-legacy para referirnos a la lectura de contadores a través de PMCtrack y el *backend* de la arquitectura x86.

Se han preparado una serie de *scripts* para realizar experimentos sobre ambas APIs y efectuar la lectura de tiempos, para poder analizar la diferencia. Dichos *scripts* ejecutan 52 *benchmarks* de las suites SPEC CPU2006 y CPU2017 y recaban 12 métricas de rendimiento distintas para cada aplicación a lo largo del tiempo durante 60 segundos. Por una parte, efectúa mediciones en modo TBS a través de perf, de PMCtrack-perf y de PMCtrack-legacy y recoge los resultados. Por otra parte, efectúa mediciones en modo EBS a través de PMCtrack-perf y de PMCtrack-legacy.

Las métricas que se monitorizan son las siguientes:

- Instrucciones por ciclo (IPC).
- Instrucciones de salto por cada mil instrucciones (BRKINSTR).
- Predicciones de salto fallidas por cada mil instrucciones (BRMPKINSTR).
- Fallos en último nivel de caché cada mil instrucciones (LLCMPKI).
- Fallos en último nivel de caché cada mil ciclos (LLCMPKC).
- Accesos a la caché de último nivel cada mil instrucciones (LLCRPKI).
- Accesos a la caché de último nivel cada mil ciclos (LLCRPKC).
- Tasa de fallos en último nivel de caché (LLCMR).
- Fallos en DTLB por cada millón de instrucciones (MDTLBPMI).
- Fallos en ITLB por cada millón de instrucciones (MITLBPMI).

Las pruebas se ejecutan en una máquina con las siguientes características: en cuanto al hardware, se trata de una máquina UMA con un procesador Intel® Xeon® Gold 6138 CPU @ 2.00 Ghz de 20 núcleos, con una caché de último nivel de 28MBs conjunta, así como cada core tiene dos niveles privados de caché con 32KB y 1024KB. La configuración software es la siguiente: se ejecuta un sistema operativo Debian GNU/Linux 10 (buster) con una versión de kernel linux-5.4.35.

1. Validación de gráficas de precisión PMCtrack-perf vs perf

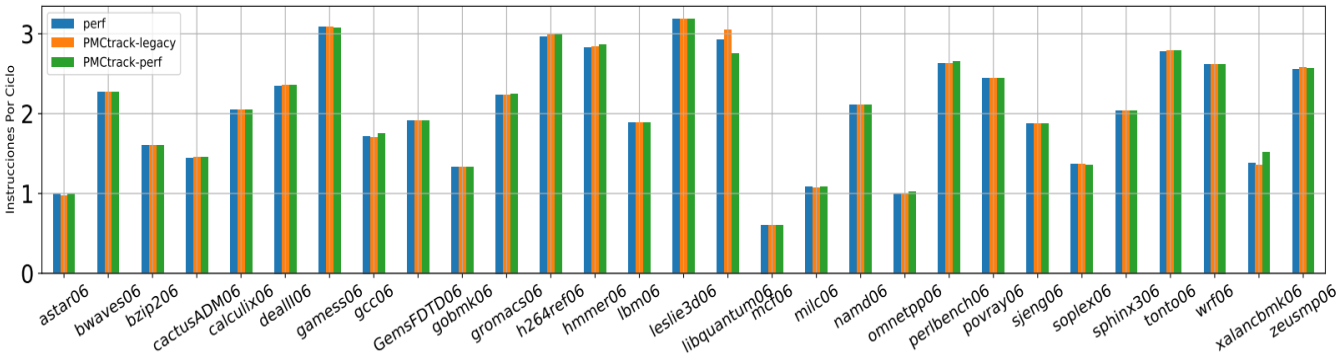
En dicha máquina se ha instalado PMCtrack, siguiendo la guía de instalación descrita en el Github oficial de la herramienta [12].

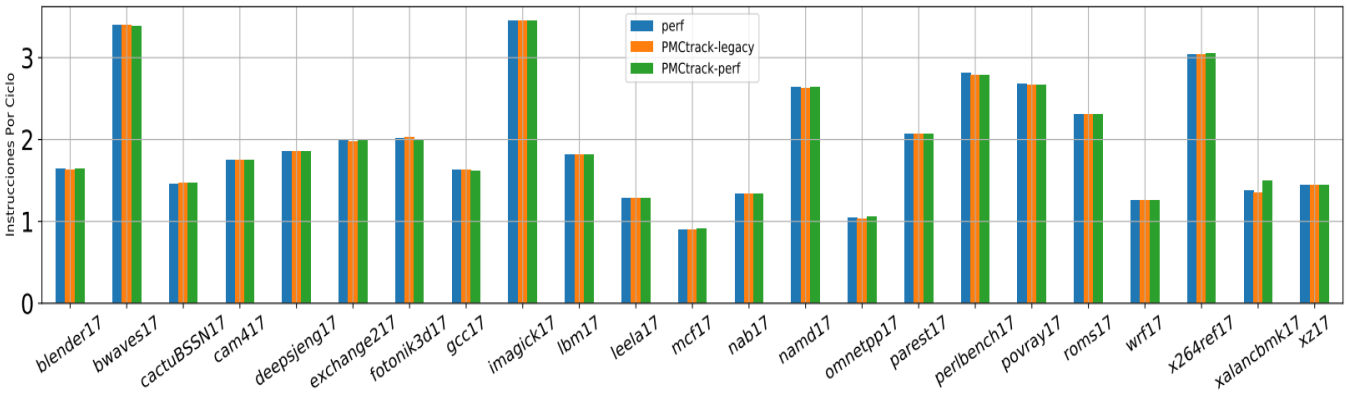
1.1 Validación de gráficas en modo TBS

En primer lugar, se va a hacer una comparación del modo TBS (*Time-Based Sampling*) con los tres métodos (PMCtrack-legacy, PMCtrack-perf y perf). Para cada una de las métricas elegidas se van a ejecutar todos los benchmarks disponibles, es decir, 29 benchmarks de SPEC CPU 2006 y 23 de SPEC CPU 2017. De todas las métricas mencionadas en el comienzo del capítulo, se van a estudiar en detalle, y por simplicidad, las siguientes:

- Número de Instrucciones por Ciclo (IPC).
- Fallos en último nivel de caché cada mil instrucciones (LLCMPKI).
- Saltos por cada mil instrucciones (BRKINSTR).

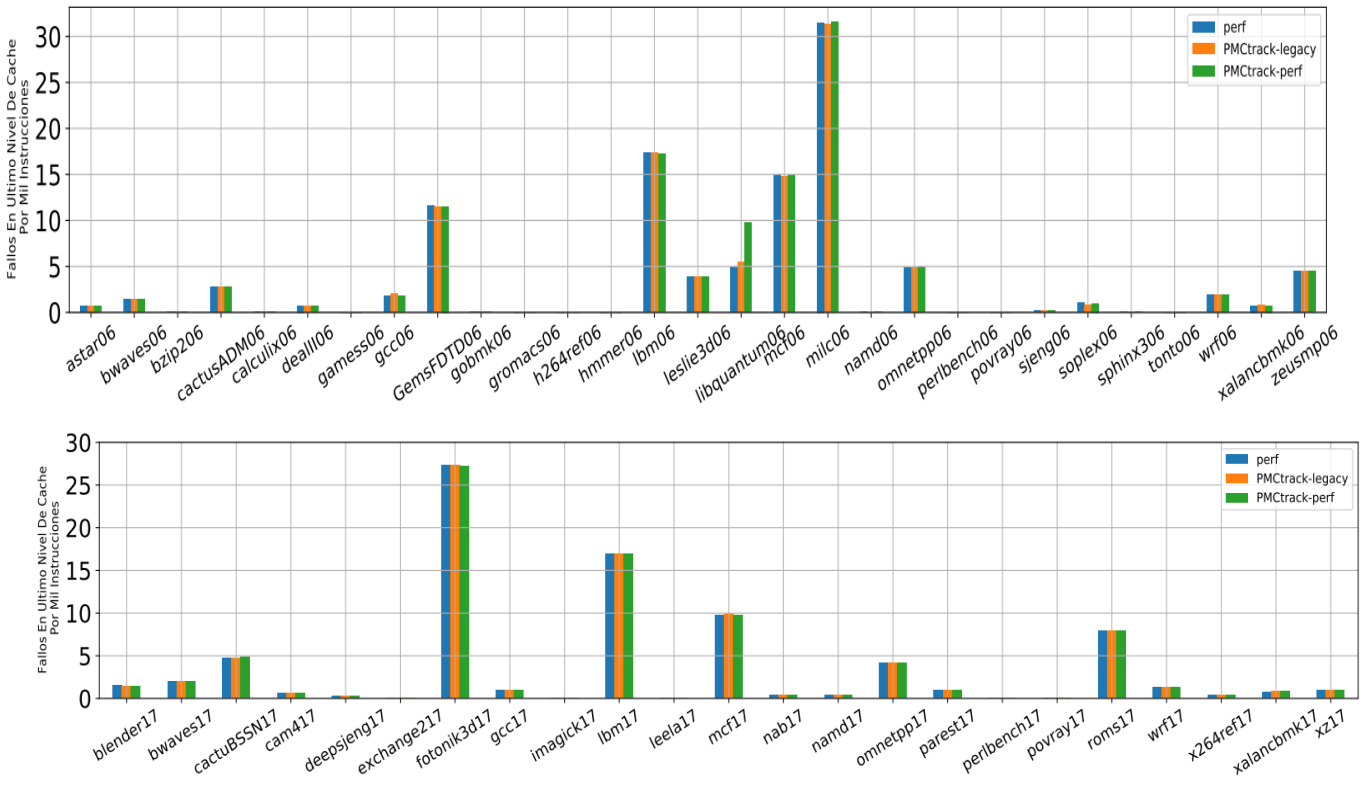
No se muestran todas las gráficas de todas las muestras, ya que se ha observado la misma tendencia de similitud que en las métricas expuestas. Por cuestiones de espacio y de mantener focalizada la discusión, no se incluyen las gráficas de todos los experimentos, aunque sí se han recabado los datos.





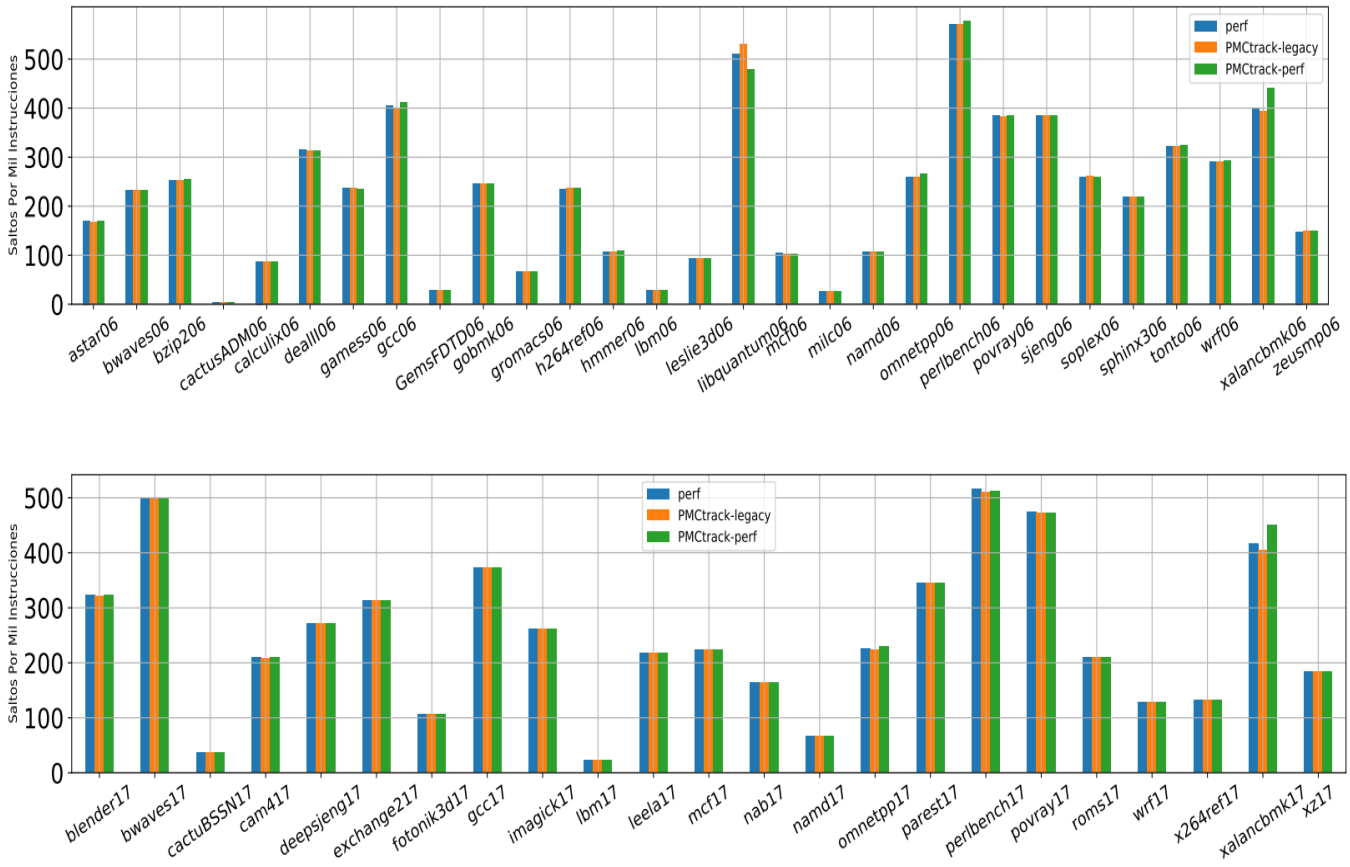
Promedio de Instrucciones por Ciclo obtenido con Time-Based Sampling (TBS) para los SPEC CPU 2006 (arriba) y SPEC CPU 2017 (abajo)

La información proporcionada por estas gráficas es plana y clara: no se observa diferencia en las mediciones del número de instrucciones por ciclo según el método de medición. Se observa disparidad en los valores obtenidos para el benchmark ***leslie3d06***, siendo el IPC superior para PMCtrack-legacy y menor para el *backend* de PMCtrack-perf. Hemos observado que esto se debe a la variabilidad que se produce a lo largo de distintas ejecuciones del benchmark, y no tiene nada que ver con la herramienta utilizada para capturar los valores de los contadores.



