Registro de Fallos en Accesos a Caché (L2) del Pentium 4 mediante *Performance Counters*.

Agustín Isasa Cuartero
ITIS

Consultor: Enric Morancho Llena

Octubre 2004 / Enero 2005.

A mis padres.

Sin lo que ellos me han mostrado, no habría sabido hacerlo.

Agradecimientos

La paciencia y comprensión de mi familia no han tenido límites en estos últimos años. Sé con seguridad que el esfuerzo volcado en esta carrera y en este trabajo es más de ellos que mío, y nunca podré agradecerlo suficientemente.

Mi tutor en este TFC, Enric Morancho Llena, supo darme la confianza que necesitaba para afrontar el tema del trabajo. En su desarrollo, no sólo ha resuelto dudas y aclarado misterios del lenguaje C, sino que, en una especie de objetivo no explícito del TFC, con sus indicaciones y consejos me ha ayudado a subir unos cuantos peldaños en mi nivel de programación de sistemas.

A todos ellos, gracias.

RESUMEN

Para representar de forma objetiva el comportamiento y **rendimiento de un determinado software** se hace uso, entre otras estrategias, de la métrica de **eventos**, que es la contabilización y estudio contextual de las ocurrencias de determinados hechos relevantes que acaecen al ejecutar el código en estudio. Los eventos que nos interesan deben ser analizados desde la perspectiva de la eficiencia en el uso del procesador y subsistemas vinculados (buses y memorias, fundamentalmente). Para ello, los microprocesadores actuales incluyen un repertorio de eventos que pueden ser monitorizados, así como los mecanismos hardware para hacerlo. En general, estos mecanismos se denominan ***performance counters***. Sin embargo, los sistemas operativos de mayor difusión no ofrecen la gestión de estos contadores, por lo que cualquier aproximación a los mismos pasa por la inclusión de nueva lógica en el código del SO. Puesto que una familia de eventos importante en el estudio del rendimiento de una aplicación, y de su potencial mejora, es la que trata del análisis de los **fallos en los accesos a las memorias caché** y, en particular, a las de último nivel (de mayor impacto en el rendimiento en comparación con las de niveles anteriores: los fallos en su acceso son mucho más costosos en términos de instrucciones), abordaremos la implementación del acceso a los *performance counters* del **Pentium 4** (microprocesador generalista y muy extendido) para el estudio de los fallos de lectura a su caché de segundo nivel. El sistema operativo soporte será **Linux**, que deberemos modificar y complementar en lo necesario para lograr este acceso.

INDICE

CAPITULO 1

INTRODUCCION

1.1 Presentación del TFC

Tras cuatro años de estudio de la Ingeniería Técnica en Informática de Sistemas ofrecida por la UOC nada más iniciar, exactamente hace ese periodo, su proyecto Iberoamericano, ha llegado el momento de intentar resumir lo trabajado hasta ahora en la última pieza de la titulación: el Trabajo de Fin de Carrera.

Para ello, y de forma previa, es conveniente dirigir la vista hacia atrás y repasar las áreas y herramientas que más conocimiento, interés y satisfacciones han aportado.

Y es difícil encontrar una herramienta que otorgue estos dones con tanta liberalidad como Linux: como usuario, y tras coronar la curva de aprendizaje inicial (de menor pendiente en cada nueva distribución que se nos ofrece), el control que permite a través de la parametrización de *todos* sus aspectos es imposible de lograr con otros SO propietarios. Su espectacular extensión en los últimos años, por otra parte, hace que el conjunto de hardware soportado supere en mucho al del resto de sistemas operativos de código abierto, y ha alcanzado ya la magnitud del sistema propietario más extendido.

Como desarrollador, las características del software libre, su calidad (resultado directo, precisamente, de las características de su desarrollo abierto) y la ingente cantidad de documentación disponible hacen de Linux, cuando se trata de modificar o aplicar funcionalidades nuevas a un SO, el sistema de referencia no sólo para el tipo de trabajos de naturaleza didáctica como el que nos ocupa, sino para áreas tales como las comunicaciones, supercomputación, I+D, bases de datos o servidores de todo tipo. Además, su uso se está extendiendo, cada vez más, al soporte de administraciones públicas y otras tareas de gestión administrativa hasta ahora prácticamente circunscritas a la familia Windows, sistemas operativos tradicionales en estos campos .

Bajo este sistema he realizado la mayor parte de las prácticas de las asignaturas que contaban con este tipo de trabajos, y todos los problemas que he ido encontrando han podido resolverse rápidamente gracias a la gran difusión que ha experimentado el conocimiento sobre los aspectos más técnicos y complejos de Linux.

El hecho de tener que haber trabajado un buen número de horas con software de código abierto, así como con documentación clara y valiosa que debe estas propiedades al hecho de estar basada en este tipo de software, me ha hecho valorar en especial medida la corriente de código abierto que ha conseguido elaborar software tan útil, complejo y barato como el propio sistema Linux o el conjunto de sus utilidades.

No menos importante es la extensión (con la eficaz ayuda de Internet) de esta filosofía. Sus poderes están resultando de tal magnitud que hasta el actual paradigma de software propietario se está poniendo en cuestión, y las compañías que lo desarrollan y lo comercializan asisten, un poco aturdidas y confusas, al acoso a dicho paradigma. Aunque parece pronto para apostar por un desenlace, la batalla se ha puesto de manifiesto y se están librando las primeras escaramuzas.

Por lo tanto, en mi caso parecía destinado, como así ha sido, a elegir de entre las posibles áreas del TFC de la UOC la relativa a la plataforma GNU/LINUX. De manera que tras algunos planteamientos alternativos con el tutor del Trabajo, finalmente se decide enfocar el TFC hacia alguna habilitación en Linux de los contadores de rendimiento (*performance counters*) del Pentium 4.

Es ésta un área no demasiado extendida ni conocida. Incluso parece que las unidades del procesador que soportan estos contadores aún no han alcanzado un alto grado de madurez. De hecho, ya avanzado este trabajo, vimos que Intel declara errores (hasta factores de 2 y de 1/2 de los resultados ofrecidos) en los muestreos de la familia de eventos que estudiaremos.

El acceso a este hardware no es trivial, y apenas existe literatura de divulgación básica (mucho menos en castellano). Debo reconocer que para mí era absolutamente nuevo, tanto desde la perpectiva del hardware (apenas sabía de la existencia de estos contadores) como del software (nunca me había enfrentado a la programación del kernel de Linux). El desafío, por lo tanto, estaba planteado.

El objeto de nuestro estudio, en mi opinión, es ideal como Trabajo de Fin de Carrera sobre GNU/LINUX para una ingeniería técnica: por un lado, obliga a una búsqueda y estudio en profundidad de la documentación técnica disponible sobre el hardware, actividad a la que todo ingeniero de sistemas deberá enfrentarse con relativa frecuencia para unos u otros fines; por otra parte, el diseño y programación del correspondiente *driver* exigirá una importante comprensión de los mecanismos que el sistema operativo utiliza para comunicarse con este tipo de dispositivos: en UNIX/LINUX todo elemento que soporte operaciones I/O se trata como un archivo, por lo que será pertinente el estudio del Sistema de Ficheros Virtual (VFS), además de, como iremos viendo, comprender cómo actúan los procesos en Linux, la programación de módulos, conocer con cierto detalle el funcionamiento de las principales llamadas al sistema que nos sean relevantes y revisar materias como el ensamblador del Pentium 4, sincronización u otros temas avanzados de programación.

1.2 Objetivos

Con los primeros procesadores Pentium, Intel empieza a implementar en sus productos unos registros especiales que ya existían en procesadores de otros diseñadores, y cuyo fin era almacenar el resultado de procesos cuantificadores de distintos eventos de la arquitectura que podían monitorizarse si se deseaba. En adelante, nos referiremos a estos registros como contadores de rendimiento o, casi preferentemente, con su nombre original en inglés ***performance counters***, que no genera tanta ambigüedad entre términos aplicables a la traducción como *comportamiento* o *rendimiento*.

La información que puede obtenerse de estos contadores corresponde a un buen número de **eventos** que se organizan en diferentes grupos de acuerdo a su finalidad y entre los que pueden mencionarse (por nombrar un pequeño conjunto) los fallos de acceso a memoria caché, los accesos de diferente tipo al bus del sistema, el número de diferentes tipos de fallos de predicción en código condicional o el número de ejecuciones de una determinada familia de instrucciones.

Mediante el análisis directo de los datos obtenidos de cada evento, o mediante una métrica calculada por relaciones aritméticas entre los resultados obtenidos de varios eventos (por ejemplo: el porcentaje de fallos en lectura de caché vendría dado por la suma de los diferentes tipos de fallo obtenidos, dividida por la suma de los diferentes tipos de fallo más los diferentes tipos de acierto), es posible la detección de zonas de código que originen cuellos de botella en su ejecución (***hot spots***), facilitando su posterior ajuste y optimización.

En un principio, la información sobre buena parte de estos recursos permaneció propietaria de Intel, ya que se utilizaban, principalmente, para comprobaciones internas de calidad. Pero no tardaron en aparecer informes no oficiales e independientes que fueron poniendo al descubierto el funcionamiento (al menos parcial) de los nuevos registros implementados. En la actualidad, la información del fabricante sobre esta materia es exhaustiva y está distribuida en diferentes manuales orientados a diferentes objetivos y usuarios.

Conforme la gama Pentium evolucionaba, también lo hacía el conjunto de registros y el de eventos que podían monitorizarse, así como los mecanismos de acceso y las estructuras de datos implicados en la cuantificación de los eventos.

Esta evolución, que ha llevado pareja el incremento de la complejidad en el manejo de los *performance counters*, ha sido tan dependiente de cada modelo de procesador que no existe una metodología común de acceso a estos registros especiales en toda la gama Pentium.

Un proyecto que abarcara la implementación de los accesos a los contadores de rendimiento de cualquier Pentium requeriría un tamaño que sobrepasa el del trabajo que debe hacerse. Nos centraremos, pues, en los *performance counters* del **Pentium 4** (y que también corresponden a los del Pentium Xeon), el procesador más común y reciente de la serie.

Asímismo, obviaremos también las implicaciones de la todavía más reciente tecnología *Hyper-Threading*, que duplica el número de procesadores lógicos en cada procesador físico. Por un lado, sólo supone una duplicidad de registros especiales y, en general, de la infraestructura de acceso a los *performance counters* que ya existen

en un procesador sin *Hyper-Threading*. De otro, el procesador en el que vamos a hacer las pruebas no posee esta tecnología, lo que zanja definitivamente la cuestión.

Un vistazo rápido a la documentación de Intel nos permite ver que los Pentium 4 contemplan el control de varias decenas de grupos de eventos. Cada una de estos grupos puede tener a su vez varios tipos de eventos, lo que hace que el número de éstos sobrepase de largo el centenar.