Promedio de Fallos en Último nivel de Caché Por Mil Instrucciones obtenido con Time-Based Sampling (TBS) para los SPEC CPU 2006 (arriba) y SPEC CPU 2017 (abajo)

Estas gráficas son claras y planas: no se observa disparidad en las métricas obtenidas según la herramienta utilizada para realizar la lectura sobre los contadores hardware.

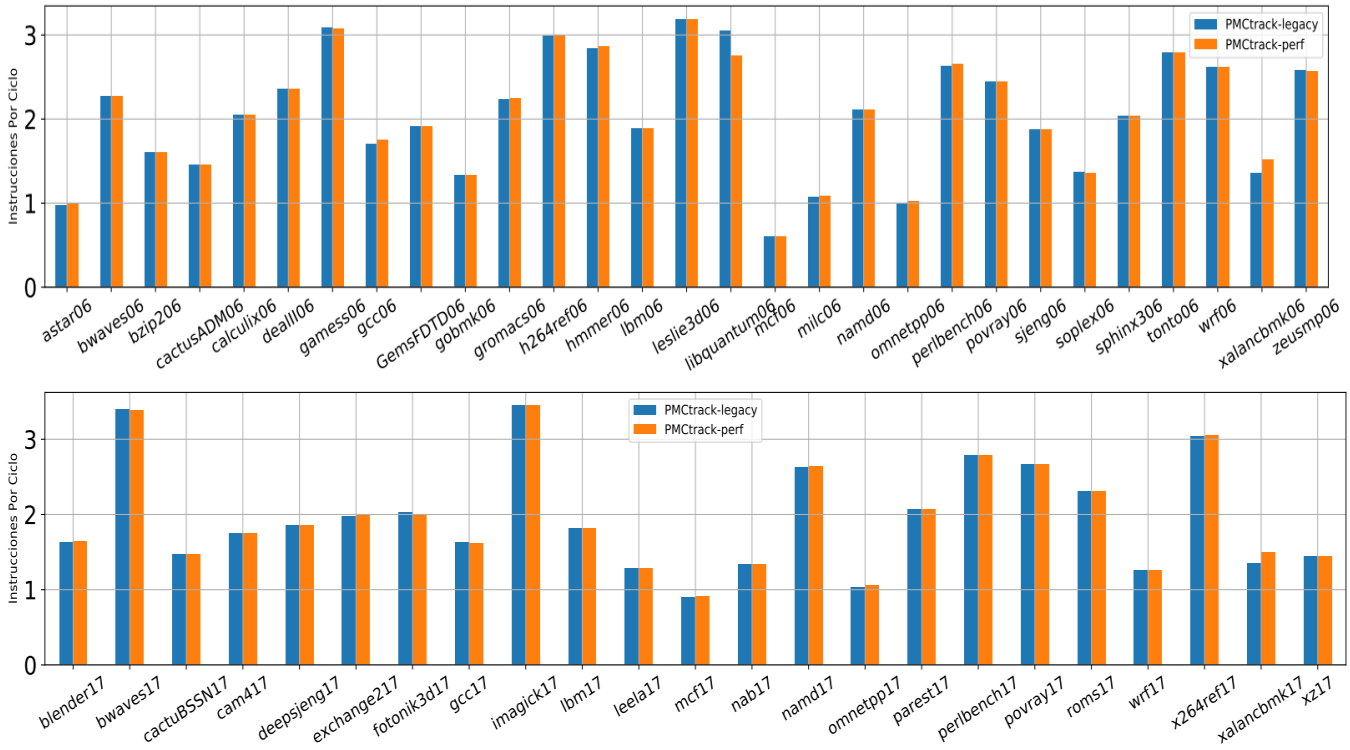


Promedio de Instrucciones de Salto Por Mil Instrucciones obtenido con Time-Based Sampling (TBS) para los SPEC CPU 2016 (arriba) y SPEC CPU 2017 (abajo)

La información concluyente de las otras tres métricas refuerza las conclusiones obtenidas: no hay disparidad en el valor devuelto por las métricas en función de la herramienta utilizada para la lectura, tan solo se observa una ligera diferencia en los benchmarks ***leslie3d06*** y ***xalancbmk***, que se puede achacar a la variabilidad en las ejecuciones del algoritmo, ya que solo hemos llevado a cabo una ejecución.

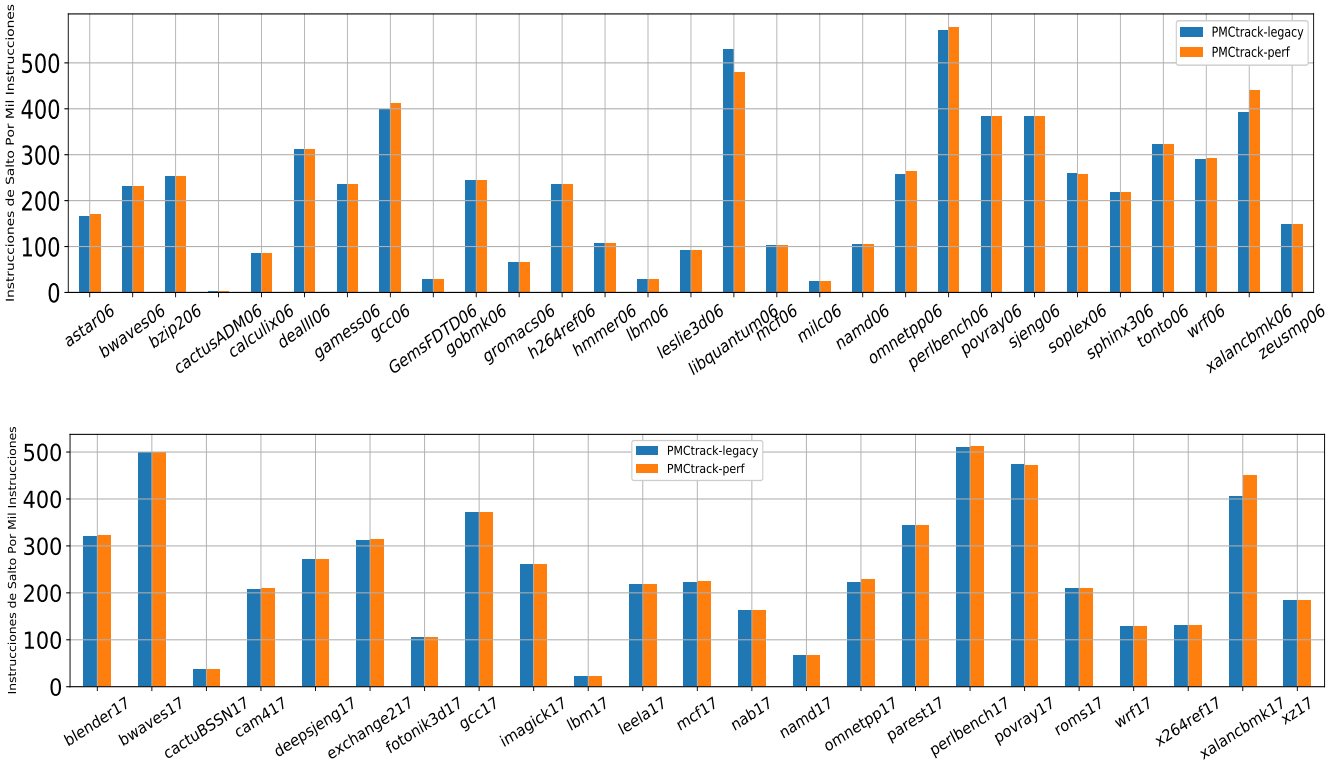
1.2 Validación de gráficas en modo EBS

Para continuar, va a tener lugar una comparativa de los valores obtenidos con el modo EBS (*Event-Based Sampling*), con los dos métodos que soportan este modo: PMCtrack-legacy y PMCtrack-perf. Para cada una de las métricas elegidas se van a ejecutar 29 benchmarks de SPEC CPU 2006 y 23 de SPEC CPU 2017. De todas las métricas mencionadas en el comienzo del capítulo, se van a estudiar en detalle las mismas que en el apartado anterior.



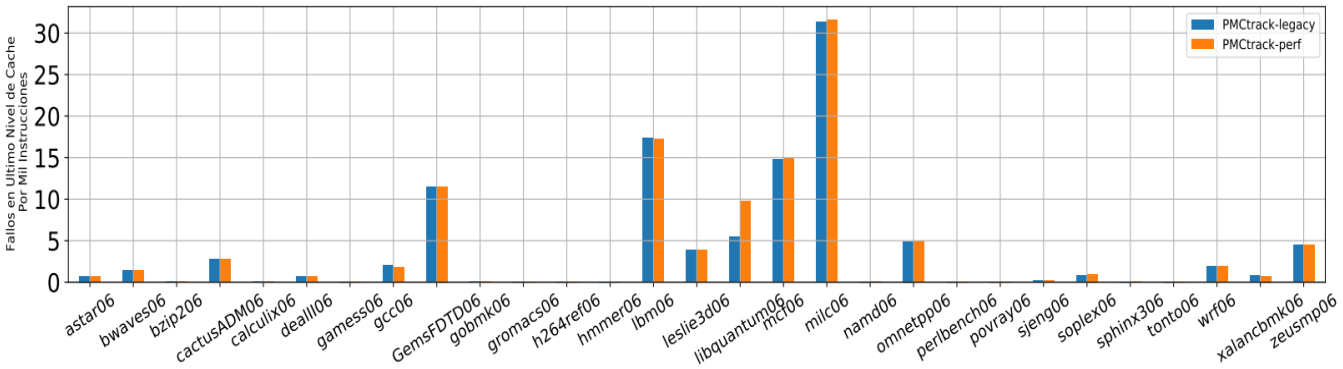
Promedio de Instrucciones por Ciclo obtenido con Event-Based Sampling (EBS) para los SPEC CPU 2006 (arriba) y SPEC CPU 2017 (abajo)

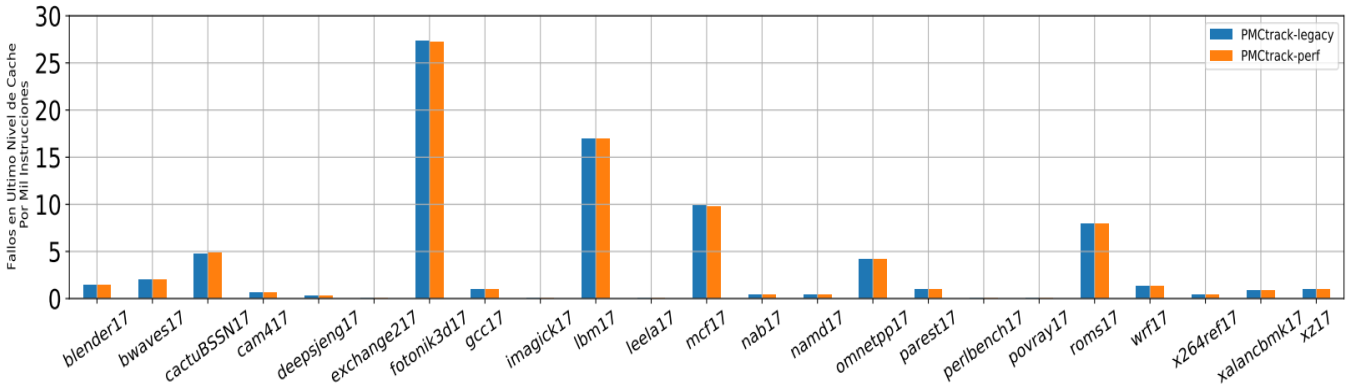
Estas gráficas indican el número de instrucciones que han sido ejecutadas por cada ciclo de ejecución del procesador (IPC) sobre los 29 benchmarks de SPEC CPU2006 y CPU2017 en modo EBS. De estas gráficas podemos concluir que los dos métodos producen un IPC muy parecido para el mismo benchmark, lo cuál era de esperar. Se observa un pico en el método ***leslie3d06***, siendo el IPC superior para PMCtrack-legacy y menor para el backend de PMCtrack-perf. Como hemos concluido en apartados anteriores, se debe a variabilidad que se produce a lo largo de distintas ejecuciones del benchmark, y no tiene nada que ver con la herramienta utilizada para realiar las lecturas.



Promedio de Instrucciones de Salto Por Mil Instrucciones obtenido con Event-Based Sampling (EBS) para los SPEC CPU 2006 (arriba) y SPEC CPU 2017 (abajo)

La información es consecuente con lo observado anteriormente: solo existe variabilidad no despreciable para el *benchmark* ***leslie3d06*** y se puede achacar a variabilidad en distintas ejecuciones del algoritmo.





Promedio de Fallos en Último nivel de Caché Por Mil Instrucciones obtenido con Event-Based Sampling (EBS) para los SPEC CPU 2006 (arriba) y SPEC CPU 2017 (abajo)

Esta gráfica indica el número de fallos en la caché de último nivel sucedidos por cada mil instrucciones (LLCMPKI) sobre los 29 benchmarks de CPU SPEC 2006 en modo EBS. En el benchmark ***leslie3d06*** la diferencia de mediciones entre PMCtrack-perf y PMCtrack-legacy es notable, siendo un 96% más en PMCtrack-perf que en PMCtrack-legacy.

1.3 Validación de gráficas de los Benchmarks

Para finalizar, tiene lugar un estudio individualizado de los benchmarks, guardando el resultado de cada métrica para cada uno de los 3 métodos. Para cada benchmark, se han recogido al rededor de 300 muestras en las que se indica el valor de todas las métricas. Se van a realizar las mismas mediciones sobre los tres métodos (PMCtrack-legacy, PMCtrack-perf y perf) en modo TBS, y sobre los métodos PMCtrack-legacy y PMCtrack-perf en modo EBS.
En modo TBS se ejecutan 300 mediciones realizadas cada 200 milisegundos, lo que no es crítico para perf ya que en al documentación se menciona que el *overhead* aparece a partir de 100ms. En modo EBS se ejecutan mediciones cada 50 millones de instrucciones.

Las métricas elegidas para el estudio han sido las siguientes:

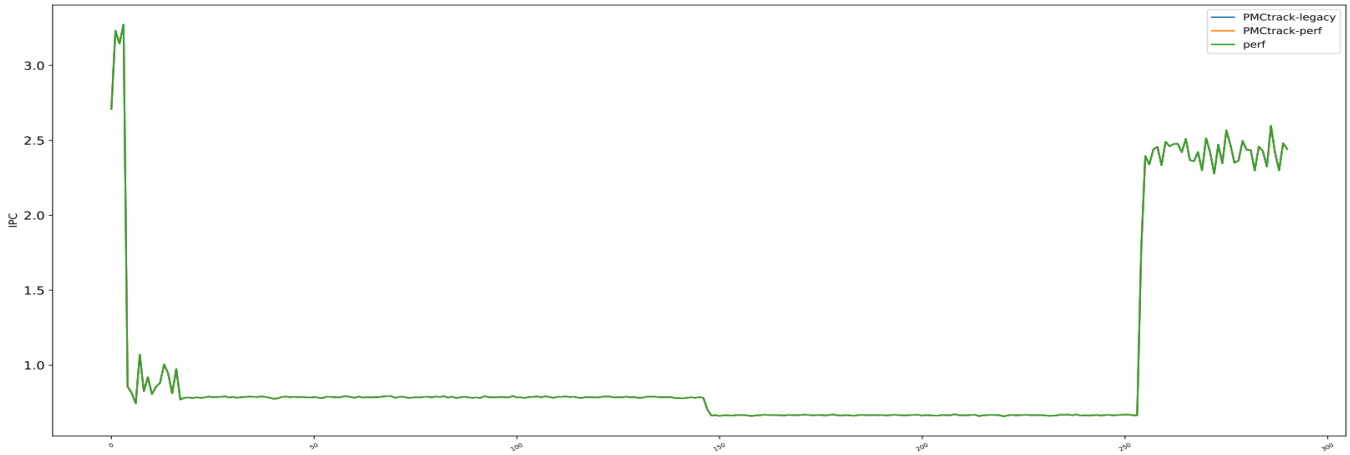
- Número de Instrucciones por Ciclo (IPC).
- Fallos en último nivel de caché cada mil instrucciones (LLCMPKI).
- Saltos por cada mil instrucciones (BRKINSTR).
- Fallos en DTLB por cada Millón de instrucciones (DTLB).

Los benchmarks seleccionados para el estudio han sido los sigueintes:

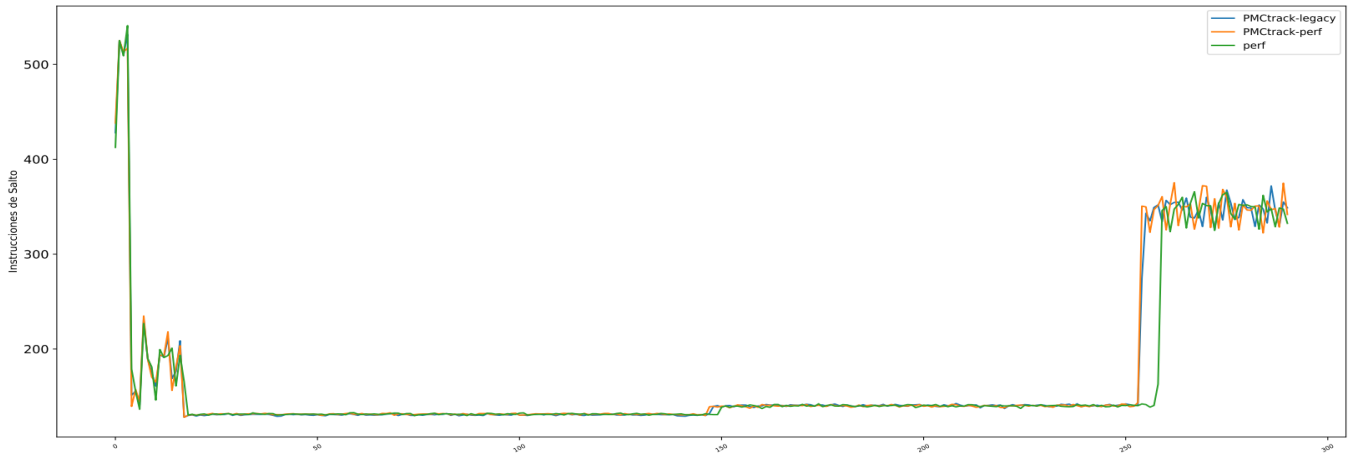
- CPU SPEC 2006-Astar.
- CPU SPEC 2017-Lbm.
- CPU SPEC 2017-Xalancbmk.
- CPU SPEC 2017-Fotonik3d.

1.3.1 Gráficas TBS

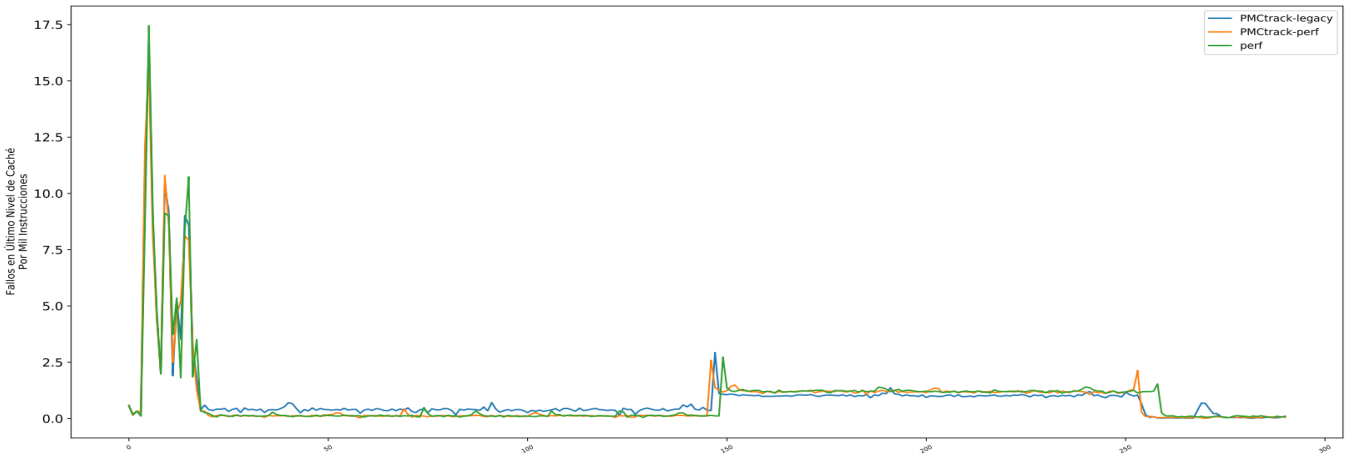
Astar06



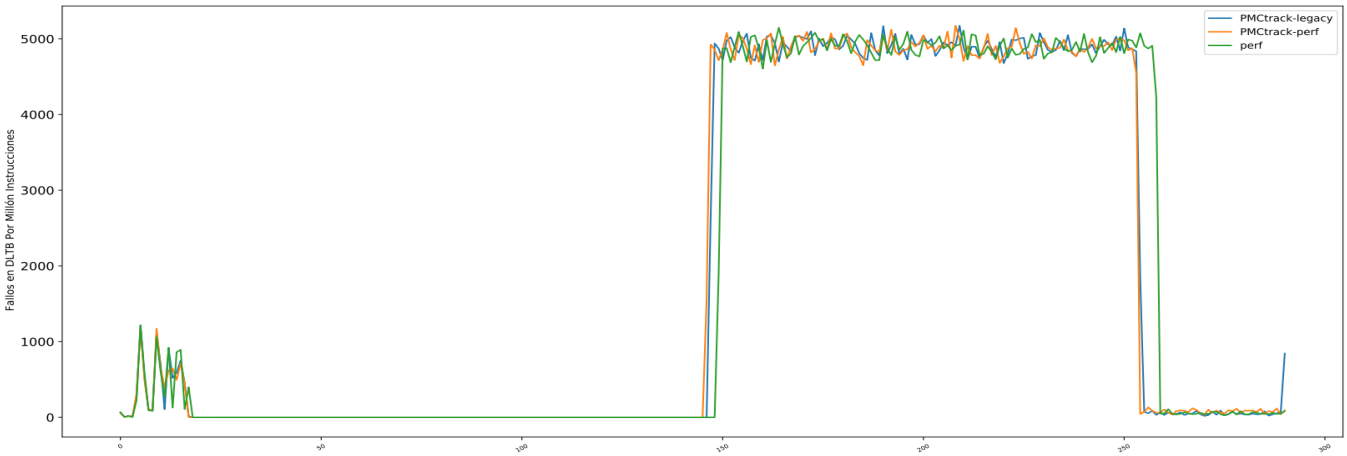
Instrucciones por Ciclo en el tiempo obtenido con Time-Based Sampling (TBS) para *astar* de SPEC CPU 2006



Instrucciones de Salto Por Mil Instrucciones en el tiempo obtenido con Time-Based Sampling (TBS) para *astar* de SPEC CPU 2006

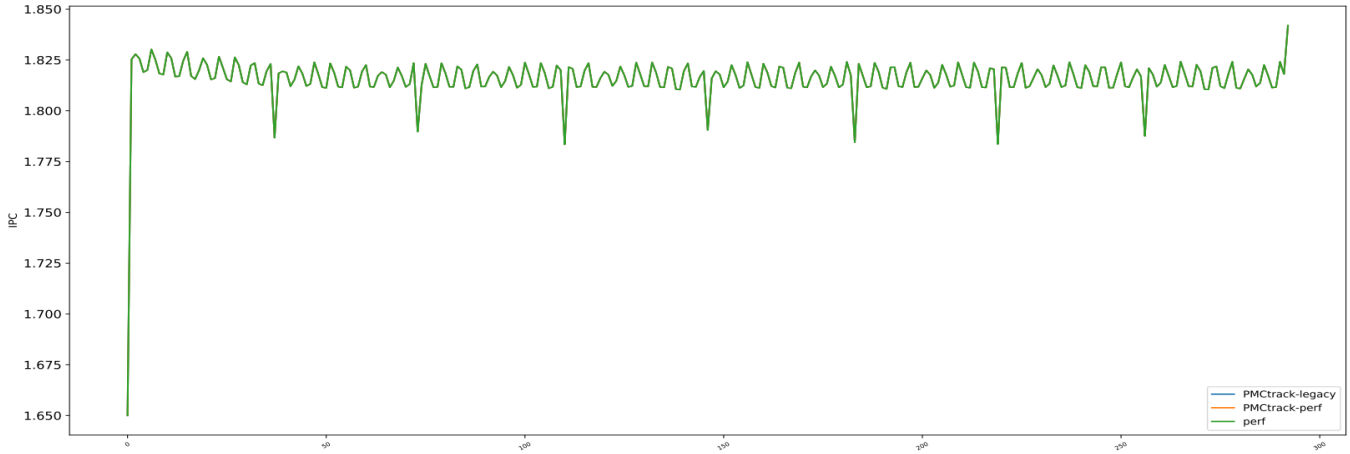


Fallos en Último Nivel de Caché Por Mil Instrucciones en el tiempo obtenido con Time-Based Sampling (TBS) para *astar* de SPEC CPU 2006

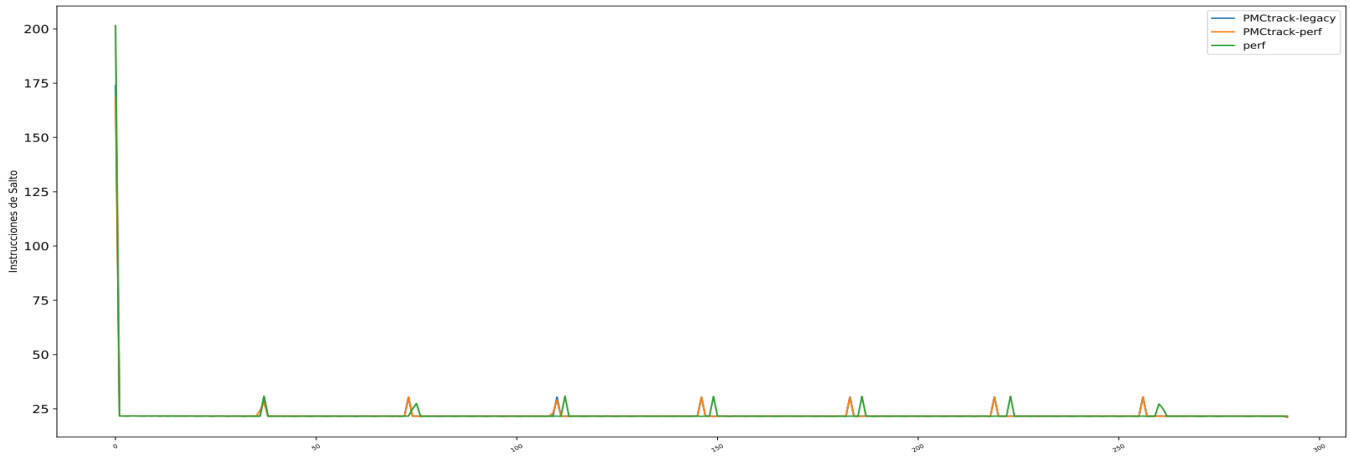


Fallos en DTLB Por Millón Instrucciones en el tiempo obtenido con Time-Based Sampling (TBS) para *astar* de SPEC CPU 2006