Tampo sería factible que nuestro proyecto abarcara el control de la totalidad de los eventos. Ni siquiera del conjunto de los más significativos y conocidos.

En este sentido, se ha consensuado que un **objetivo** realista sería el acceso, lectura y, en su caso, cálculo de la correspondiente métrica de eventos pertenecientes a uno de los grupos de relevancia. En particular, el grupo correspondiente a los accesos a la memoria caché, que nos permitiría analizar los fallos en los accesos a dicha memoria, parece por motivos obvios un grupo muy interesante de entre los existentes. Y precisamos todavía más nuestro objetivo considerando que los fallos en la caché de último nivel (segundo, o L2, en el Pentium 4) son mucho peores en términos de rendimiento que los fallos en las cachés de niveles más cercanos al procesador, ya que originan un tráfico de instrucciones de una magnitud mayor (tanto en número como en tiempos de ejecución) por exigir accesos a la memoria física, muy lenta en comparación con los circuitos de las cachés y que, además, puede encontrarse intercambiada, en sistemas paginados, en mecanismos de archivo todavía mucho más lentos. Se tomó, por lo tanto, el grupo de eventos que ponen de manifiesto los fallos en los accesos a lectura de la L2 del Pentium 4 como objetivo principal de nuestro trabajo.

Pero no sólo hemos debido trabajar la gestión del hardware correspondiente. Hemos implementado el **acceso a los *performance counters* desde el núcleo de Linux**, para lo que tuvimos que dilucidar en su momento si crear una estructura adicional que soportara nuevas llamadas de sistema, incluir un nuevo driver en el kernel o habilitar su funcionamiento mediante un módulo, con aparentes ventajas de comodidad a la hora de compilar y depurar. Como se verá, se decidió esta última estrategia debido a su mayor flexibilidad.

Un importante problema a resolver ha sido el de la obtención de información para un *determinado* proceso y un periodo dado, y no para la ocurrencia de un determinado evento en un determinado periodo, que supondría contabilizar ocurrencias debidas a diferentes procesos. Esta última información no tiene, generalmente, la misma relevancia que el análisis por proceso: puesto que nuestro finalidad última será construir una herramienta para el análisis del comportamiento de un determinado código, debemos asegurarnos que la monitorización efectuada recoja información generada única y exclusivamente en la ejecución del código en estudio.

La documentación de Intel ya apunta a las posibles modificaciones a efectuar en los sistemas operativos que quieran discriminar los resultados por procesos, modificaciones que hubo que valorar.

Como **resumen** de todo lo anterior, nuestro proyecto intentará conseguir la contabilización de los fallos en los accesos a caché de segundo nivel (L2) de un determinado proceso, para lo que trataremos de habilitar los contadores de

rendimiento del Pentium 4, configurarlos para obtener la información buscada de acuerdo a los eventos a monitorizar y, por último, obtener esta información y dejarla a disposición de la aplicación de usuario que la requiera.

Nos marcamos como objetivo adicional el estructurar el trabajo (y el código resultante) de forma tal que permita una implementación rápida y sencilla de la consulta a otras familias de eventos, lo que significa que nuestro trabajo debe proveer un marco para el acceso genérico a los contadores, que sea independiente del evento o eventos a analizar, y que permita la rápida definición e implementación de nuevos eventos.

1.3 Enfoque y método

El proyecto ha seguido un **diseño descendente**, según el cual se han ido dividiendo y refinando los problemas en estudio en otros de menor tamaño que permitiera atacarlos de forma eficiente.

Las primeras etapas son fundamentales para una correcta planificación y su posterior cumplimiento. En nuestro caso, estas **primeras etapas** las resumimos de la manera siguiente:

1. Elaboración de los objetivos y su formulación con la menor ambigüedad posible.

2. Comprensión del problema: conceptos básicos sobre monitorización del rendimiento.

3. Comprensión del problema: implicaciones hardware.

4. Comprensión del problema: implicaciones software.

Acabamos de ver en el anterior apartado los antecedentes y la formulación de los objetivos de nuestro TFC.

En las siguientes etapas, y para comprender en profundidad el problema a resolver, se ha tenido que trabajar, a veces de forma secuencial, a veces de forma simultánea, en los diferentes frentes que han tenido que ser atendidos. Básicamente, estos frentes corresponden a las connotaciones hardware y software del problema a resolver, aunque previamente ha resultado conveniente contar con información genérica sobre la **monitorización del rendimiento** de sofware.

En cuanto al **hardware**, se han tenido que estudiar y comprender los mecanismos de configuración y de lectura de los *performance counters*. Para ello, la mejor fuente ha resultado ser el conjunto de manuales que componen la documentación técnica de Intel sobre el procesador Pentium 4, aunque también resultaron de utilidad otros artículos o notas que se incluyen en la bibliografía. Más adelante, en el capítulo correspondiente de esta memoria, se hace una introducción a la medida del rendimiento y a los contadores hardware del mismo en general, para pasar seguidamente al análisis de los del Pentium 4 en particular.

Por lo que se refiere a los aspectos **software**, se han tenido que identificar los diferentes campos de acción (o subproblemas) sobre los que aplicar las correspondientes soluciones. Los más relevantes han sido:

- Programación del kernel de Linux.
- Programación de módulos.
- Programación de dispositivos de caracteres.
- Instrucciones del ensamblador del Pentium 4 necesarias para la configuración de los contadores y su implementación con gcc.
- Estructuras de datos: abstracción de la información de configuración del hardware.
- Estructura de los procesos y su planificación y ciclo de vida en Linux.

Tras estas primeras fases de análisis se pasaba a las **etapas de diseño y desarrollo** (incremental) del correspondiente sofware (ver el siguiente apartado para más detalles sobre la planificación del trabajo). Estas nuevas etapas podemos clasificarlas:

5. Diseño y desarrollo del software para un acceso básico a los contadores del Pentium 4, y que supone que un contador contabiliza *todas* las ocurrencias de un evento originado por *todos* los procesos activos. A su vez, esta etapa se ha subdividido en las siguientes:

    a) Configuración, puesta en marcha, lectura y detención de un contador determinado.

    b) Implementación de la configuración, puesta en marcha, lectura y detención del conjunto de los 18 contadores del Pentium 4 de forma concurrente, y activables por uno o varios procesos diferentes, con los mecanismos de control y seguridad adecuados.

    c) Elaboración de una API definitiva para su uso por aplicaciones de usuario.

6. Diseño y desarrollo de la evolución del software anterior para monitorizar un determinado código a especificar por la aplicación de usuario. Los contadores, en este caso, sólo están activos a la ejecución del código en cuestión. Esta etapa ha supuesto:

    a) Modificaciones apropiadas del sistema operativo.

    b) Modificación del software básico inicial para que pueda hacer uso del SO modificado y conseguir nuestro objetivo.

Pueden verse mayores detalles sobre el diseño y el software resultante en los capítulos de la memoria destinados a tal fin.

Consignamos, para terminar, una última **etapa de pruebas**, anterior a la entrega del producto final:

7. Pruebas del código, así como revisión, adición de complementos y limpieza del mismo.

1.4 Planificación

Al principio del proyecto contábamos con once o doce semanas para el desarrollo del *producto* del TFC (en nuestro caso, el software para el acceso a los registros contadores), a las que habría que añadir dos o tres adicionales para la elaboración de la memoria y la presentación virtual.

La **temporización** que se propuso fue la siguiente:

- **Semanas 1 a 3:** estudio de la documentación necesaria sobre los *performance counters*, programación del kernel de Linux y revisión de la documentación de gcc sobre implementación de ensamblador en rutinas C.

- **Semanas 4 a 5:** diseño básico del acceso a los contadores, incluyendo decisiones sobre forma definitiva de implementación (¿librería de sistema?).

- **Semanas 6 a 7:** implementación en código del acceso a los contadores para obtención de resultados genéricos (sin discriminar por proceso).

- **Semanas 8 a 9:** diseño de la discriminación por proceso para permitir monitorizar accesos a caché de un proceso determinado.

- **Semanas 10 a 12:** implementación en código del anterior diseño, acoplamiento con el código de acceso a los contadores, posible API, interfaz de usuario (modo texto) y ajustes finales.

El **seguimiento temporal** de estos hitos mediante la segunda y tercera **PECs** a entregar durante el transcurso del trabajo (la primera PEC correspondió al Plan de Trabajo que acabamos de resumir), significó la asunción del compromiso de que cada una de ellas comprendiera:

- **PEC 2** (entrega en aproximadamente 1/3 del plazo de realización del TFC): informe sobre la estructura y funcionamiento de los *performance counters* del Pentium 4, así como informe sobre el diseño de un acceso básico a los mismos.

- **PEC 3** (entrega en aproximadamente 2/3 del plazo de realización del TFC): software ejecutable para acceso a los contadores (básico, sin discriminación de procesos), así como informe sobre el diseño del acceso a información sobre un determinado proceso.

Los informes contenidos en las PECs han servido de base para el contenido de la presente memoria, y tanto los plazos para elaborla como los destinados al desarrollo del software han estado prácticamente ajustados a lo planificado, con un cierto adelanto de casi dos semanas en las etapas finales por haber resultado la modificación del SO más sencilla de lo esperado.

1.5 Productos obtenidos

Dadas las características del Trabajo que nos ocupa, el producto final del mismo es el software implementado para el acceso al hardware de monitorización.

A partir de este momento utilizaremos el prefijo p4pc (iniciales de *pentium 4 performance counters*) en ficheros, constantes, variables y funciones como indicador del espacio de nombres en el que se desarrolla nuestro proyecto.

Se distinguen **dos productos**:

- Por una parte, las **modificaciones y adiciones al kernel**, efectuadas bajo dos formas específicas:
  - *Device driver* en forma de módulo cargable dinámicamente, cuyo fichero fuente es p4pc_dev.c. Corresponde al controlador del dispositivo que creamos para recoger las acciones sobre los contadores.
  - Modificaciones de funciones y otro código preexistentes en el kernel de Linux. En nuestro caso, hemos utilizado para nuestras modificaciones la versión 2.4.21, y se adjunta documento de texto identificando la totalidad de modificaciones y adiciones al kernel.
- Por otra parte, el software diseñado para el **espacio de usuario**, que se compone de:
  - Ficheros de cabecera y de código básicos y necesarios para que cualquier aplicación de usuario acceda al hardware de los contadores: p4pc_event.h (*header* principal), p4pc_event_catalog.h (catálogo de eventos definidos listos para usar por una aplicación), y p4pc_low_level_lib.c (funciones de bajo nivel para formateo de la información necesaria).
  - Ficheros de API: p4pc_api.c (funciones que componen nuestra API, propiamente dicha) y p4pc_TFC_api.h (macros que abstraen configuraciones y activaciones de eventos a un nivel todavía mayor, y de aplicación directa a nuestro TFC por tratarse de eventos de acceso a la caché L2).
  - Aplicación de usuario: matrix.c (aplicación para probar todo el código anterior, y que hace uso de la forma de almacenamiento de vectores en C para generar mayor o menor número de fallos al acceder a la memoria caché).