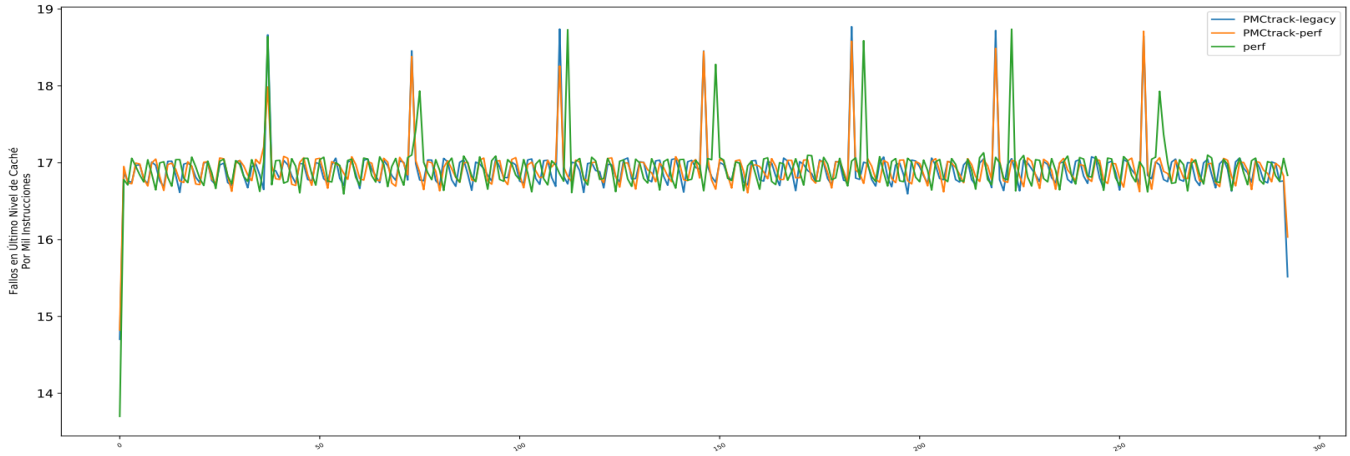
LBM17



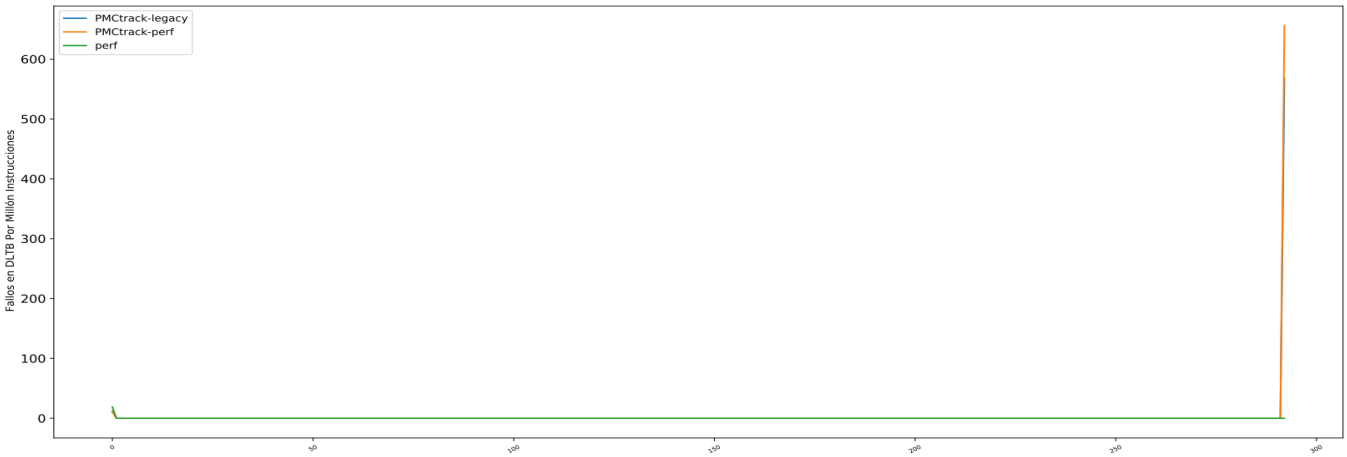
Instrucciones por Ciclo en el tiempo obtenido con Time-Based Sampling (TBS) para *lbm* de SPEC CPU 2017



Instrucciones de Salto Por Mil Instrucciones en el tiempo obtenido con Time-Based Sampling (TBS) para *lbm* de SPEC CPU 2017

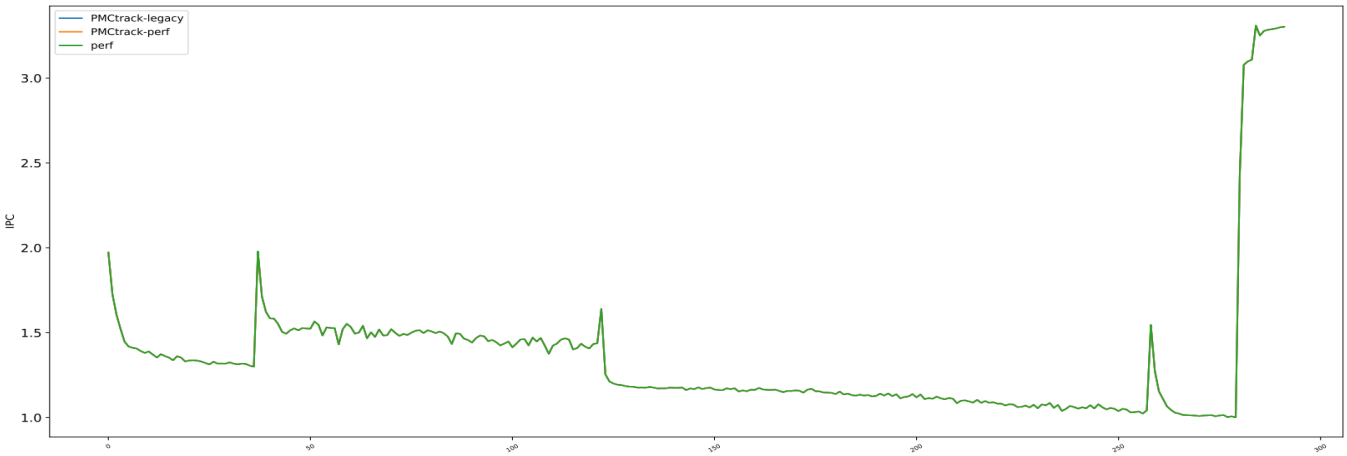


Fallos en Último Nivel de Caché Por Mil Instrucciones en el tiempo obtenido con Time-Based Sampling (TBS) para *lbm* de SPEC CPU 2017

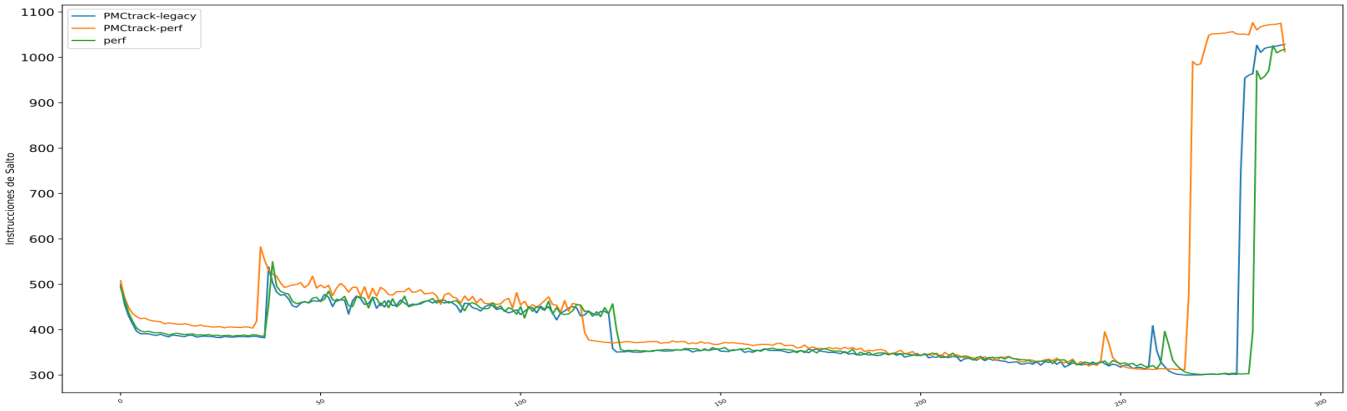


Fallos en DTLB por Millón de instrucciones en el tiempo obtenido con Time-Based Sampling (TBS) para *lbm* de SPEC CPU 2017

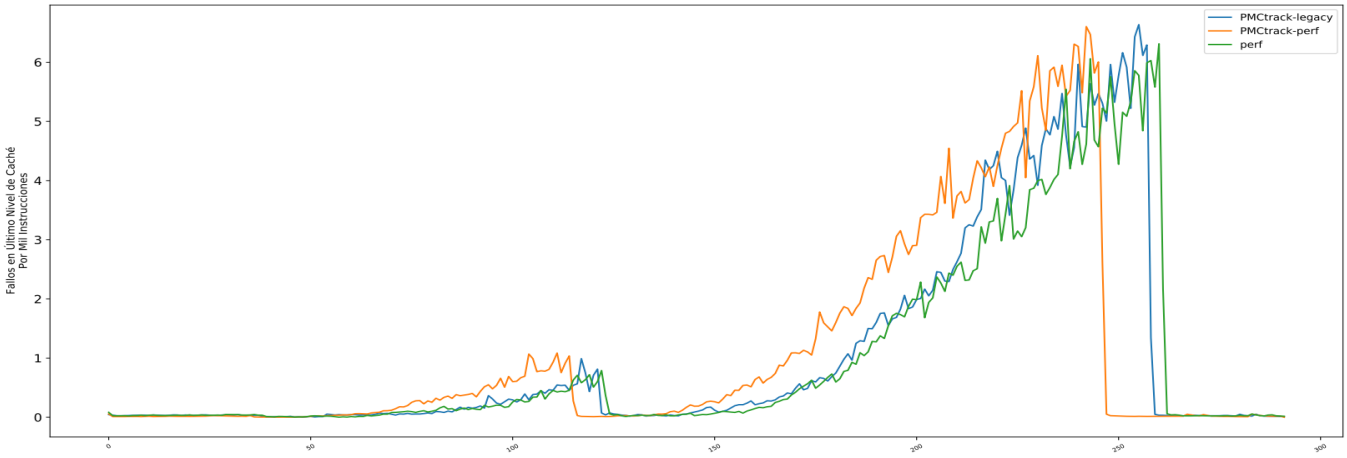
Xalancbmk17



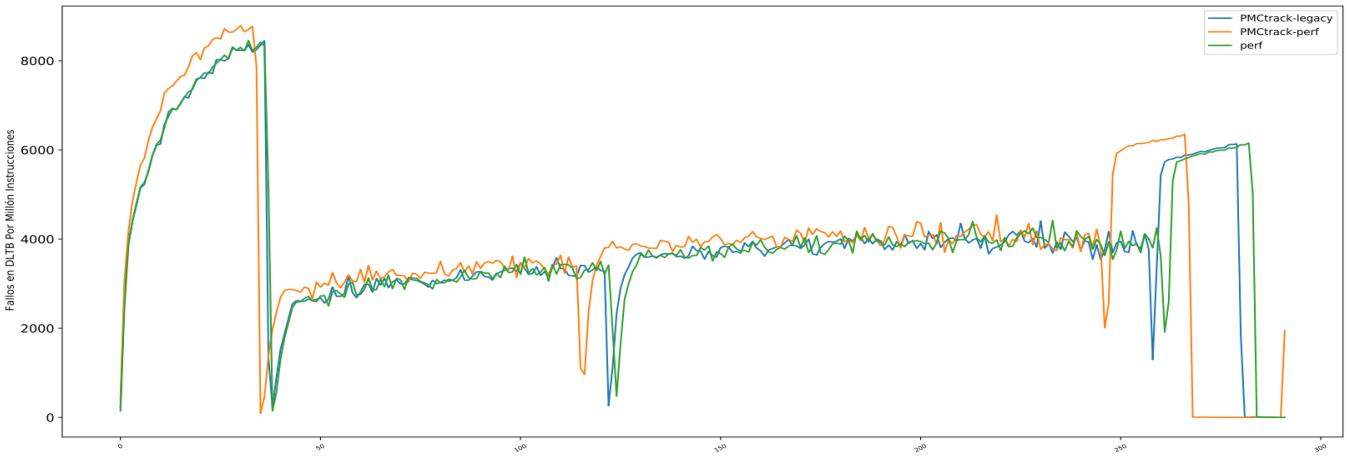
Instrucciones por Ciclo en el tiempo obtenido con Time-Based Sampling (TBS) para *xalancbmk* de SPEC CPU 2017



Instrucciones de Salto Por Mil Instrucciones en el tiempo obtenido con Time-Based Sampling (TBS) para *xalancbmk* de SPEC CPU 2017

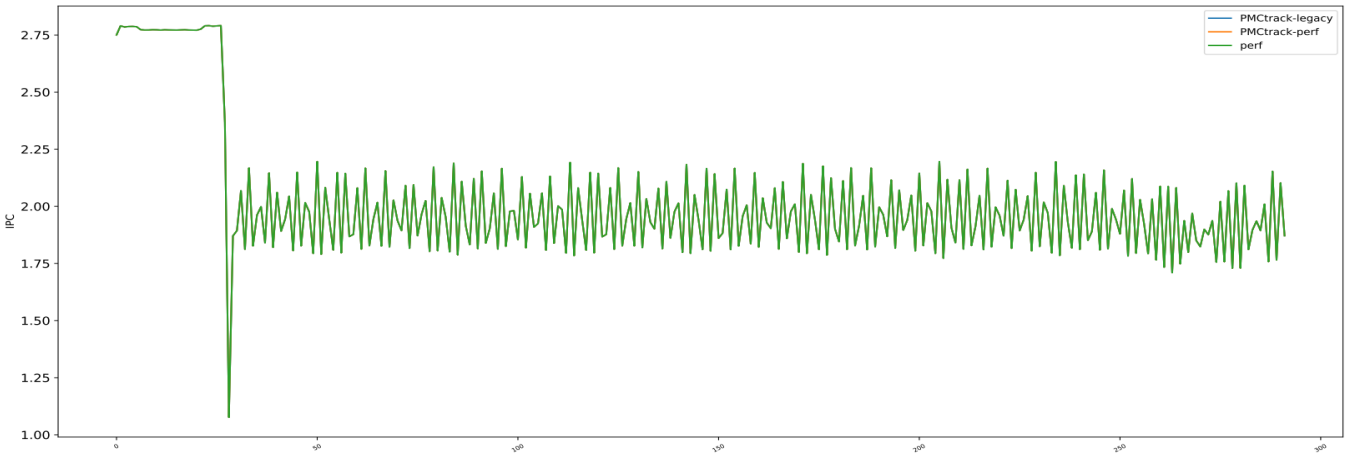


Fallos en Último Nivel de Caché Por Mil Instrucciones en el tiempo obtenido con Time-Based Sampling (TBS) para *xalancbmk* de SPEC CPU 2017

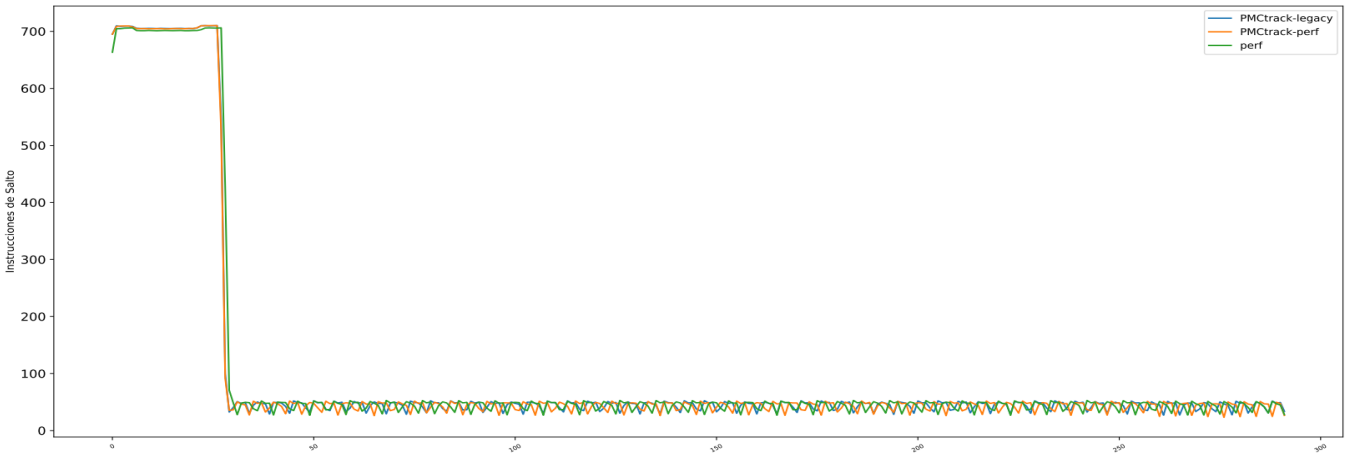


Fallos en DTLB Por Millón Instrucciones en el tiempo obtenido con Time-Based Sampling (TBS) para *xalancbmk* de SPEC CPU 2017