Además de aportarse el correspondiente código fuente, éste se transcribe como anexos en la memoria.

1.6 La memoria de aquí en adelante: el resto de contenidos

Tras este capítulo de introducción, se abordarán los siguientes temas:

- En el capítulo 2 veremos una introducción a la monitorización del rendimiento del software, así como a las estrategias para efectuar esta monitorización mediante hardware. Seguidamente, se materializa este análisis en el procesador Pentium 4 y se revisarán, de forma somera, las aplicaciones disponibles más conocidas para la configuración y acceso a esta unidad funcional del microprocesador.

- El capítulo 3 se dedica a las bases del diseño del software que compone nuestro TFC: revisaremos en primer lugar los principios de diseño del acceso básico a los contadores para, acto seguido, introducir los procesos y su planificación en Linux, introducción que servirá para completar la arquitectura de nuestro código para un acceso al hardware por proceso.

- Se dedica el siguiente capítulo al análisis detallado del código que se ha desarrollado, por lo que se pasará revista a las modificaciones del kernel, al driver, a los ficheros que componen la API y a las aplicaciones de usuario.

- En el quinto y último capítulo se resume y concluye nuestro proyecto con los comentarios finales de la memoria.

- Para terminar el documento, y tras un breve glosario de los términos específicos que se han empleado, y de la bibliografía consultada, se anexa la transcripción de todo el código desarrollado.

Para mantener homogéneos los capítulos, se ha preferido seguir un esquema de presentación diseño-código a uno diseño1-codigo1-diseño2-codigo2, más acorde este último al desarrollo funcional y temporal del proyecto pero, probablemente, de peor organización en la exposición.

CAPITULO 2

MONITORIZACION DEL RENDIMIENTO MEDIANTE HARDWARE: EL PROCESADOR PENTIUM 4.

2.1 Introducción

Conforme se ha ido avanzando en el desarrollo de las herramientas, lenguajes técnicas y procedimientos de programación, se iba manifestando claramente la necesidad de poder analizar el **rendimiento de las aplicaciones** en desarrollo, comparando los parámetros relevantes de su ejecución (habitualmente, parámetros de rendimiento en términos temporales) según se iba depurando el código o comparando su desenvolvimiento en diferentes sistemas o, incluso, diferentes arquitecturas.

Este análisis permitiría descubrir zonas o etapas de la aplicación que lastraran de forma significativa su desenvolvimiento. En particular, el acceso a los limitados recursos de una arquitectura (memoria, bus, sistema I/O, servicios del microprocesador…) y, lo que agrava sustancialmente el problema, la posibilidad del acceso concurrente a los mismos, origina incidencias o eventos en la ejecución del código que, si no son correctamente gestionados, pueden hacer que el rendimiento de la aplicación se degrade sustancialmente.

La detección de este código problemático (*hot spots*) se cimenta en la identificación, recuento y posterior análisis contextual de los eventos del tipo que acabamos de comentar (ampliaremos estos conceptos más adelante). Esta detección, junto con la posterior modificación de las zonas de código en cuestión para mejorar su actuación mediante el cambio o revisión de los algoritmos causantes del problema, es el objetivo primordial de las técnicas de **optimización (*tuning*) de software**.

La aproximación más evidente y sencilla a este problema implica el rediseño de las aplicaciones en cuestión para, desde dentro de las mismas y durante su ejecución, poder recoger la información necesaria que permita identificar las zonas de bajo rendimiento.

Esta aproximación, sin embargo, tiene algunos incovenientes importantes: las modificaciones que se efectúen en el código pueden inducir la creación de nuevos *hot spots* que compliquen aún más el problema del análisis del rendimiento; además, no siempre se tiene acceso al código fuente o la posibilidad de modificarlo.

Aunque existen otras alternativas para la detección de zonas de código con problemas (como la utilización de simuladores de procesadores, poco extendidos tanto por las restricciones impuestas por los fabricantes como por su carencia de garantías en cuanto a su precisión para operaciones complejas), la mejor solución encontrada se basa en la utilización de hardware destinado a la monitorización del rendimiento.

Las ventajas de este último enfoque con respecto a los anteriores son evidentes: de un lado, la aplicación en estudio no sufre modificaciones (aunque sí pueden existir ligeros cambios –los correspondientes drivers- en el sistema operativo que debe acceder al correspondiente hardware); por otra parte, permite obtener una precisión imposible de conseguir por otros medios.

Antes de profundizar algo más en este tipo de hardware, ampliaremos algo más los conceptos sobre eventos y su monitorización.

2.2 Eventos y medida del rendimiento

Hemos presentado ya de forma implícita el concepto de **evento** como incidencia relevante acaecida en la ejecución de un determinado código. Sin perder de vista nuestro objetivo de detección de código con bajo rendimiento, deberemos ser conscientes de que por evento entenderemos tanto ocurrencias de hechos que afectan directamente al rendimiento (como los fallos de acceso a la memoria caché, por ejemplo) como ocurrencias de hechos meramente estadísticos (como el recuento del número de instrucciones de un determinado código, por ejemplo). En cualquier caso, ambos tipos nos serán necesarios para crear métricas que nos permitan analizar y comparar los resultados que se obtengan.

Los eventos que interesan a nuestros fines se suelen clasificar en cinco **grupos**:

- Caracterización de código: los eventos de este grupo se utilizan para medir parámetros relacionados con los atributos de un determinado código, y no están vinculados a las características del procesador o la arquitectura (por ejemplo, el número de instrucciones de un determinado tipo).

- Memoria: estos eventos son utilizados para comprobar el rendimiento de la jerarquía de memorias en uso (por ejemplo, fallos al acceder a la caché de último nivel). Por su incidencia en el rendimiento, es uno de los grupos más usados.

- Predicción de saltos: este tipo de eventos cuantifica los éxitos o fracasos del mecanismo de predicción de saltos del procesador.

- Retenciones en etapas de segmentación: analizan cómo circula el flujo del programa por las diferentes etapas de segmentación del procesador, cuantificando las retenciones habidas por los diferentes tipos de riesgos.

- Utilización de recursos: analizan cómo y cuánto se utilizan el resto de recursos de la máquina (por ejemplo, accesos de un determinado tipo al bus).

¿Cómo se utiliza la medición de eventos para el análisis del rendimiento? De las dos técnicas básicas existentes que dan lugar a otros tantos **perfiles de estudio del rendimiento**, una de ellas se relaciona directamente con el análisis de eventos. Recordémoslas:

- **Perfil basado en tiempo**: facilita comprobar el tiempo invertido por la aplicación en sus diferentes secciones, permitiendo focalizar los esfuerzos de depuración en las zonas de mayor porcentaje de tiempo de ejecución. Para ello, se suele utilizar un temporizador o interrupción con el objeto de salvar con una determinada frecuencia el contador del programa. El análisis de este muestreo permitirá establecer un histograma donde, gracias a las diferentes densidades obtenidas, queden explícitas las zonas conflictivas.

- **Perfil basado en eventos**: en este caso debemos tomar un (o varios) evento de referencia para su análisis. El correspondiente hardware debe programarse para que a cada ocurrencia del evento (o bien tras un determinado número prefijado de ocurrencias) una interrupción salve el contador del programa. El histograma que obtenemos en este caso contrasta las ocurrencias del evento

contra la localización de las instrucciones o grupos de instrucciones que lo originan, lo que permite relacionarlas.

Como vemos, la primera técnica, aunque útil, peca de generalista. Si bien identificamos *hot spots*, no recibimos información adicional sobre los motivos de tal conducta ni, por lo tanto, sobre cómo puede ser corregida. Estos motivos deberán encontrarse con la segunda de las técnicas, la basada en eventos.

Pero tampoco esta última técnica está exenta de **problemas**. Las características de las diferentes arquitecturas que soportan hardware de recuento de eventos hacen que, en general, el registro de la instrucción que en última instancia provoca el evento no sea exacto. Los motivos de esta imprecisión se encuentran en la latencia existente entre la detección del evento y la actuación de la correspondiente interrupción, así como en la profundidad y complejidad tanto de las *pipelines* como de la ejecución superescalar con las que se resuelve la segmentación en los actuales procesadores. Así, en condiciones normales, la instrucción que se salva no es la que provoca el evento.

¿Invalida esto último el uso del análisis de eventos? No, desde luego. Por una parte, los histogramas obtenidos son generalmente válidos puesto que relacionan zonas de código con incidencias de eventos. El que la instrucción registrada esté unas decenas de posiciones más adelante de la que provocó el evento no es esencial normalmente. Lo importante es que la información estadística obtenida permita identificar las zonas de código mejorables, y eso puede hacerse en la mayoría de los casos aún con estas imprecisiones.

Pero, por otra parte, si se necesita absoluta precisión, los modernos procesadores van incorporando mecanismos que habilitan un muestreo por eventos preciso, almacenando correctamente la instrucción que origina el evento. El mecanismo es complejo, y suele venir acompañado de una pérdida de rendimiento importante en el procesador (por ejemplo, y según informa Intel, de alrededor del 20% para el Pentium 4).

2.3 Hardware para monitorización de eventos

Este tipo de hardware ha ido evolucionando (y lo continúa haciendo) conforme así lo han hecho los procesadores en los que reside. Su evolución da fe de la importancia creciente que este tema supone para los fabricantes, y ha pasado de ser un tema prácticamente indocumentado a figurar casi exhaustivamente en los manuales de cada diseñador.

No obstante, aún parece detectarse una cierta inmadurez de estas unidades cuando se comparan con las que conforman el núcleo duro de funcionalidades de cualquier procesador, y los errores documentados en su desempeño no son infrecuentes.

También se ha incrementado la complejidad en su manejo, que además no sólo no es específico de una arquitectura sino que varía sustancialmente entre diferentes modelos de procesador dentro de una misma familia.

Sin embargo, pueden distinguirse los siguientes **componentes** comunes en este tipo de hardware:

- **Contadores de eventos** o de rendimiento (*performance counters*): son registros específicos del procesador. Recogen la cuenta de las ocurrencias de un determinado evento seleccionado previamente. En algunos procesadores pueden utilizarse para establecer una "cuenta atrás" de ocurrencias tras la cual se dispararía la interrupción indicadora de tal hecho.

- **Registros seleccionadores de eventos**: son, también, registros especiales del procesador, normalmente accesibles sólo por el sistema operativo. Mediante una apropiada selección de sus bits permiten seleccionar qué evento o eventos van a ser monitorizados, así como la manera en la que van a serlo. Además de la elección del evento a analizar, y entre otras acciones, permiten:

    - Elegir, si está habilitado, entre muestreo preciso o impreciso de eventos.

    - Elegir el nivel de privilegio del código a analizar (normalmente, SO o usuario).

    - Habilitar el filtrado de resultados (más adelante se amplía este concepto).

    - Habilitar y conformar la gestión de las interrupciones generadas por el mecanismo de monitorización (ampliamos esta gestión más adelante).

- **Búfferes** de información adicional: en el muestreo por eventos se suele guardar información adicional. Ya hemos anticipado que, normalmente, se incluye el contenido del registro que contiene la instrucción que provoca el evento. Pero los actuales procesadores también guardan otros datos (volcado de los registros de propósito general, resto de registros de estado, zonas de memoria involucradas…) que necesitan ser almacenados en sus correspondientes búfferes. La dirección de inicio se mantiene en un registro específico.

- **Registros específicos auxiliares**: se suelen encontrar flags que indican si existe habilitación de determinadas características (por ejemplo, si se puede activar el muestreo preciso de eventos), así como punteros a áreas relevantes, como la comentada en el anterior párrafo.

Mientras que hasta hace pocos años la circuitería de los contadores era limitada y hasta cierto punto exótica, los últimos modelos de cada gama van superando de largo los límites inicialmente propuestos. Así, y como ejemplo en una familia de procesadores comunes, el Pentium III cuenta con dos registros contadores que permiten monitorizar algo más de ochenta eventos diferentes (obviamente, no de forma simultánea), mientras que el Pentium 4, como veremos, cuenta con dieciocho contadores y un conjunto de eventos individuales a analizar que sobrepasa de largo el centenar.

Además de la mejora en las prestaciones de este hardware y el incremento en el número de eventos que pueden ser analizados, hay otros **perfeccionamientos** y temas complementarios que es preciso mencionar:

- **Filtrado de resultados**: algunos eventos, cuando acaecen, pueden generar resultados mayores que 1. Se trata habitualmente de eventos que pueden darse varias veces simultáneamente en un solo ciclo en procesadores superescalares. Este resultado puede filtrarse (esto es: acumular o no al contador un valor de 1) dependiendo de si es mayor, igual o menor que un valor umbral de referencia (*threshold*).

- **Detección de flanco**: en ocasiones interesa conocer, no cuántas veces se detecta una ocurrencia de un determinado evento, sino cuándo empieza a darse una serie continuada de ocurrencias (o de no ocurrencias) y cuándo se acaba esta serie y/o empieza otra (por ejemplo: si tras un acierto se producen cinco fallos consecutivos al acceder a la segunda caché, un contador de eventos ordinario programado para recoger este evento acumulará un resultado de cinco, mientras que si se activa la detección de flanco, sólo se acumulará un resultado de uno al pasar del acierto al primer fallo). La detección puede ser de flanco de subida o de bajada.

- **Gestión de desbordamientos (*overflows*)**: los contadores, cuyo hardware normalmente no supera los 64 bits, pueden sufrir desbordamientos bien porque la acumulación de la cuenta ha rebasado su capacidad de almacenamiento o bien porque se les ha programado para hacerlo cuando alcanzan una determinada cantidad (cantidad que puede ser uno; es decir, se produce desbordamiento en cada ocurrencia del evento). En general, cada *overflow* origina una interrupción que debe servir para guardar la información necesaria para los propósitos buscados y devolver el sistema de contadores a su estado previo. Un complemento importante a esta gestión de desbordamientos es la posibilidad de, cuando se produce, activar un segundo contador en cascada para el mismo evento, con lo que se duplica la cantidad de ocurrencias de las que puede llevarse cuenta.

2.4 Performance counters en el Pentium 4

Puesto que el área en la que nos encontramos trabajando es propensa al uso de siglas, es conveniente recordar que los significados de todas con las que nos vamos a encontrar en adelante están recopilados en el correspondiente Glosario. En cualquier caso, se dará el significado de cada sigla previamente a su uso.

Antes de entrar en materia, debe puntualizarse que todo nuestro trabajo está basado en el procesador Pentium 4 básico. No entraremos en el estudio de la más reciente tecnología *Hyper-Threading* (HT en lo sucesivo) de los últimos procesadores, que, aunque para nuestros efectos no incluye cambios excesivamente drásticos, se sale de nuestros objetivos presentes.

Recordemos que el mecanismo de *performance counters* en el Pentium 4 supuso un salto adelante muy importante en relación al procesador de la familia inmediatamente anterior, el Pentium III, pero no sólo por el incremento en contadores o en eventos analizar, sino por las siguientes **características nuevas en la familia Pentium**:

- Posibilidad de **etiquetado (*tagging*) de instrucciones**: el Pentium 4 ofrece un mecanismo de etiquetado de instrucciones que, a su retiro (es decir, que se hayan ejecutado realmente) permite discriminar las instrucciones para su contabilización de forma no especulativa: si una instrucción origina un evento y, además, se ejecuta en su integridad, se etiqueta de tal forma que, a su salida de la última etapa de la segmentación, contribuirá a la acumulación del evento en el correspondiente contador. Por el contrario, si una instrucción, aún originando el evento, se elimina por haber sido erróneamente predicha, no hará variar la cuenta del evento.

- Posibilidad de ***Precise Event-Based Sampling*** **(PEBS)**: basada, precisamente, en la posibilidad de etiquetado de instrucciones, permite identificar con exactitud qué instrucción ha provocado el evento.

- **Perfiles de direcciones de memoria**: el mecanismo PEBS soporta, asímismo, un mecanismo adicional para identificar las zonas de memoria implicadas en los eventos relacionados con este recurso.

- Introducción de una nueva instrucción para lectura de los contadores de rendimiento, **RDPMC**, mucho más eficiente que la RDMSR si se limita la magnitud a contar a 32 bits.

Podemos establecer la abstracción de que la estructura básica del mecanismo de los *performance counters* en el Pentium 4 descansa en tres registros MSRs (*Model Specific Registers*), cada uno de los cuales sirve para representar una de las siguientes cuestiones: qué queremos contar, cómo queremos hacerlo y dónde queremos obtener el resultado.

Un análisis de mayor calado nos confirmará que se trata de una abstracción con ciertas debilidades puesto que cada registro contiene características cuya funcionalidad sería atribuíble a los otros. Por ejemplo: el propio contador donde recibimos los resultados puede ser programado para desbordar a nuestro antojo, funcionalidad que, con una abstracción rígida, debería estar incluída en el registro que recogiera la configuración sobre cómo queremos contar.

No obstante, se considera útil tal esquema y lo utilizaremos hasta donde podamos, conociendo sus limitaciones.

2.4.1 ESCRs o qué queremos contar:

Los ESCRs (*Event Selection Control Register*) son 45 MSRs que permiten al software configurar qué evento o eventos específicos se desea analizar.

De sus 8 bytes sólo son utilizados actualmente los 4 menos significativos. En ausencia de tecnología HT, los **bits activos** son los siguientes (comenzando desde el bit 0, el menos significativo, hasta el bit 31; los bits no mencionados se definen como reservados por Intel):

- Bit 2 – **flag USR**: si se activa, se cuentan los eventos cuando el procesador corre a niveles inferiores al del SO.

- Bit 3 – **flag OS**: se cuentan los eventos cuando el procesador corre a nivel de SO. Si se activan tanto este como el anterior, se contarán todas las ocurrencias del evento, sea cual sea el nivel de privilegios.

- Bit 4 – ***Tag Enable***: se activa el etiquetado de instrucciones (en realidad, microinstrucciones) para su contabilidad a su retiro.

- Bits 5 a 8 – ***Tag Value***: valor a asignar a las etiquetas anteriores que permitirá discriminar diferentes instrucciones etiquetadas por diferentes eventos.

- Bits 9 a 24 – ***Event Mask***: dentro de un grupo de eventos (ver el siguiente campo del ESCR) este campo de bits nos permite discriminar el o los eventos a contar.

- Bits 25 a 30 – ***Event Select***: 6 bits que permiten elegir el grupo de eventos en el que estamos interesados.

Veamos un ejemplo de configuración de este registro en particular: nuestro trabajo se centrará en el acceso a la memoria caché L2, por lo que deberemos, en primer lugar, encontrar en los manuales de Intel el correspondiente grupo.

En el correspondiente anexo del "*IA-32 Intel Architecture Software Developer's Manual – Volume 3: System Programming Guide*" encontramos las tablas de todos los grupos de eventos. La primera columna de estas tablas hace referencia al nombre de cada grupo. Para nuestros efectos, el grupo de interés es el **BSQ_cache_reference**, del que pasamos a reproducir su entrada completa en el Anexo B.

Entre otras informaciones, en los campos siguientes de la tabla encontramos lo que necesitamos: el campo Event Select deberemos cumplimentarlo con un 0Ch, mientras que para el campo Event Mask podremos elegir una máscara de 9 bits (son 9 los eventos individuales disponibles). Si, por ejemplo, queremos analizar el número total de referencias de lectura a la caché de segundo nivel (tanto exitosas como no), deberemos activar los bits 0, 1, 2 y 8, definidos en la documentación que manejamos como RD_2ndL_HITS, RD_2ndL_HITE, RD_2ndL_HITM y RD_2ndL_MISS, respectivamente. Como vemos, la nomenclatura utilizada por Intel es representativa del evento individual que referencia.

Suponiendo que sólo queremos contar eventos del nivel USR y que no utilizaremos etiquetado, es inmediato averiguar el valor que deberemos cargar en el ESCR mediante la instrucción adecuada (WRMSR). No obstante, para activar el circuito del contador deberemos trabajar con el CCCR correspondiente, que veremos a continuación.

Los 45 ESCRs están distribuídos en 4 grupos de entre 3 y 8 parejas. Cada ESCR está asociado a dos contadores (excepto los del último grupo, asociados con tres), mientras que cada contador puede tener hasta 8 ESCRs diferentes asociados a él, lo que equivale a decir que un contador puede hacerse cargo de hasta ocho grupos de eventos, obviamente no de forma simultánea. La tabla sobre la configuración de MSRs que se adjunta como Anexo A sirve para aclarar esta cuestión, aunque abundaremos en este tema cuando pongamos en breve todo el sistema en relación.

2.4.2 CCCRs o cómo queremos contar:

De forma semejante al anterior registro, en cada uno de los 18 (uno por cada contador) CCCRs (*Counter Configuration Control Register*) se utilizan sólo sus 8 bytes menos significativos (7, en realidad, ya que el primer byte menos significativo también figura reservado).