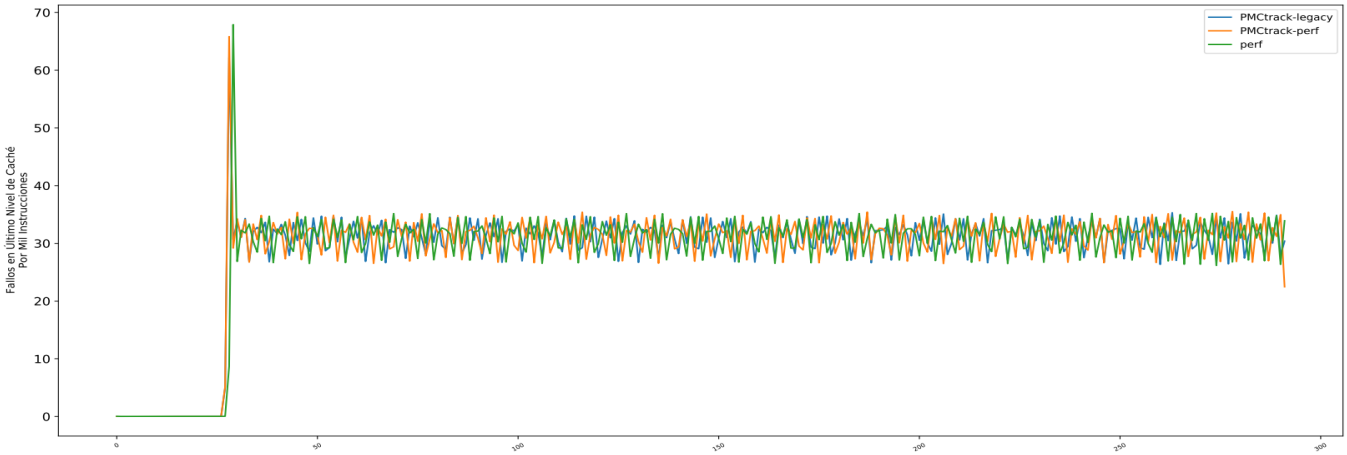
Fotonik3d



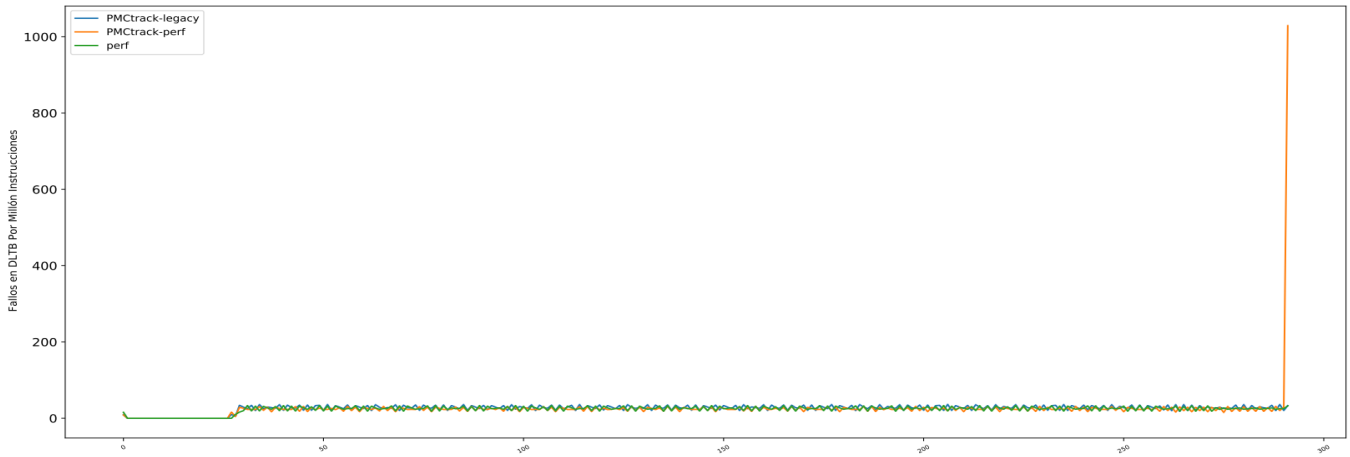
Instrucciones por Ciclo en el tiempo obtenido con Time-Based Sampling (TBS) para *fotonik3d* de SPEC CPU 2017



Instrucciones de Salto Por Mil Instrucciones en el tiempo obtenido con Time-Based Sampling (TBS) para *fotonik3d* de SPEC CPU 2017



Fallos en Último Nivel de Caché Por Mil Instrucciones en el tiempo obtenido con Time-Based Sampling (TBS) para *fotonik3d* de SPEC CPU 2017



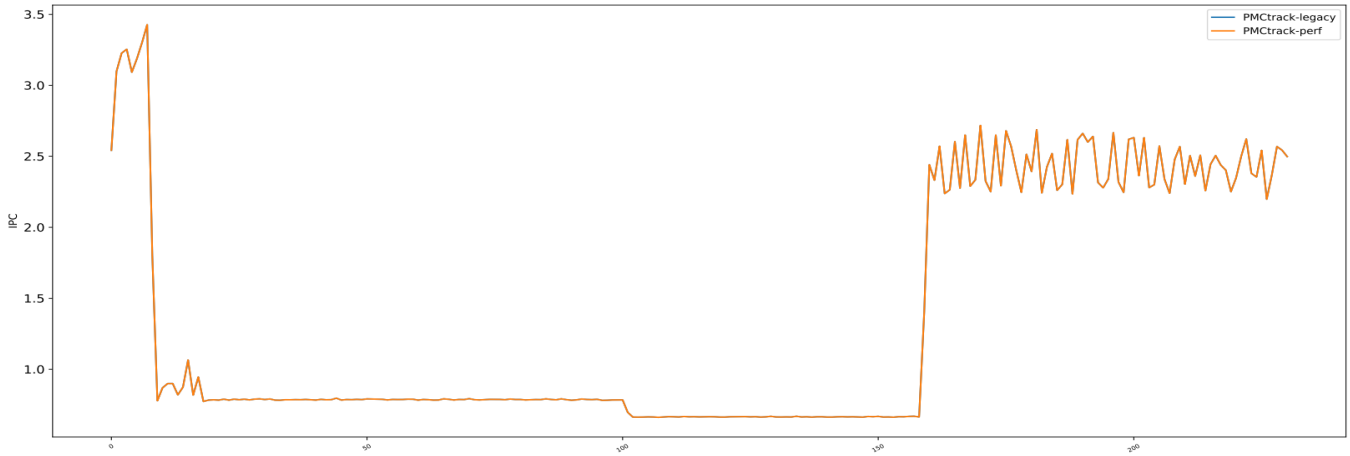
Fallos en DTLB Por Mil Instrucciones en el tiempo obtenido con Time-Based Sampling (TBS) para *fotonik3d* de SPEC CPU 2017

En las gráficas de esta sección se observa una peculiaridad: en primer lugar aparecen las mediciones de perf, en segundo lugar las mediciones efectuadas a través de PMCtrack-perf y, por último, PMCtrack-legacy.

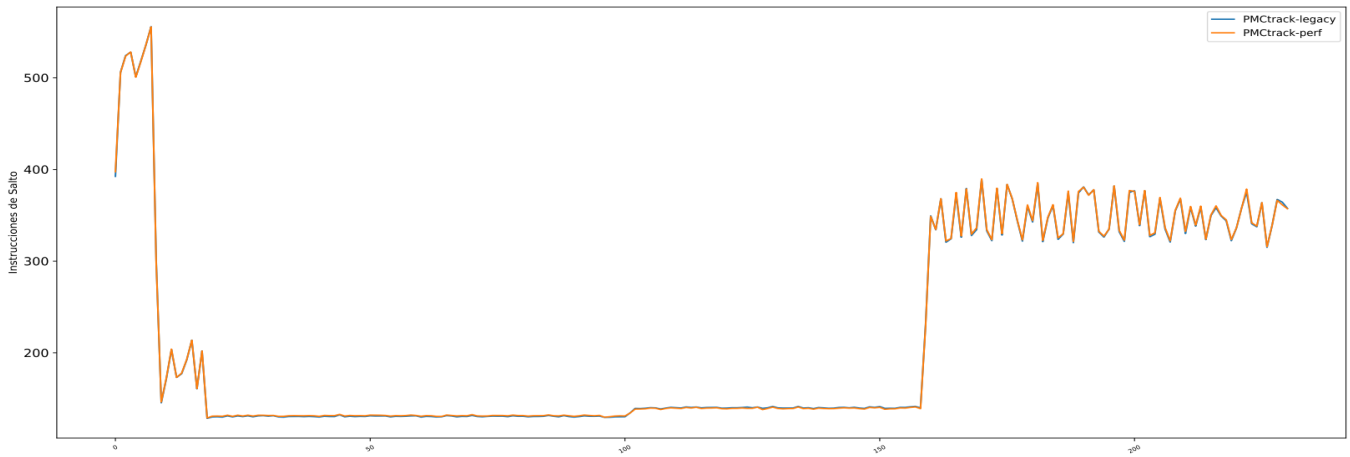
Esto tiene una explicación: PMCtrack utiliza temporizadors del kernel normales que funcionan a nivel de *tick*, sin especificar menos de 4 milisegundos. Perf, sin embargo, utiliza *timers* de alta resolución (*High Resolution Timers*) con precisión a nivel de nanosegundos, por lo que es de esperar que las mediciones devueltas por perf sean más rápidas. Además, la integración PMCtrack-perf utiliza tareas diferidas, donde hasta que el planificador no introduce el *kworker* a realizar el trabajo no se va a realizar la lectura.

1.3.2 Gráficas EBS

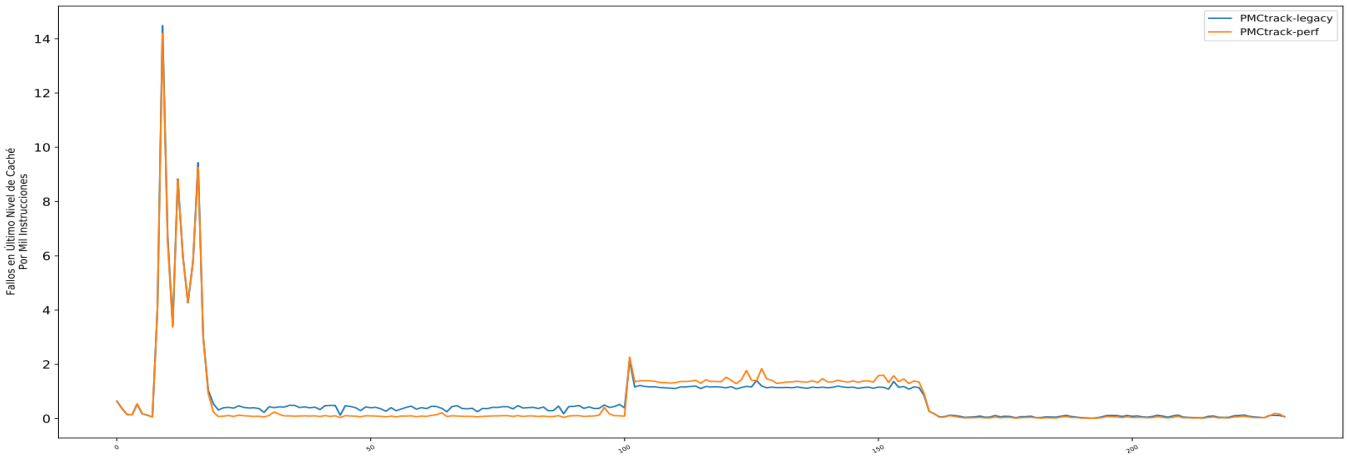
Astar06



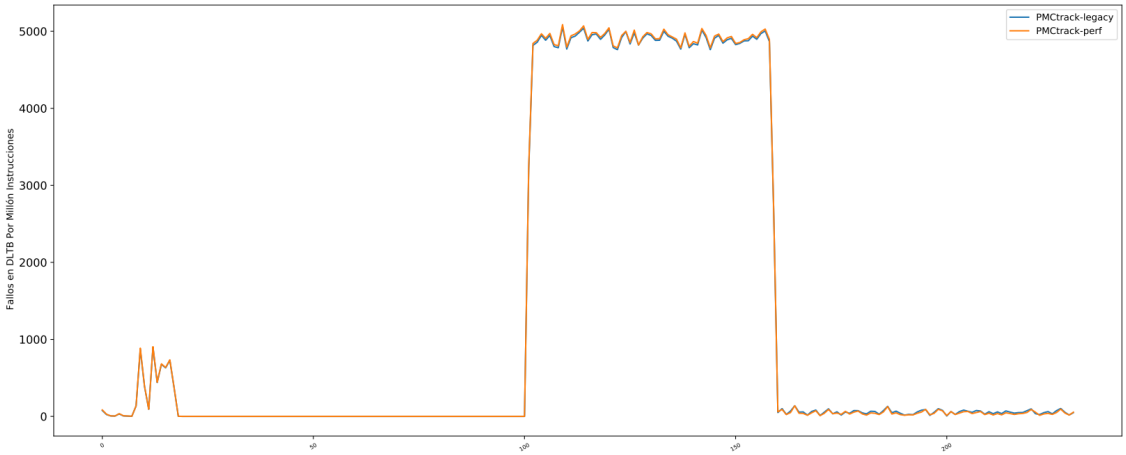
Instrucciones por Ciclo en el tiempo obtenido con Event-Based Sampling (EBS) para *astar* de SPEC CPU 2006