En este caso tenemos:

- Bit 12 – **flag *Enable***: es el interruptor que, en definitiva, habilita el recuento tan pronto se activa este bit. Para interrumpir el contador, basta con colocarlo a 0.

- Bits 13 a 15 – **ESCR *Select***: tres bits que identifican qué ESCR seleccionar del grupo de hasta 8 que pueden estar asociados al contador (o, mejor expresado, al par contador-CCCR).

- Bits 16 a 17 – reservados: de utilización con tecnología HT. En ausencia de ésta, deben activarse.

- Bit 18 – **flag *Compare***: si se activa, se habilita el filtrado de los resultados de la cuenta de eventos.

- Bit 19 – **flag *Complement***: si el filtrado está habilitado, efectúa el mismo comparando el resultado del evento con un valor umbral: si este bit está activo se acumulará una ocurrencia del evento si el resultado es menor o igual que el valor umbral; si no está activo, el contador se aumentará si el resultado del evento es mayor que el umbral.

- Bits 20 a 23 – ***Threshold***: valor del umbral para filtrado.

- Bit 24 – **flag *Edge***: si se activa el bit (y está activado el filtrado), se habilita la detección de flanco. En el Pentium 4 el flanco es de subida (*false-to-true*).

- Bit 25 – **flag FORCE_OVF**: si se activa, se fuerza al contador a un desbordamiento en cada incremento, es decir, cada vez que acaece el evento. Si no está activado, el desbordamiento es real (cuando se sobrepasa la capacidad de almacenamiento del registro contador o cuando se alcanza una cantidad de ocurrencias previamente estipulada).

- Bit 26 – flag **OVF_PMI**: si se activa, origina una PMI (*Performance Monitor Interrupt*) cuando se produce un desbordamiento del contador.

- Bit 30 – **flag *Cascade***: habilita la colaboración en cascada de dos contadores para duplicar la capacidad de recuento.

- Bit 31 – **flag OVF**: indicador que se activa cuando se produce un desbordamiento. Se debe desactivar explícitamente por software.

De igual forma que hacíamos con el ESCR, y siguiendo con el ejemplo presentado, el manual de Intel nos informa que para el grupo de eventos que nos ocupa el campo de 3 bits ESCR Select debe cumplimentarse con 07h. Asumiendo que, por ejemplo, no activaremos el filtrado, ni el recuento en cascada, que deseamos una interrupción en cada *overflow*, y que queremos empezar a contar tan pronto como carguemos esta información en el CCCR, tendremos el valor a grabar de forma inmediata.

Cada uno de los 18 contadores está asociado a un CCCR, y viceversa.

2.4.3 *Counters* o dónde queremos contar:

Los 18 contadores existentes se organizan en 9 pares. Cada par está asociado a un conjunto de hasta 8 grupos de eventos, lo que equivale a decir que está asociado a un máximo de 8 ESCRs. Veremos con mayor claridad estas relaciones en breve y, de nuevo, la figura del Anexo A clarifica este tema.

De los 64 bits de cada contador sólo pueden utilizarse 40, por lo que, considerando la posibilidad de asociar dos contadores en cascada, la cantidad máxima de ocurrencias que pueden registrarse de un evento antes de tener que volver a empezar a contar queda fijada en $2x2^{40}$.

Como hemos anticipado, la lectura de los contadores con la instrucción RDPMC es más eficiente que con RDMSR, sobre todo si sólo se utilizan los 32 bits menos significativos del registro.

Además, los registros contadores son programables para su desbordamiento a voluntad del software. Para ello, deben ser cargados (por la instrucción WRMSR) con la magnitud de ocurrencias, con signo negativo (complemento a 2), que deseamos de un determinado evento antes del desbordamiento. El contador irá acumulando desde la cantidad inicial hasta llegar a 0, momento en que se producirá el overflow.

2.4.4 La visión de conjunto:

Llegados aquí debemos tener una idea formada de cómo actúan individualmente cada uno de los tres registros necesarios para analizar un evento. Pero, sabiendo que cada MSR tiene una dirección propia, ¿cómo saber cuál de los ESCR y CCCR configurar para contar un determinado evento, y en qué contador obtendremos los resultados?

Para verlo con detenimiento hemos pedido prestada la tabla del manual de Intel "*Events on Intel Pentium 4 Processors,…*", y que adjuntamos como primer anexo, donde podemos confirmar los siguientes puntos, la mayoría ya anticipados:

- A cada contador le corresponde un CCCR, y viceversa.

- Tanto los 18 contadores como los 45 ESCRs se distribuyen en 4 grupos que se denominan según el primer ESCR de cada grupo.

- Cada ESCR está duplicado (excepto el de la unidad SSU), lo que permite analizar simultáneamente dos eventos (o dos conjuntos de eventos) pertenecientes al mismo grupo.

- Cada uno de los 4 grupos de ESCRs tiene un máximo de 8 pares de este tipo de registros.

- Cada par contador/CCCR está asociado con un conjunto de entre 3 y 8 ESCRs.

- Cada ESCR está asociado a dos pares contador/CCCR, excepto los pertenecientes al último grupo, que lo están con tres pares (son los contadores que permiten el PEBS).

Por lo tanto, y siguiendo con nuestro ejemplo, veamos qué debemos hacer para conocer las correspondientes direcciones de cada uno de los MSRs a los que deberemos acceder para configurar el recuento deseado:

1. En primer lugar, obtenemos de las ya mencionadas tablas de grupos de eventos (Anexo B) las restricciones en cuanto a ESCRs que pueden utilizarse del grupo que estamos estudiando. Como podemos ver, el grupo BSQ_cache_reference está restringido a usar los ESCRs BSU_ESCR0 y BSU_ESCR1. Un vistazo a la tabla de MSRs (Anexo A) nos indica que estos dos registros son accesibles en las direcciones 3A0h y 3A1h. Escogeremos arbitrariamente uno de ellos.

2. Si suponemos que elegimos el primero, BSU_ESCR0, tendremos para elegir dos pares contador/CCCR donde almacenar el resultado: el CCCR/Counter0 y el CCCR/Counter1. La misma tabla del anexo 1 nos ofrece las direcciones tanto del contador como del CCCR que elijamos finalmente.

En definitiva: tenemos las direcciones de los tres MSRs implicados; además, conocemos también el contenido con el que debemos cargarlos según hemos visto en 2.4.1, 2.4.2 y 2.4.3.

Tres instrucciones WRMSR más tarde (dos, si no tenemos que configurar el propio contador) nuestro sistema de *performance counters* deberá estar funcionando a pleno rendimiento, registrando en el contador seleccionado el recuento de todas los accesos a la caché de segundo nivel.

2.5 Herramientas existentes para el acceso a los performance counters

El campo del análisis basado en hardware del rendimiento de las aplicaciones, aún estando bastante extendido en ambientes educativos y de desarrollo, no se caracteriza por generar volúmenes de software importantes y de aplicación generalizada sobre el acceso y configuración de este tipo de hardware.

De las búsquedas realizadas en Internet y en la bibliografía oportuna, apenas son destacables una decena de aplicaciones. Que, además, cubran el hardware de monitorización de rendimiento del Pentium 4, sólo se han encontrado reseñables las siguientes:

2.5.1 Abyss:

abyss_dev es un *device driver* para Linux que viene siendo desarrollado desde el año 2000 por Brinkley Sprunt, aunque utiliza trabajos previos sobre la misma materia de otros programadores.

El programador fue hasta 1999 ingeniero responsable del desarrollo de los *performance counters* del microprocesador Pentium 4, en aquellos tiempos denominado *Willamette*.

abyss_dev es un trabajo completo sobre el acceso y gestión de *todos* los recursos que conforman el hardware de los performance counters en el Pentium 4, incluyendo la habilitación de *Precise Events-Based Sampling* y la correspondiente gestión y análisis de los búffers de memoria que este mecanismo provee, así como la completa gestión de las interrupciones por desbordamiento de los contadores.

Esta visión completa del hardware motiva que la interfaz con el driver sea compleja, por lo que el software hace uso de ioctl() para la comunicación entre el espacio de usuario y el de kernel. Además, para la gestión de las interrupciones, necesita parchear el kernel con código adicional.

La correspondiente API está pobremente (si algo) documentada, remitiéndose a las fuentes en C o Perl de sendos *front-ends* escritos para atacar al driver.

La limitación más evidente es que sólo un proceso puede acceder al driver cada vez que se usa.

Tampoco está preparado para muestreo bajo tecnología *Hyper-Threading*, lo que parecería indicar un cierto abandono o desinterés por el proyecto.

2.5.2 PCL:

Siglas que corresponden a Performance Counter Library, proyecto desarrollado en Alemania por varios autores como parche para el kernel del SO.

A diferencia de abyss, PCL está orientado a obtener servicios de muestreo de la mayoría de los principales procesadores del mercado, incluyendo la familia Intel Pentium y, en particular, el Pentium 4.

No obstante, esta orientación generalista impide una completa aproximación al hardware de monitorización del Pentium 4, tal y como se consigue en abyss. Así, por

ejemplo, no parece dar soporte a PEBS, ni a la gestión de interrupciones ni a otros opciones más complicadas que no sean la activación, lectura y desactivación de los *performance counters*.

Además, parece un proyecto abandonado puesto que aparenta no tener actividad desde hace dos años y, de hecho, no está terminada la implementación de la librería completa de PCL al Pentium 4.

La interfaz con el núcleo es sencilla y bien definida, con las necesarias y ya anticipadas funciones para activar los contadores pasando como parámetros los eventos a muestrear, desactivar los contadores en uso y leer los resultados.

2.5.3 OProfile:

Proyecto en curso y, por lo que se ve, activo, con objetivos similares a los que acabamos de ver (abarca la mayor parte de los procesadores más extendidos).

Su principal estrategia se basa en la obtención de perfiles de rendimiento basados en tiempo (ver apartado 2.2) mediante mecanismos no modificativos del SO soporte (Linux), por lo que su monitorización se extiende a todos los procesos existentes en el sistema sin ser capaz de discriminar análisis por proceso.

Se arranca desde la línea de comandos, con argumentos para representar los diferentes eventos o contadores a analizar.

Sus desarrolladores califican el estado actual del software como “alfa”, por lo que son de esperar mayores funcionalidades en el futuro. De momento, no soporta PEBS ni otras funciones avanzadas, ni tampoco accede a la totalidad de los contadores del Pentium 4.

2.5.4 PAPI:

Siglas de Performance API. Se trata, como su nombre indica, de una *Application Programming Interface* también generalista, dirigida a soportar el acceso a los contadores de rendimiento de la mayoría de los procesadores actuales más significativos, incluyendo los nuevos de 64 bits.

Es multiplataforma, y soporta los principales sistemas operativos. En Linux, el desarrollo se efectúa como un patch para el kernel (2.2, 2.4 y 2.6).

Está siendo desarrollada activamente (va por su tercera versión, cuya última actualización es de hace unos días) por el *Innovative Computing Laboratory* de la Universidad de Tennessee, y está financiada, entre otros, por IBM, Intel, Microsoft y el Departamento de Defensa de los EE.UU..

Ofrece una interfaz (tanto para Fortran como para C) de dos niveles:

- La de mayor nivel mantiene una lista predefinida de eventos, normalmente comunes en todo el rango de procesadores contemplado. Ofrece, además, las operaciones básicas a efectuar con este hardware: iniciar un contador, acceder a sus resultados y detenerlo.

- La interfaz de bajo nivel, completamente programable, además de ofrecer el uso de los eventos predefinidos permite que el usuario defina otros de su interés y que sean nativos del procesador en uso. Por otra parte, facilita el acceso al hardware para que éste aporte información sobre sus prestaciones y provee opciones para la gestión de los desbordamientos.

Aunque su uso, por la extensión y número de funcionalidades, es complejo, los desarrolladores aportan documentación formateada en diferentes niveles y elaborada para su estudio desde diversos puntos de vista, con una cobertura completa y precisa de toda la API. El programador que necesite hacer uso de ésta no debería encontrar especiales dificultades en sus proyectos.

Ya desde su propia página web se notan los recursos que están siendo dedicados a este software y, en mi opinión y hasta donde he podido analizar, se trata del referente principal de este tipo de aplicaciones, al menos por lo que se refiere a software de código abierto.

2.5.5 VTune:

Es la herramienta gráfica desarrollada por la propia Intel para el análisis de los contadores de sus últimos procesadores, que ya alcanza su séptima versión, y es la única no gratuita ni de software libre de las aquí contempladas (su precio, a diciembre de 2004, ronda los 700 US$). Existen versiones para Linux y Windows.

Mantiene un conjunto algo menor de un centenar de eventos predefinidos, que junto con una interfaz gráfica para su gestión, configuran una herramienta completa, muy potente y relativamente sencilla de usar, más orientada, en mi opinión, al análisis de rendimiento en el desarrollo de aplicaciones comerciales que, por ejemplo, a la monitorización de entornos de computación paralela o supercomputación, donde PAPI *puede* (y debe recalcarse el término *puede* porque este análisis es necesariamente superficial) resultar más interesante.

La posible configuración de los contadores es completa: desde PEBS hasta completo filtrado, pasando por el soporte de tecnología *Hyper-Threading*, y todo ello con la posibilidad de muestreo bajo SMP sin ningún tipo de incidencias.

Pero no sólo se trata de un envoltorio gráfico que facilita la configuración y gestión del hardware: en realidad, VTune es una completa aplicación de ayuda a la depuración de programas y orientada al estudio de la mejora de su rendimiento utilizando todos los recursos posibles, siendo el uso de los *performance counters* tan sólo una parte (aunque importante) de éstos.

Dado un determinado código, y gracias a las posibilidades de enlazar las funcionalidades de VTune con herramientas propietarias como Visual Studio, su análisis es poco menos que espectacular: la interfaz ofrece, con pocos clicks de ratón, el acceso directo a las diferentes funciones del código que suponen *hot spots*, incluyendo estadísticas del uso del procesador prácticamente por línea o sentencia ejecutable del programa, para, una vez localizadas las sentencias problemáticas, ofrecer incluso consejos sobre cómo modificar el código para mejorar el rendimiento.

CAPITULO 3

EL DISEÑO DEL SOFTWARE DE ACCESO A LOS PERFORMANCE COUNTERS DEL PENTIUM 4.

3.1 Introducción

Como ya hemos visto en capítulos anteriores, el diseño del software del TFC que nos ocupa ha pasado por **dos etapas** bien diferenciadas. En la primera se ha pretendido sentar las bases de un acceso a los contadores que venimos calificando como básico, y que no discrimina el recuento de los eventos por proceso.

En la segunda etapa del diseño se trabajó con el objetivo de aislar la contabilización de las ocurrencias de los eventos en una "métrica-por-proceso", es decir, recontando los eventos para un determinado código cuyo alcance es marcado por una aplicación de usuario.

Es reseñable que cada una de estas dos etapas ha contado con su propio mecanismo de diseño.

En la **primera** se han intentado seguir, aunque con cierta liberalidad, los procesos iniciales propios del diseño de software orientado a objetos. No es un contrasentido utilizar esta orientación en un software puramente de sistemas y estructurado como es éste. Al contrario: el kernel de Linux está lleno de ejemplos de orientación a objetos en sus unidades funcionales más importantes y conocidas.

Por lo tanto, nos ayudaremos de los diagramas propios de las primeras fases del diseño orientado a objetos para aclarar *qué* debemos hacer. Fue particularmente útil esta aproximación en unos momentos iniciales en los que no se tenía nada claro el *cómo* resolver las funcionalidades necesarias del código puesto que, como ya se ha comentado, se avanzaba sobre territorios de hardware y software completamente desconocidos.

El enfoque del diseño de la **segunda parte** es muy diferente. En ese momento ya se conocían los mecanismos tanto de los contadores del procesador como los del *char driver* bajo Linux. También se había repasado en profundidad el ciclo de vida de los procesos en este SO. Por lo tanto, el diseño como tal apenas existió, y quedó limitado prácticamente a la explicitación de las modificaciones a efectuar al software disponible (el kernel por un lado, y el driver diseñado e implementado anteriormente para el acceso básico a los contadores por otro) que, además, tampoco resultaron numerosas.

Veremos seguidamente el diseño de la primera etapa, para terminar este capítulo introduciendo las bases de los procesos bajo Linux y revisando el diseño de la segunda etapa.

3.2 La aplicación para acceso básico a los contadores

Recordaremos, en primer lugar, nuestros objetivos de forma inequívoca. De momento, como primera fase del Trabajo de Fin de Carrera, pretendemos elaborar una aplicación que nos permita contar los fallos de la memoria caché L2 que se produzcan en un procesador Pentium 4, en un periodo de tiempo arbitrario y por cualquiera de los procesos en ejecución.

No obstante la concreción de este objetivo, nos fijamos otro adicional que consiste en estructurar la aplicación y la información que maneja de tal forma que sea fácilmente aprovechable para el análisis de otros eventos.

Para conseguir nuestros fines debemos configurar determinados registros del procesador según figura en los manuales de programación del fabricante. Una vez establecida esta configuración podremos obtener el resultado del recuento en otro de los registros específicos (contador).

Visto lo anterior, nuestra aplicación deberá:

1. Ser capaz de manejar la información que defina el evento “fallos en acceso a la caché L2 del Pentium 4”.

2. Ser capaz de configurar los correspondientes registros del procesador para activar el contador de este evento.

3. Ser capaz de obtener el resultado que ofrecerá el contador.

4. Ser fácilmente configurable para contemplar el análisis de otros eventos de los posibles que pueden monitorizarse en el Pentium 4.

3.2.1 Requisitos

Aunque la técnica de programación que utilizaremos es estructurada, ya hemos anticipado que puede resultarnos útil para su diseño inicial una aproximación orientada a objetos. Por lo tanto, para resaltar e identificar con mayor facilidad los requisitos de nuestra aplicación apuntados en el anterior apartado, nos ayudamos de un **diagrama inicial del dominio** en estudio, que podría ser el siguiente:

Es inmediato identificar varios objetos cuyo contenido y relaciones se resumen de la forma siguiente:

- Un objeto **evento** resumirá qué evento o eventos pretendemos analizar, así como la manera en la que vamos a hacerlo (vinculado a los requisitos 1 y 4 relacionados en el anterior apartado).

- Necesitamos un recurso que, de forma no ambigua, transforme cualquier instancia del anterior objeto en conjuntos de bits estructurados de manera rígida y que alimenten los correspondientes registros del procesador para obtener los resultados buscados. A este recurso lo denominamos, de momento, **librería de conversión a bajo nivel** (relacionado con los requisitos 1, 2 y 4).

- Debemos contar con una **aplicación de usuario** que se limite a escoger de un posible catálogo el evento a muestrear y, tras su conversión a bajo nivel con la correspondiente función de la librería anterior, lo encamine hacia el procesador (vinculada a todos los requisitos).

- El punto de entrada al sistema operativo será un módulo configurado como **device driver** cargable a voluntad del usuario. Una vez cargado pertenecerá al espacio del kernel y su código se ejecutará al mayor nivel de privilegios. Por lo tanto será capaz de configurar los registros del procesador necesarios para la función de recuento del evento y leer la información del contador (requisitos 2 y 3).

- Por último, los **registros del procesador** involucrados en estos procesos son las piezas de hardware para la que debemos implementar el device driver: deberemos escribir en este dispositivo las configuraciones necesarias y podremos leer de él el resultado pertinente (también relacionados con los requisitos 2 y 3).

A continuación entramos en detalle sobre cada uno de los objetos/elementos vistos. La aproximación mediante objetos que hemos elegido nos permitirá resaltar las estructuras de datos o variables relevantes de cada objeto (atributos), así como las posibles funciones o procedimientos que hagan uso de éstas (métodos), y que necesitaremos implementar en nuestro software.

Evento:

Se configura como la pieza clave de nuestra aplicación. Corresponderá a este objeto definir tanto el sujeto a analizar (por elección del grupo de eventos y de la máscara de eventos correspondiente) como el mecanismo que se va a utilizar para hacerlo (por parametrización de las diversas opciones de muestreo o análisis).

El estudio que se ha efectuado sobre la implementación de los performance counters en el Pentium 4 nos aporta las siguientes bases para distinguir los posibles componentes o atributos:

- **Identificación del evento**: hemos visto que en el Pentium 4 los eventos se agrupan en conjuntos según funcionalidades, y sus elementos son, precisamente, los eventos individuales que el fabricante ha implementado. Es de reseñar que un determinado evento puede estar compuesto de uno o varios eventos individuales (por ejemplo, el evento “número de referencias de lectura

a caché" se compone de la suma de los eventos individuales que definen tanto los aciertos como los fallos sobre dicha caché). En este atributo de identificación, por tanto, deberá constar tanto el grupo al que pertenece el evento como la correcta configuración del flag que lo define dentro del grupo.

- **Nivel de privilegio** a analizar: una vez definido el evento, podemos elegir si lo muestreamos cuando el procesador corre en en un nivel de privilegios de SO, de usuario o de ambos.

- **Etiquetado de µops**: decidiremos si procedemos al etiquetado de instrucciones, que nos permitirá un análisis no especulativo con las instrucciones efectivamente ejecutadas.

- **Filtrado**: el usuario debe poder filtrar los resultados por comparación con un umbral, así como efectuar análisis de detección de flanco.