Instrucciones de Salto Por Mil Instrucciones en el tiempo obtenido con Event-Based Sampling (EBS) para *astar* de SPEC CPU 2006

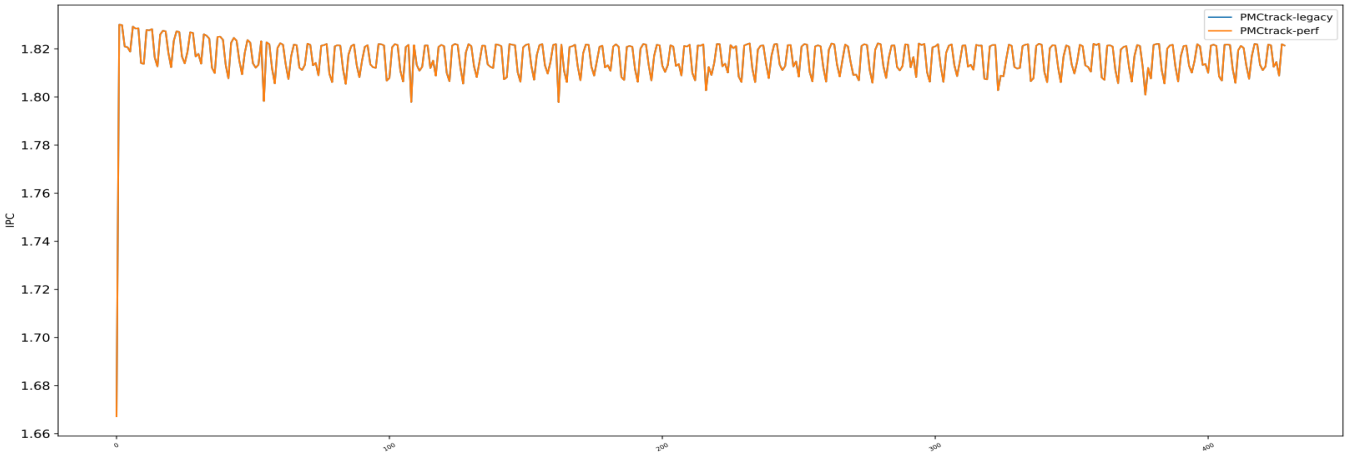


Fallos en Último Nivel de Caché Por Mil Instrucciones en el tiempo obtenido con Event-Based Sampling (EBS) para *astar* de SPEC CPU 2006

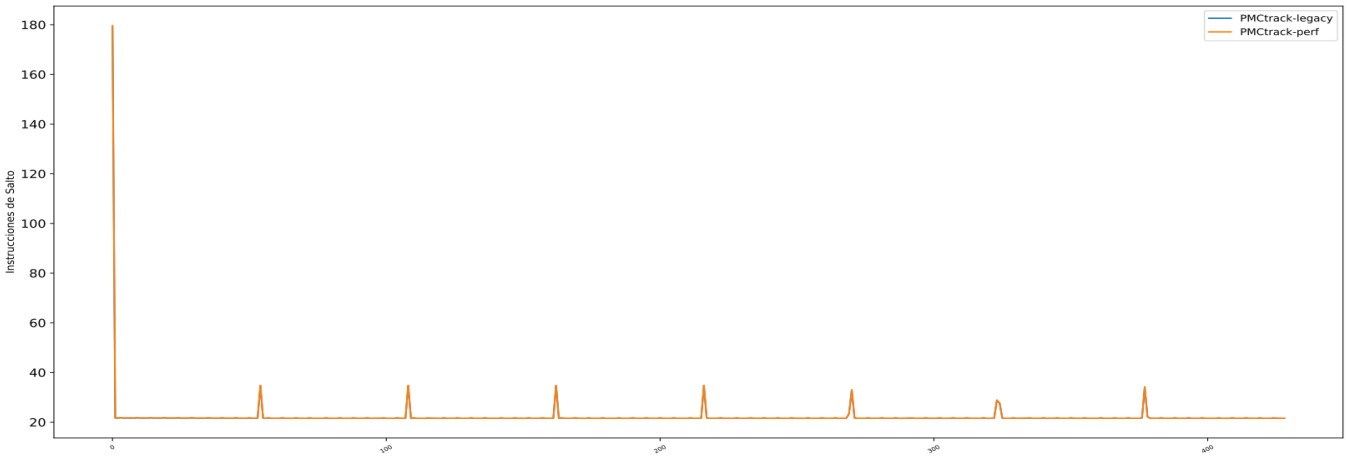


Fallos en DTLB Por Mil Instrucciones en el tiempo obtenido con Event-Based Sampling (EBS) para *astar* de SPEC CPU 2017

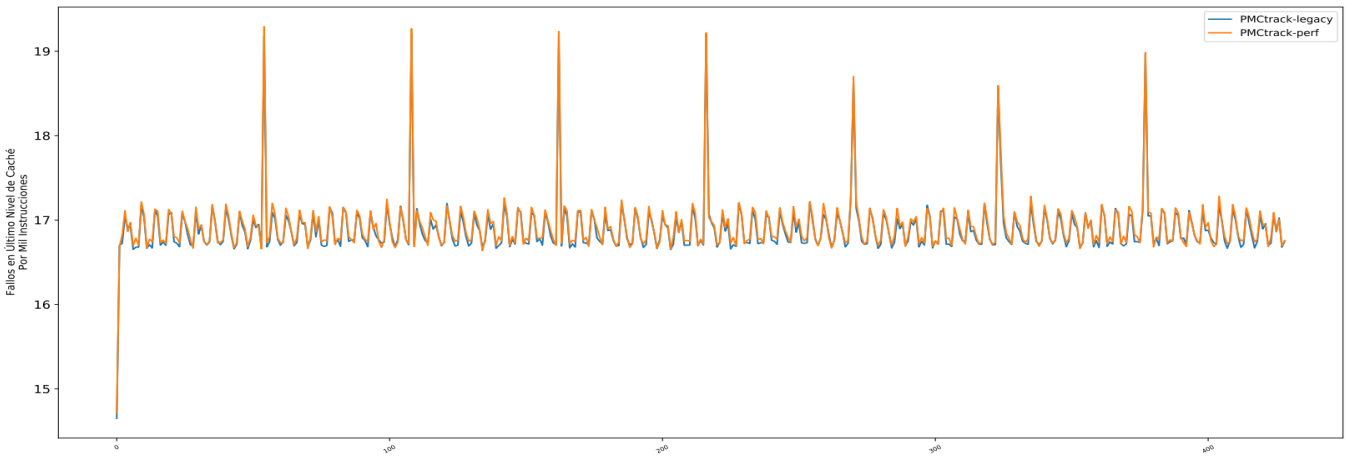
Lbm17



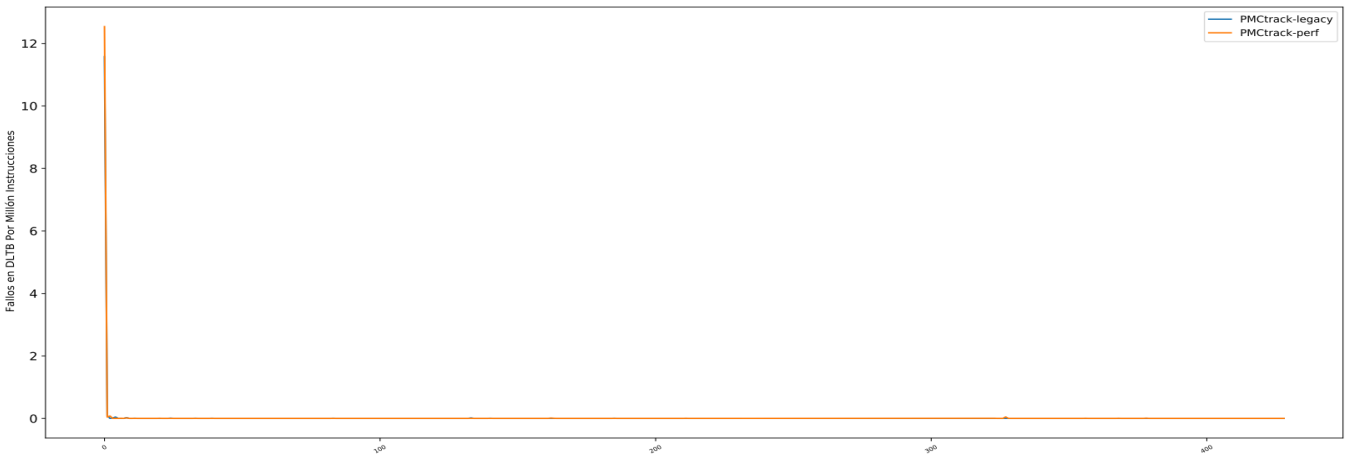
Instrucciones por Ciclo en el tiempo obtenido con Event-Based Sampling (EBS) para *lbm* de SPEC CPU 2017



Instrucciones de Salto Por Mil Instrucciones en el tiempo obtenido con Event-Based Sampling (EBS) para *lbm* de SPEC CPU 2017

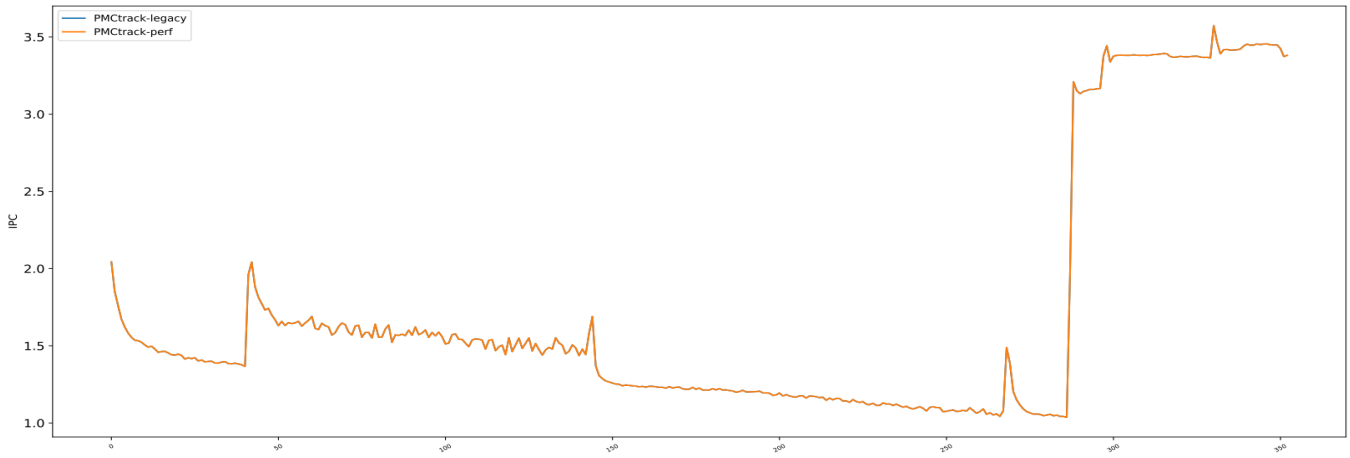


Fallos en Último Nivel de Caché Por Mil Instrucciones en el tiempo obtenido con Event-Based Sampling (EBS) para *lbm* de SPEC CPU 2017

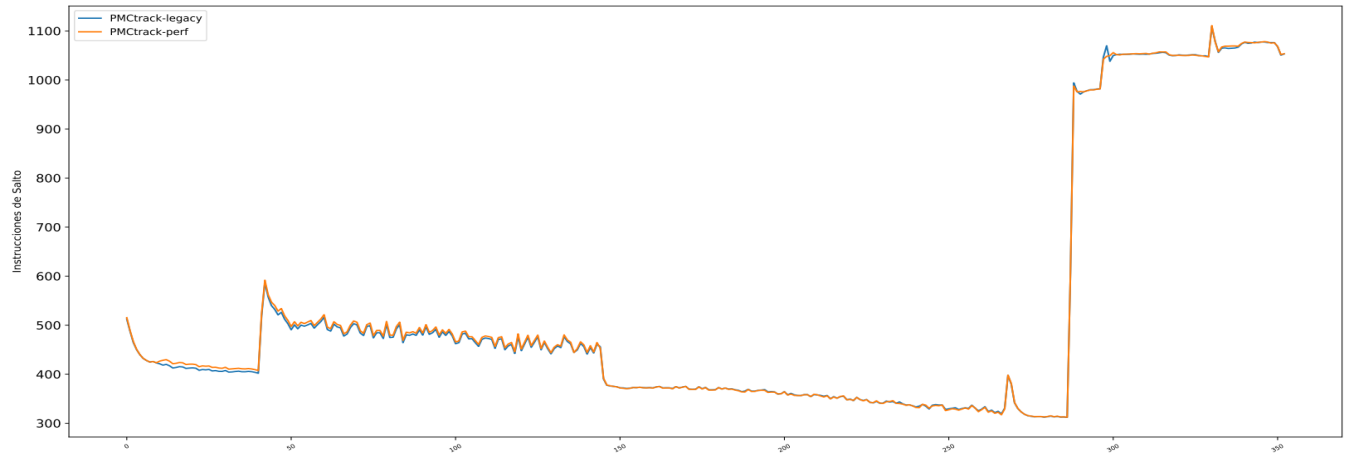


Fallos en DTLB Por Millón Instrucciones en el tiempo obtenido con Event-Based Sampling (EBS) para *lbm* de SPEC CPU 2017