- **Desbordamientos**: este atributo lo compone un conjunto de recursos que marcan la política de interrupciones a generar por el sistema contador.

- **PEBS**: permite a la aplicación de usuario habilitar la opción de muestreo preciso de eventos.

Aunque nuestra aplicación, como se ha comentado, se dirige al estudio de un solo tipo de eventos, es conveniente crear las estructuras de datos necesarias que permitan elaborar con facilidad un catálogo de diferentes eventos listos para ser utilizados por un código de usuario. Estas estructuras de datos, que se incluirán en el correspondiente archivo de cabecera, se instanciarán en nuestro caso en el evento en estudio.

Librería de conversión a bajo nivel:

Con las estructuras de datos apuntadas en el último apartado abstraemos una rígida configuración de bajo nivel (configuración de determinados registros específicos mediante bits) fijada por el fabricante del procesador. En consecuencia, tendremos que disponer de un mecanismo que permita traducir las estructuras de información del nivel de usuario a las del procesador.

Parece procedente implementar este mecanismo como una librería cuyo elemento principal sería la función encargada de la traducción entre niveles, y cuya salida deberá volcarse a un nuevo recipiente de datos de bajo nivel.

Aplicación de usuario:

Si, como planeamos, podremos disponer de un catálogo de eventos y de una librería auxiliar que permita traducirlos a los vectores de bits equivalentes, la aplicación de usuario se simplifica sobremanera: toma un evento, lo transforma en los vectores correspondientes con la ayuda de la librería y lo remite al procesador a través del sistema operativo mediante una llamada `write()` dirigida al correspondiente controlador de dispositivo. El acceso al resultado lo conseguirá mediante una llamada `read()` al driver.

Controlador de dispositivo:

La programación de módulos para el kernel de Linux está fuertemente estructurada en torno a diversos procedimientos prácticamente universales para la mayoría de los drivers (respuestas a las funciones de sistema `open()`, `close()`, `write()`, `read()`, e inicialización y liberación del dispositivo, principalmente). Se caracteriza por la utilización de las llamadas de sistema del SO y su imposibilidad de acceso a la librería estándar.

Para la aplicación que intentamos desarrollar serán de relevancia los siguientes atributos:

- Buffer de entrada de bits: el espacio de usuario dirigirá al dispositivo una llamada `write()` conteniendo los vectores de bits destinados a la configuración de los registros específicos del procesador. Estos vectores serán recogidos en los correspondientes búffers, que servirán de parámetros para la instrucción WRMSR (escritura en los registros específicos).

- Buffer de salida del resultado: la función `read()` al dispositivo provocará la lectura con RDPMC o RDMSR del contador, que deberá ser salvada para su envío al espacio de usuario.

Vamos a reseñar a continuación tres de los métodos que deberemos utilizar en este dispositivo. Dos de ellos, predefinidos en el sistema Linux y que traemos aquí para tratar de definir el mecanismo del diseño, nos ayudaran a la circulación de la información entre el usuario y el procesador, mientras que un tercero a implementar nos permitirá salvar las limitaciones de `printk()` y escribir a una consola X:

- `copy_from_user()`: función de bajo nivel predefinida en el correspondiente fichero cabecera de las fuentes del kernel. Nos permite pasar un buffer del espacio de usuario al de kernel.

- `copy_to_user()`: contrapartida de la anterior que permite pasar información del espacio de kernel al de usuario.

- `printkf()`: función a implementar que nos permitirá llevar los mensajes del núcleo hasta una consola bajo el sistema X, lo que nos ahorrará la consulta a los ficheros de log que se usan bajo `printk()`.

Para finalizar, este módulo contará con las características adicionales siguientes:

- Sólo se permitirá una instancia del mismo. El análisis concurrente de eventos, si no es definido correctamente, puede provocar resultados poco fiables. Además, de esta forma se simplifica el diseño y la implementación de la aplicación[1].

[1] Si se analiza el producto software final, y como veremos más adelante, esta restricción fue eliminada por improcedente en posteriores fases del desarrollo. Se recoge aquí tal y como se explicitó en los estudios iniciales del diseño, manteniendo así su congruencia con la documentación enviada a revisión en su momento al tutor del TFC.

- El módulo sólo deberá activarse en presencia de un procesador Pentium 4, ya que, como sabemos, el mecanismo de monitorización de eventos es fuertemente dependiente del tipo de procesador. Definida como de menor prioridad, la detección de la CPU será una de las últimas características a implementar en la aplicación, tras la correcta implementación de las principales funcionalidades.

Procesador:

En nuestro diagrama de dominio es el único objeto con entidad física. De él nos interesan fundamentalmente los registros vinculados a la definición, puesta en marcha del muestreo y obtención de resultados de un determinado evento:

- **ESCR**: registro MSR cuya configuración, en líneas generales, viene a conformar la identificación del evento o eventos a analizar. Para nuestros efectos, será accesible por el usuario mediante `write()`, función que en definitiva hará operar una WRMSR sobre él.

- **CCCR**: registro MSR que recogerá, también en líneas generales, la forma en la que pretendemos que el evento identificado en el ESCR sea

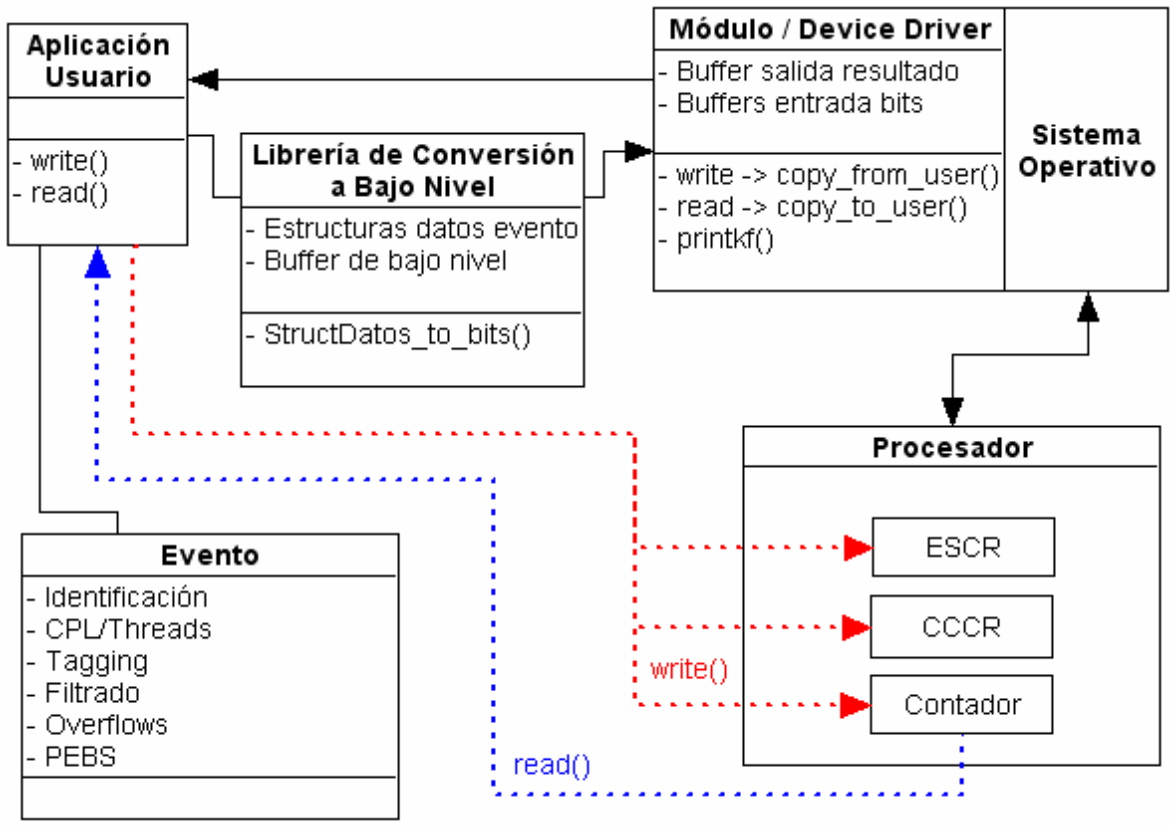
  analizado. De la misma forma que el ESCR, es accesible para la aplicación de usuario mediante `write()`.

- **Contador**: registro MSR que soporta, bien el número de ocurrencias de un evento, bien una cuenta atrás de las que quedan para que el sistema emita una interrupción tras haberse registrado un número determinado de ocurrencias del evento. Aunque en un principio nuestra aplicación sólo contempla la acumulación de ocurrencias del evento en estudio (y el acceso a esta magnitud mediante RDPMC o RDMSR, desencadenadas desde un `read()` originado en la aplicación de usuario), este registro contador también soporta su escritura (como entero negativo) para establecer el número de ocurrencias que originan la PMI (interrupción de monitorización de rendimiento).

Puede obtenerse mayor información sobre estos registros, su configuración y su funcionamiento en el correspondiente apartado de las notas sobre los *performance counters* del Pentium 4, en el capítulo 2.

3.2.2 El diagrama de dominio como resumen del diseño:

Basado en el diagrama simplificado con el que hemos empezado, obtenemos el siguiente, más completo, pero en el que seguimos disfrutando de la flexibilidad y de la libertad formal de este tipo de diagramas:



Quedan explícitos los elementos más interesantes y que, de una forma u otra, caracterizarán nuestra aplicación. En definitiva, serán los que configuren las principales partes de su lógica.

Además de los métodos y atributos que acabamos de detallar, se aprecia el flujo de datos entre los componentes: de la aplicación de usuario al procesador, intermediando la correspondiente librería de conversión entre el usuario y el dispositivo; y del procesador a la aplicación, con la sóla intermediación, en este caso, del driver por delegación del sistema operativo.

Por último, se identifican con líneas de puntos las relaciones últimas que nuestra aplicación debe establecer entre el usuario y las unidades de monitorización de rendimiento del procesador.

3.2.3 Software:

Como resumen de todo lo anterior, el producto de nuestra **primera fase del TFC** deberá contener:

- El fichero fuente del correspondiente módulo, cuyo código objeto, una vez compilado, será añadido de forma dinámica al núcleo del sistema operativo.
- Un fichero de cabecera que recoja las diferentes estructuras de datos con las que representar cualquier (*objeto del tipo*) evento.
- Una función de librería utilizable por la aplicación de usuario para convertir las estructuras de datos que definen un evento en los vectores de bits reconocibles directamente por el sistema de monitorización de rendimiento del procesador.

- Un fichero de cabecera que recoja un catálogo de eventos predefinidos mediante instancias del tipo evento.

- Una aplicación de usuario que soporte todo lo anterior.