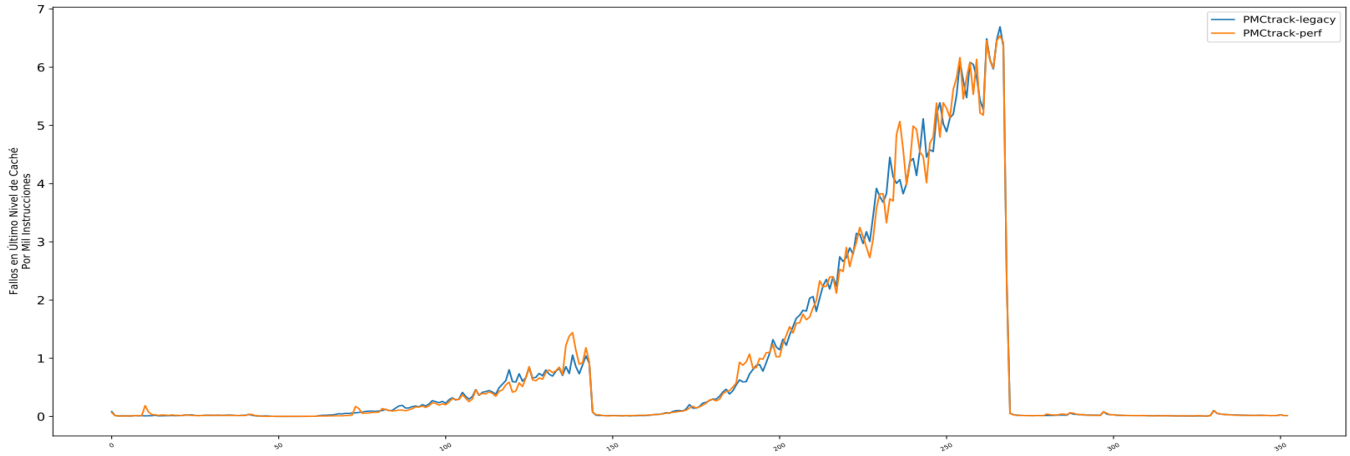
Xalancbmk17



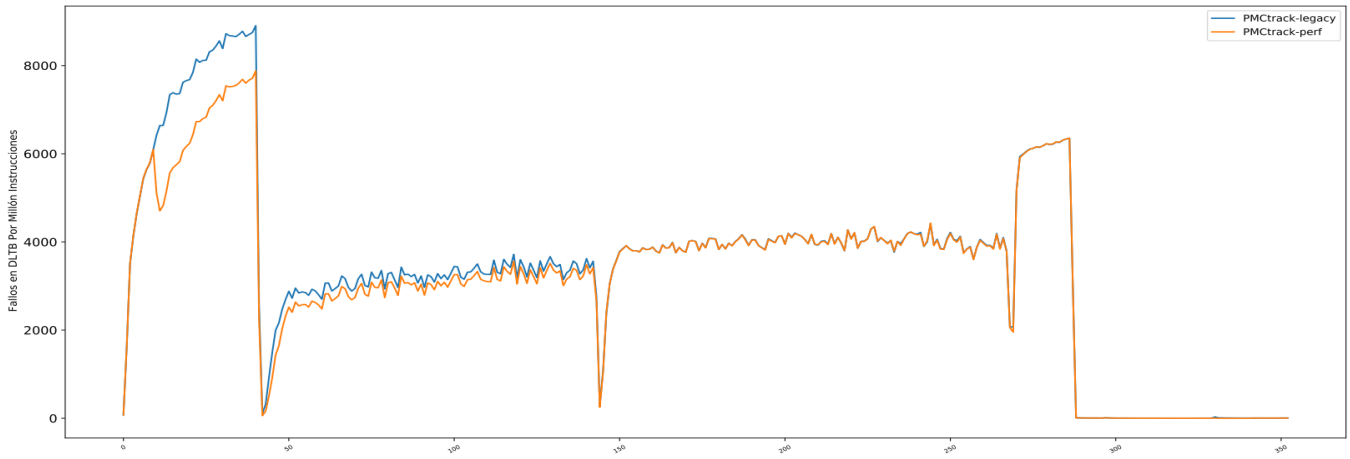
Instrucciones por Ciclo en el tiempo obtenido con Event-Based Sampling (EBS) para *xalancbmk* de SPEC CPU 2017



Instrucciones de Salto Por Mil Instrucciones en el tiempo obtenido con Event-Based Sampling (EBS) para *xalancbmk* de SPEC CPU 2017



Fallos en Último Nivel de Caché Por Mil Instrucciones en el tiempo obtenido con Event-Based Sampling (EBS) para *xalancbmk* de SPEC CPU 2017



Fallos en DTLB Por Millón Instrucciones en el tiempo obtenido con Event-Based Sampling (EBS) para *xalancbmk* de SPEC CPU 2017

En las gráficas de esta sección se observa una disparidad en las mediciones. Esto se debe a las diferencias en el funcionamiento del modo EBS en ambas herramientas. PMCtrack carga en el contador hardware usado para la interrupción el valor de umbral -U, y cuando éste desborda hay una interrupción que no se gestiona inmediatamente, ya que con los procesadores fuera de orden hasta que no se ejecuta la fase *commit* de la instrucción se han podido lanzar de manera especulativa instrucciones y no se gestiona. Entonces, la interrupción no se produce cuando se produce un desbordamiento, sino un tiempo después debido a la ejecución fuera de orden y superescalar. En PMCtrack-perf, sin embargo, cuando se efectúa modo EBS, perf modifica el valor -U de manera estática según el evento elegido mediante una **tabla de ajustes** que aplica un *offset* según el evento elegido. Esta es la razón de que las muestras se saquen antes y existe este desfase, que se ve muy claro en la gráfica que indica los fallos en último nivel de caché en modo EBS para *astar06*, o los fallos en DTLB por millón de instrucciones en modo EBS para *xalancbmk17*.

2. Estudio de la sobrecarga en la lectura de contadores

En esta última sección discutiremos los datos de tiempo requerido para la lectura de los contadores hardware en el kernel registrados para PMCtrack-legacy y para PMCtrack-perf. Para ello, se han efectuado mediciones a través de ambos métodos con 2, 4, 6, 8, y 10 eventos, y para cada una de ellas se han realizado alrededor de al menos 100 mediciones con el tiempo de ejecución asociado.

La siguiente tabla muestra los tiempos en nanosegundos para cada backend.

	PMCtrack-perf			PMCtrack-legacy		
Número de eventos	Media	Máx.	Mín.	Media	Máx.	Mín.
2	3191.51	4325	2696	3371.3828	4807	2108
4	3331.91	4501	2519	4348.1328	5719	2863
6	3688.14	4448	3252	4791.6798	6102	3569
8	3666.58	5093	2935	4859.7812	7039	4300
10	4477.64	5716	3769	6696.3203	8025	5109

A continuación, se muestra la misma tabla normalizada a la media obtenida con PMCtrack-perf, para ilustrar de una manera más visual la diferencia de tiempos en la lectura de contadores hardware.

	PMCtrack-perf			PMCtrack-legacy		
Número de eventos	Media	Máx.	Mín.	Media	Máx.	Mín.
2	1	1,35	0,84	1,05	1,50	0,66
4	1	1,35	0,75	1,30	1,76	0,86
6	1	1,20	0,88	1,23	1,65	0,96
8	1	1,39	0,80	1,32	1,92	1,17
10	1	1,27	0,84	1,49	1,79	1,14

De esta tabla podemos concluir que la lectura de contadores hardware en el *backend* de PMCtrack-perf es más rápida que en PMCtrack-legacy, creciendo la diferencia con el número de eventos a medir. Además, se puede observar que PMCtrack-legacy es mucho más sensible al número de eventos que el backend de perf en PMCtrack.

Se observa que PMCtrac-legacy es más sensible que PMCtrack-perf en cuanto al número de eventos a monitorizar. Sin embargo, el aumento de tiempo es despreciable: 8 microsegundos de overhead por cada 200 milisegundos es un aumento de tiempo despreciable.

Para efectuar unas conclusiones más profundas es necesario hacer una evaluación una vez que la implementación de PMCtrack-perf esté hecho con varios *backends* con varias arqutiectuars, ya que es difícil separar el *overhead* dado por x86 , ya que lo mismo en ARM no hay desfase.

La lectura de los contadores en PMCtrack-perf y PMCtrack-legacy funciona de manera muy distinta. En PMCtrack-legacy, cada vez que se lee el contador se para y se resetea, volviéndolo a configurar (tanto el contador como el evento de control donde residen los códigos del evento), lo que conlleva una carga operacional. Sin embargo, perf hace un *diff* del valor anterior almacenado con respecto al valor devuelto por el contador, sin parar ni reconfigurar el contador.

Capítulo 7

Conclusiones y Trabajo Futuro

PMCtrack es una herramienta para Linux que accede a los contadores hardware de monitorización del rendimiento del procesador, y permite recabar información relevante a partir de éstos para detectar cuellos de botella de rendimiento en las aplicaciones o realizar optimizaciones de diversa naturaleza. PMCtrack es capaz de ceder dicha información a diversos componentes de espacio de usuario --para que el usuario final pueda realizar una monitorización externa de las aplicaciones--, pero también proporciona la misma información al planificador del sistema operativo para poder realizar optimizaciones en tiempo de ejecución. Con respecto a otras herramientas de gestión de contadores hardware, PMCTrack tiene la ventaja de que la mayor parte de su funcionalidad en modo kernel se implementa dentro de un módulo cargable del núcleo, permitiendo poder llevar a cabo casi cualquier tipo de extensión sin la necesidad de reiniciar el sistema operativo. Sin embargo, requiere de un parche del kernel específico y soporte para el acceso a bajo nivel a los contadores hardware de cada arquitectura de procesador, que ha de programarse en parte usando código ensamblador.

Para hacer la herramienta más flexible y permitir, en un futuro no muy lejano, su uso en un kernel sin modificar, en este TFG se han llevado a cabo una serie de modificaciones críticas en PMCTrack. En primer lugar se ha desarrollado un nuevo *backend* de PMCTRack (módulo software específico de arquitectura para acceso a los contadores hardware) haciendo uso de la API de kernel del subsistema *perf events* del núcleo. Este subsistema fue creado para centralizar y estandarizar el acceso a contadores hardware en el núcleo, y ofrecer rutinas de bajo nivel para la implementación de herramientas de monitorización del rendimiento en Linux. Al comienzo del proyecto, no se tenía completa certeza de que la creación del backend de PMCTrack fuera posible con *perf events*: su API de bajo nivel ha sufrido muchas modificaciones en poco tiempo y la documentación es inexistente. Tras un estudio exhaustivo de la implementación de la API de *perf events* en el kernel Linux [16], ha sido posible crear una versión funcional del *backend* para procesadores actuales de Intel. Para el correcto funcionamiento de este *backend*, ha sido también necesario llevar a cabo modificaciones estructurales en el diseño de PMCTrack, y con ello realizar cambios relevantes en el código independiente de arquitectura de la herramienta. Otra de las novedades incorporadas en PMCTrack durante este TFG ha sido el soporte para nuevas versiones del kernel Linux (v5.4.y), que ha requerido la creación de un nuevo parche del kernel y las adaptaciones necesarias en el módulo del kernel.

Las mejoras realizadas en PMCTrack suponen un hito importante en la historia de esta herramienta, ya que por primera vez se permite contar con el soporte de *perf events* para acceso a contadores hardware en una gran cantidad de arquitecturas y modelos de procesador. Dicho de otra forma, si *perf events* ofrece soporte para un nuevo procesador en la versión del kernel que se ejecuta en el sistema, será ahora posible adaptar la herramienta (realizando cambios mínimos) para poder acceder a los contadores hardware en esa arquitectura a partir del nuevo backend Perf en PMCTrack, y todo ello sin tener que realizar la árdua tarea de escribir código ensamblador para gestionar los contadores hardware a bajo nivel. A pesar de esta ventaja, se puede concluir que el API del kernel de *perf events*, aunque es versátil, no resulta sencilla de utilizar y, provoca frecuentes interbloqueos en el sistema al invocar sus funciones en ciertas circunstancias (p.ej., contexto de interrupción con interrupciones no bloqueantes). La dependencia de esta API (cuyo modo de uso correcto no está documentado) ha obligado a rediseñar ligeramente aspectos de la herramienta PMCTrack que no habían requerido modificación alguna desde sus orígenes, como el mecanismo para la lectura periódica de los contadores hardware empleando un temporizador del kernel.

Para la validación del funcionamiento del *backend* creado sobre una nueva versión del kernel Linux (no soportada al comienzo del proyecto) se ha procedido a recabar un conjunto de métricas de rendimiento para los benchmarks de las suites SPEC CPU2006 y SPEC CPU2017. Los valores obtenidos se han comparado con los reportados por la herramienta de línea de comandos *perf*, y por la herramienta PMCTrack original, que accede a los contadores directamente usando código ensamblador. Los datos obtenidos revelan que el *backend* de *perf* para PMCTrack obtiene información precisa, e incluso es capaz de recabar la información de los contadores hardware con una menor sobrecarga que la que tiene la herramienta PMCTrack original.

Trabajo futuro

A pesar de las mejoras significativas realizadas en PMCTrack durante el TFG, aún se precisan cambios sustanciales para proporcionar funcionalidad relevante solicitada por la comunidad de usuarios, como lo que se menciona a continuación:

- **Soporte completo para todos los modos de monitorización de PMCTrack empleando el backend de *perf events*.** El código desarrollado es una prueba de concepto de que el PMCTrack se podía rediseñar por completo empleando el API de *perf events*, y, por tanto, eliminando la necesidad de acceder a los contadores hardware directamente. Para ello se ha incluido el soporte para los tres modos de monitorización soportados desde espacio de usuario: TBS, EBS y *automonitorización* (instrumentación de aplicaciones con *libpmctrack*). Sin embargo, los modos de monitorización a nivel de kernel (para optimizaciones en

el SO) no están aún disponibles mediante *perf events*. No obstante, se estima que el esfuerzo de desarrollo para soportar estos modos es muy reducido, y los grandes interrogantes que existían sobre si su implementación era posible con *perf events*, se han resuelto con la implementación de los modos de monitorización de modo usuario. Actualmente, se puede afirmar que el soporte de monitorización dentro del kernel es completamente viable empleando *perf events.*

- **Visibilidad de todos los eventos de *perf events* como contadores virtuales.** Otra gran ventaja asociada a la integración PMCTrack-perf es que ahora es posible dejar activa toda la funcionalidad de *perf events* en el kernel (antes era necesario desactivar parte de ella por incompatibilidades con PMCTrack). Esto abre la posibilidad de acceder a otro tipo de eventos de *perf events*, tanto software como hardware, y exponerlos como contadores virtuales de PMCTrack, para poder obtener de forma coordinada información sobre eventos hardware y otros aspectos --como fallos de página, cambios de contexto o medidas de consumo de energía-- que *perf events* también expone como eventos en su API.
- **Eliminación de la dependencia de un parche del kernel para funcionar.** La necesidad de parchear el kernel Linux es quizás una de las limitaciones más importantes de PMCTrack. Esto dificulta su adopción en entornos de producción donde el uso de kernels *custom* es menos práctico y más restrictivo. Una de las vías posibles para eliminar el parche, o al menos reducirlo de forma significativa, es explotar tecnologías de *tracing* del kernel como Linux TracePoints o *kprobes* para capturar los eventos de planificación relevantes para PMCTrack (creación/destrucción de hilos, cambios de contexto, invocaciones de exec(), etc,) y poder instalar un manejador (*probe*) para cada evento en el módulo del kernel. Esto permitiría eliminar por completo las notificaciones introducidas manualmente en el parche del núcleo. La eliminación de la dependencia del resto de cambios del parche (p.ej., incorporación de nuevos campos en el task_struct de Linux) requerirían la adopción de estos cambios en la versión *mainline* del kernel, y por lo tanto contar con el apoyo correspondiente de la Comunidad de Linux.

Capítulo 8

Bibliografía

[1] J. C. Sáez, J. I. Gomez, and M. Prieto, “Improving priority enforcement via non-work-conserving scheduling,” in *ICPP ’08: Proceedings of the 2008 37th international conference on parallel processing*, 2008, pp. 99–106.

[2] J. C. Saez *et al.*, “Delivering fairness and priority enforcement on asymmetric multicore systems via OS scheduling,” in *Proceedings of the acm sigmetrics/international conference on measurement and modeling of computer systems*, 2013, pp. 343–344.

[3] A. Garcia-Garcia, J. C. Saez, and M. Prieto-Matias, “Contention-aware fair scheduling for asymmetric single-ISA multicore systems,” *IEEE Transactions on Computers*, vol. 67, no. 12, pp. 1703–1719, Dec. 2018.

[4] J. C. Saez et.al, “On the interplay between throughput, fairness and energy efficiency on asymmetric multicore processors,” *The Computer Journal*, vol. 61, no. 1, pp. 74–94, 2018.

[5] J. C. Saez *et al.*, “An OS-oriented performance monitoring tool for multicore systems,” in *Proc. Of euro-par 2015: Parallel processing workshops*, 2015, pp. 697–709.

[6] A. Garcia-Garcia *et al.*, “LFOC: A lightweight fairness-oriented cache clustering policy for commodity multicores,” in *Proc. Of ICPP’19*, 2019.

[7] PMCtrack oficial site. https://pmctrack.dacya.ucm.es

[8] Perf wiki: Main Page. https://perf.wiki.kernel.org/index.php/Main_Page

[9] J. C. Sáez, A. Pousa, R. Rodríguez-Rodríguez, F. Castro, M. Prieto-Matías. “PMCtrack: Delivering performance monitoring counter support to the OS scheduler”, in *The Computer Journal*, vol.60, no. 1, 2017.

[10] Bootlin: oProfile documentation. https://elixir.bootlin.com/linux/v5.6.4/source/drivers/oprofile

[11] S. Jarp, R. Jurga, A. Nowak. “Perfmon2: A leap forward in Performance Monitoring.”, in *Journal of Physics Conference Series*, 2008.

[12] PMCtrack github site. https://gitlabce.dacya.ucm.es/jcsaez/pmctrack

[13] J. Casas, A. Serrano. “Interfaz de uso de contadores hardware multiarquitectura”, 2015.

[14] Perf Examples. http://www.brendangregg.com/perf.html

[15] perf-list(1): Linux manual page. https://www.man7.org/linux/man-pages/man1/perf-list.1.html

[16] Linux Kernel Documentation: core.c File Reference version 3.7.1. https://docs.huihoo.com/doxygen/linux/kernel/3.7/kernel_2events_2core_8c.html

[17] perf_event Programming Guide. http://web.eece.maine.edu/~vweaver/projects/perf_events/programming.html

[18] Heechul Yun: Memory bandwith controller for multi-core systems. https://github.com/heechul/memguard/blob/release/memguard.c

A. Introduction

PMCtrack is an open-source OS-oriented **performance monitoring tool** for GNU/Linux. It is specifically designed to assist kernel developers in the implementation of process scheduling algorithms that exploit information provided by **performance monitoring counters** (PMCs) to perform **run-time** optimizations [7].

Despite being an OS-oriented tool, it also allows to gather PMC values from **user space** in a number of ways, necessary to assist developers during scheduler design process. In addition to providing the capture of metrics such as the number of instructions executed per processor cycle or the last-level cache misses rate, PMCtrack also lends **hardware monitoring metrics** available on moder processors such as **space used at the last cache level** or **power consumption** to the scheduler [1,2,3,4,5,6] and to different user space components of the tool.

To access hardware counters in a low level, it is necessary to use assembly code, so any tool that accesses them is architecture-dependant. PMCtrack offers a substantial advantage over other tools, which is the fact that much of its functionality is implemented within a kernel loadable module. This allows almost any extension that it is necessary to implement in the tool be implemented without the need to restart the operating system. However, PMCtrack has structural limitations: (1) it requires low-level support for direct access to hardware counters on the various processor architectures and platforms where it is intended to be used. For each architecture it must be implemented a specific *backend*, that is, a loadable architecture specific software module to access the meters at a low level.

Another tool available in Linux to simplify the management and access to hardware counters is Perf. It is a performance monitoring and tracing tool built into the Linux kernel (from v2.6.31 on 2009), with user space control. Much of the functionality of *perf* can be exploited from the command line, and the corresponding tool allow as, among other things, to perform a statistical profiling of the entire system [8].

Perf is currently one of the most used performance monitoring tools. However, it has two very imposing limitations. The theorical limitation is the documentation shortage available for the tool; for example, most events and aliases for Perf are not documented. The practical limitation is harder: Perf is a tool fully implemented within the kernel, which means that any extension that is needed to incorporate requires the compilation of the Linux kernel, and the corresponding machine reboot for activation and debugging. Perf contains a low-level API that was created to be used by various performance monitoring tools to collect information from counters. Furthermore, Perf was not

specifically designed to perform monitoring from the kernel itself, and the API that currently supports this has undergone numerous modifications in the last decade.

The project's final goal is to provide PMCtrack with some of the most requested by the user extensions, such as providing a *backend* based on the Linux kernel *perf events* system, and carrying the different components of PMCtrack to new longterm kernel versions (such as 5.4.y).

1. Motivation and project objectives

The problem when using tools like this is that, as mentioned above, they are **architecture-dependant,** and there are currently endless processor architectures equipped with hardware counters, each one with its own events and access mechanisms. The short-term goal for this project is the complete **integration** of the *perf events* API functionality regarding access to hardware counters from PMCtrack, encapsulating this functionality as a new PMCtrack *backend*

As mentioned above, PMCtrack cannot be used on unmodified or vanilla Linux kernels, but it requires a specific kernel patch to work properly. This limitation menas that, in practice, PMCtrack advanced users must have extensive knowledge of the kernel Linux operation. If the appropiate patch is available for the desired Linux kernel version, a user with Linux administration skills could compile it and make PMCtrack work. If the patch is not available, only a user with average Linux kernel development knowledge will be able to run PMCtrack, as they will have to port the kernel patch to the desired version. Although the PMCtrack kernel patch is simple and affects few files and relatively stable kernel code, heavy use of the Linux API from the PMCtrack kernel module can break compatibility between different kernel versions, such as we will discuss in the following sections. The existence of this type of incompatibilities sometimes requires the modification of the PMCtrack kernel module for its correct operation in arbitrary versions of the kernel (not officialy supported for PMCtrack).

This project intends to eliminate these limitations in the medium or long term by creating a PMCtrack *backend* based on perf API. In this way, the performance monitoring subsystem of Linux kernel is used, which for some years has been de facto standard for managing low-level access to hardware counters, and which has support for numerous mainline processor architectures.

To achieve the complete elimination of the aforementioned limitations, there are 3 significant barriers: the first one consists in the integration of the *perf* API as one more *backend* within the PMCtrack tool; the second is the support for new kernels that allow the use of event tracing technologies in the Linux scheduler such as TracePoints and

kprobes and, lastly, to eliminate or reduce to its minimum expression the patch needed for the execution of PMCtrack in a *vanilla* kernel. PMCtrack needs to receive notifications from the scheduler (via callbacks) to know when critical events happen –such as process creation/termination or context switches –, and this is the main problem to completely remove dependency on the kernel patch.

The main goal of this project is to remove the two first mentioned barriers. Removing PMCtrack's dependency from its kernel patch is a long-term goal too ambicious to address in this Final Degree Project. After all, at the beginning of this project it was still necessary to study whether with the *perf* API it was really feasible to create a *backend* for PMCtrack.

2. Work plan

To achieve the Final Degree Project objectives mentioned in the previous section, the development of the project has been divided into different stages:

- PMCtrack documentation reading and tool source code analysis.
- Installing the modified PMCtrack kernel patch for version 4.9.33_x86.
- Reading perf API documentation.
- Study of practical examples of the perf API in the kernel[18].
- Study of the Linux kernel code associated with the perf API calls.
- Implementing the necessary changes in PMCtrack code for its integration with *perf*.
- Exectuion of validation experiments of the functionality incorpored in PMCtrack and analysis of the overhead produced.
- Construction of graphs and analysisof results.
- Writing of the report.

3. Memory structure

The rest of the document is structured on the following chapters:

- **Chapter 2** contains a description of the historical context in which PMCtrack emerged as well as how this performance monitoring tool works.
- **Chapter 3** presents a description of the performance monitoring tool *perf*, havins a special impact on the operation and use of the API
- **Chapter 4** exposes everything related to the creation of the new PMCtrack *backend*, focusing on the need for integration, as well as the implementation details and limitations identified.

- **Chapter 5** defines the changes necessary to include the needed support in PMCtrack to run on version 5.4 of the Linux kernel, detailing the changes necessary to both the kernel patch and the kernel loadable module.
- **Chapter 6** discusses the experimental results obtained, and discusses both their validation (precision of the reported performance metrics) and the overhead associated with reading meters.
- **Chapter 7** presents the final conclusions of the Final Degree Project and discusses future work.

B. Conclusions and future work

PMCtrack is a tool for Linux that accesses performance monitoring hardware counters (PMCs), and allows gathering relevant information from them to detect performance bottlenecks in application or perform optimizations of various kinds. PMCtrack is capable of transferring this information to several user space components –so that the end user can perform external monitoring of the applications –, but it also provides the same information to the operating system scheduler to be able to perform optimizations at run time. In relation to other hardware meter management tools, PMCtrack has the advantage that most of its functionality in kernel mode is implemented within a loadable kernel module, allowing almost any type of extension to be carried out without the need of restart the operating system. However, it requires a specific kernel patch and support for low level access to the hardware counters of each processor architecture, which has to be programmed in part using assembly code.

In order to make the tool more flexible and to allow its use in an unmodified kernel in the not too far future, a series of critical modifications have been carried out in PMCtrack in this Final Degree Project (FDP). First, a new PMCtrack *backend* (architecture-specific module for access to hardware counters) has been developed using the kernel API of the core's *perf events* subsystem. This subsystem was created to centralize and standardize access to hardware counters in the kernel, and to offer low-level routines for the implementation of performance monitoring tools in Linux. At the beggining of the project, it was not completely certain that the creation of the PMCtrack *backend* was possible with *perf events*: its low-level API has undergone many modifications in a short time and the documentation is non-existent. After an exhaustive study of *perf events* API implementation in the Linux kernel[16], it has been possible to create a working version of the *backend* for current Intel processors. For the correct functioning of this *backend*, it has also been necessary to carry out structural modifications in the design of PMCtrack, and thus make relevant changes in the independent code of the tool's architecture. Another of the new features incorporated in PMCtrack during this FDP has been the support for new versions of Linux kernel (5.4.y, which has required the creation of a new kernel patch and the necessary adaptations in the kernel module.

The improvements made to PMCtrack represent an important milestone in the history of this tool, since for the first time it is possible to have the support of *perf events* for access to hardware counters in a large number of architectures and processor models. In other words, if *perf events* offers support for a new processor in the kernel version running on the system, it will not be possible to adapt the tool (making minimal changes) to be able to access the hardware counters in that architecture from the new *perf backend* in PMCtrack, and all without having to do the ardous task of writing

assembly code to manage hardware counters at a low level. Despite this advantage, it can be concluded that the *perf events* kernel API, although versatile, is not easy to use and causes frequent deadlocks in the system when invoking its functions in certaing circumstances (e.g., context of interrupt with non-blocking interrupts). The dependence on this API (whose correct use is not documented) has forced to redesign slightly aspects of PMCtrack that had not required any modification since its origins, such as the mechanism for the periodic reading of the hardware counters using a kernel timer.

For the validation of the *backend* created on a new version of the Linux kernel (not supported at the beginning of the project), a set of performance metrics has been collected for the benchmarks of the SPEC CPU 2006 and SPEC CPU 2017. The values obtained have been compared with those reported by the *perf* command line tool and by the original PMCtrack tool, which accesses the counters directly using assembly code. The data obtained reveals that the *perf backend* for PMCtrack obtains accurate information, and is even capable of collecting information from the hardware counters with less overhead than the original PMCtrack tool has.

Future work

Despite the significant improvements made to PMCtrack during this FDP, substantial changes are still required to provide relevant functionality requested by the user community, such as the following:

- **Full support for any PMCtrack monitoring modes using the *perf events* backend.** The code developed is a proof of concept that PMCtrack could be completely redesigned using the *perf events* API, and therefore eliminating the need to access the hardware counters directly. To achieve this, support has been included for the three monitoring modes supported from user space: TBS, EBS and self-monitoring (application instrumentation with *libpmctrack*). However, kernel-level monitoring modes (for OS optimizations) are not yet available via *perf events*. Nevertheless, it is estimated that the development effort to support these modes is very low, and the big questions that existed about whether their implementation was possible with *perf events* have been solved with the implementation of user-mode monitoring modes. Currently, it can be said that monitoring support within the kernel is completely viable using *perf events*.
- **Visibility of any *perf events* as virtual counters.** Another great advantage associated with the PMCtrack-perf integration is that it is now possible to leave the entire *perf events* functionality active in the kernel (previously it was necessary to deactivate part of it due to incompatibilities with PMCtrack). This opens the possibility of accessing other types of *perf events*, both software and

hardware, and exposing them as virtual PMCtrack counters, in order to obtain information on hardware events in a coordinated way –such as page faults, context switches or power consumption measuers– which *perf events* also exposes as events in its API.

- **Elimination of dependency on a kernel patch to function.** The need to patch the Linux kernel is perhaps one of the most important limitations of PMCtrack. This makes difficult to adopt in production environments where the use of custom kernel is less practical and more restrictive. One of the possible ways to remove the patch, or at least reduce it significantly, is to exploit kernel tracing technologies, such as Linux Trace Points or *kprobes*, to capture the planning events relevant to PMCtrack (create / destroy threads, context switches, invocations of *exec()*, etc.) and to be able to install a handler (*probe*) for each event in the kernel module. This would make it possible to completely remove notifications entered manually in the kernel patch. Removing the dependency on the rest of the patch's changes (f.e., adding new fields in the Linux task_struct) would require the adoption of these changes in the *mainline* kernel version, and to therefore have the corresponding support from the Linux Community.