3.3 Diseño del acceso discriminado por proceso

Tras haber elaborado un marco de trabajo que nos permite un acceso básico (pero preparado para mayores funcionalidades)[1] a los contadores de rendimiento del Pentium 4, procede dar un paso más y dejar completo nuestro software para que puedan monitorizarse los eventos originados por un determinado código o proceso.

Tal y como se ha diseñado el hardware de los performance counters, la única discriminación que nos es factible realizar, de una forma sencilla y rápida, y por mediación del driver, es la que se refiere a la monitorización de procesos de usuario, de sistema operativo o de ambos simultáneamente. Quiere esto decir que no nos es posible indicar al procesador para qué determinado proceso pretendemos que el contador esté activo, funcionalidad ésta absolutamente necesaria para analizar el comportamiento del proceso o código en estudio.

Una posible solución a la monitorización por proceso podría pasar por generar una interrupción en cada ocurrencia de un determinado evento. Configurar los correspondientes MSRs para conseguirlo es tarea sencilla. De esta forma, en cada ocurrencia del evento en estudio utilizaríamos la rutina de servicio a la interrupción para registrar qué proceso (fácilmente identificable a través de la macro `current`) es el que ha originado el evento. Como obtendríamos interrupciones por las ocurrencias de todos los procesos en ejecución, al final de la monitorización deberíamos identificar y contabilizar las producidas por el proceso en estudio.

El principal problema de esta aproximación es que no es eficiente. Si contemplamos, por ejemplo, los eventos que estudiamos en este TFC, veremos que acaecen decenas o centenares de miles de veces por segundo. En el caso del acceso a la caché L2, se obtienen magnitudes de varios millones de ocurrencias por segundo. Si se muestrearan varios eventos de estas características de forma concurrente, el tiempo de servicio a la interrupción sería tan alto que la pérdida del rendimiento del sistema sería, probablemente, inaceptable.

Aún con aproximaciones parciales (por ejemplo, activar la interrupción no en cada ocurrencia de un evento, sino tras un número determinado de éstas), quedaría por resolver de forma eficiente el problema del almacenamiento y posterior análisis de las ocurrencias de los eventos, que implicaría guardar registro de los varios miles de procesos que pueden coexistir en un sistema y que, potencialmente, podrían generar los eventos en muestreo.

Resumamos qué es lo que **debemos conseguir**:

- Nuestro sistema tiene que poder discriminar qué proceso o procesos se monitorizan y cuáles no.

- Si un proceso debe monitorizarse, deben activarse los correspondientes contadores en el momento en que el proceso va a empezar (o reanudar) su ejecución.

[1] Debe recordarse que este diseño no se realiza en las primeras fases del TFC, sino una vez implementado de forma satisfactoria el software para el acceso básico a los contadores. Para el estudio del correspondiente código, ver el capítulo siguiente.

- Si un proceso está siendo monitorizado y termina (o es retirado por el planificador del SO), los contadores activos deben detenerse.

- Evidentemente, si un proceso no está marcado para ser monitorizado, nada de lo anterior le concierne.

De esta forma, un ligero análisis de estos requisitos nos lleva enseguida a la conclusión de que una mejor solución pasa por la modificación del núcleo del sistema operativo, opción ésta con la que intentaremos conseguir nuestros objetivos. Deberemos conocer, por tanto, los mecanismos que utiliza el kernel de Linux para gestionar los procesos y su planificación.

3.3.1 Procesos y planificación de procesos en Linux

Un proceso, o su objeto vinculado hilo, se abstrae en el núcleo de Linux mediante la estructura de datos `task_struct` (definida en `include/linux/sched.h`), que recoge el grueso de la información necesaria sobre cualquier proceso activo. Esta estructura se almacena en un extremo de la pila de cada proceso, donde es fácilmente localizable y accesible (a partir de la versión del kernel 2.6, se crea una nueva estructura, `thread_info`, que es poco más que un apuntador a `task_struct`; ésta última se almacena fuera de la pila, mientras que `thread_info` permanece en ella, aunque ocupando un espacio mucho menor que el ocupado por `task_struct`).

Es bien conocido que Linux es un SO multitarea, lo que significa que el kernel debe ser responsable de:

- Llevar la cuenta y registro de todos los procesos activos en el sistema en un momento determinado.

- Asignar a cada proceso una porción de tiempo de procesador, de acuerdo a un mecanismo de prioridades asignadas y/o calculadas que denominamos planificador, dentro de cuyas tareas se encuentra, también:

- Realizar la tarea de retirar un proceso que ha agotado su cuota de procesador y empezar a ejecutar o reanudar la ejecución del código del proceso de mayor prioridad en ese instante, de forma transparante a las aplicaciones de usuario.

De una forma básica, Linux almacena los procesos en dos tipos de colas: de un lado, los procesos listos para su ejecución se almacenan en *run queues*, mientras que los procesos “dormidos” (normalmente a la espera de alguna operación de I/O) se guardan en *wait queues*. Estos últimos procesos son trasladados a las colas de ejecución cuando despiertan, es decir, cuando se ha producido el evento por el que estaban esperando.

El planificador de Linux, por lo tanto, actuará en cada cambio de contexto (detención de la ejecución de un proceso para iniciar la de otro) eligiendo el proceso de mayor prioridad de entre los que están en las *run queues*. Por su parte, el cálculo de la prioridad de cada proceso se realiza de forma dinámica, y su resultado es fuertemente dependiente del grado de actividad I/O del proceso.

El mecanismo de planificación actúa de la siguiente forma: cuando el proceso en ejecución sobrepasa el tiempo de procesador asignado, o cuando se ha despertado un proceso de mayor prioridad que el que se está ejecutando, el sistema activa el flag

`need_resched` de la `task_struct` del proceso en ejecución. Este flag es comprobado cada vez que la ejecución se devuelve al espacio de usuario desde una llamada de sistema o volviendo de la ejecución de una rutina de interrupción. Así, si el sistema encuentra activo el flag en una de esas circunstancias, llama a la función que se ocupa de las actividades de planificación y cambio de contexto.

La función que realiza en el núcleo de Linux estas actividades es `schedule()` (declarada y definida en `kernel/sched.c`, y no en un fichero de cabecera, para que no exista una interfaz evidente y proteger en lo posible un código tan sensible). En función de la versión del núcleo, y en cada cambio de contexto, `schedule()` puede llamar a la función `context_switch()` o tener el código de esta última función integrado en la propia `schedule()`. En última instancia, ésta o `context_switch()` acaban llamando a la función `switch_to()` (`include/asm/system.h`), función de bajo nivel que se ocupa de salvar el estado del procesador y registros (el contexto) del proceso que se retira de ejecución y de reponer el del proceso que se va a ejecutar.

Por lo tanto, para conseguir los objetivos funcionales que hemos resumido anteriormente, debemos centrarnos en dos de los elementos mencionados:

- La estructura **`task_struct`** nos servirá para soportar los correspondientes atributos que permitan identificar si el proceso que representa está siendo monitorizado por algún *performance counter*, a la vez que deberá identificar los contadores activos.

- La función **`schedule()`** deberá contener el nuevo código que permita activar los contadores si se va a ejecutar un proceso que tenga la monitorización activa, así como desactivarlos si se va a dejar de ejecutar un proceso de esas mismas características.

Sin recurrir a mayores formalismos de diseño, podemos convenir que el nuevo código que buscamos debe efectuar lo siguiente:

- El driver `p4pc_dev`, a petición de un proceso en ejecución correspondiente a una aplicación, recibe mediante un `write()` desde el espacio de usuario la solicitud de monitorizar dicho proceso. Entre otras acciones, el código del driver activará los indicadores o flags de la correspondiente estructura `task_struct` que indicarán al SO que el proceso está siendo monitorizado, así como identificará, en el correspondiente elemento de la estructura, qué contadores son los activos.

- Cuando el SO llame a `schedule()` (porque el proceso en estudio ha rebasado su cuota de procesador o porque exista otro proceso de mayor prioridad), una de las primeras acciones de esta función será analizar si el proceso a retirar (referenciable mediante la macro `current`) mantiene contadores activos, como es el caso. Puesto que el proceso va a ser retirado, procede paralizar los contadores activos, para lo que se efectuará lo siguiente:

    - Para cada contador activo, se lee y guarda el contenido del correspondiente CCCR.

    - Se desactiva el bit “active” en el contenido guardado.

  - Este último valor se escribe de nuevo al CCCR, por lo que se detendrá el contador, aunque se mantendrán los restantes bits de configuración inalterados.

- A punto de retornar la función schedule(), y cuando `current` apunte ya al nuevo proceso a ejecutar, deberemos acceder a la `task_struct` recién activada y averiguar si el proceso va a ser monitorizado. De ser así, deberemos:

  - Para cada contador activo, se lee y guarda el contenido del correspondiente CCCR.

  - Se activa el bit “active” en el contenido guardado.

  - Este último valor se escribe de nuevo al CCCR, por lo que se iniciará de nuevo el contador.

Para terminar, unos últimos apuntes:

- A la creación de cada proceso, la función que se ocupa de inicializar los valores de los elementos de la `task_struct` del nuevo proceso es `do_fork()` (`kernel/fork.c`), por lo que parece conveniente encomendar a esta función la inicialización a ceros de los nuevos elementos a añadir a la estructura `task_struct`.

- Por último, si se respetan los esquemas implícitos en nuestra API (que veremos en el capítulo siguiente) y se detienen los *performance counters* con una llamada a la función `stop_counter()` (que detiene de manera ordenada el contador que se le pasa como parámetro), el mecanismo hardware de los contadores en cuestión quedará en reposo. Pero una aplicación puede terminar sin llamar a esta función, bien por estar mal diseñada o bien por recibir una señal o excepción que no pueda manejar, con lo que los contadores proseguirían con su función de forma indefinida. Por ello sería conveniente habilitar en `do_exit()` (`kernel/exit.c`) un último control: si el proceso que se termina tiene contadores activos, se detienen con el mecanismo ya visto.

CAPITULO 4

EL SOFTWARE PARA EL ACCESO A LOS *PEFORMANCE COUNTERS* DEL PENTIUM 4

4.1 Introducción

Al contrario que en el apartado sobre el diseño (en el que se identificaban el acceso básico y el acceso por proceso a los contadores), en este capítulo revisamos el software definitivo que hemos obtenido como producto final del TFC.

Nuestro planteamiento para el driver que se ha implementado ha pasado por el trabajo en tres partes bien diferenciadas, ya anticipadas en los apartados de la memoria correspondientes al diseño. Para resumirlas:

- Por un lado, nos encontramos con las modificaciones a efectuar en el kernel de Linux para lograr las funcionalidades buscadas, modificaciones ya estudiadas y apuntadas en el correspondiente apartado del capítulo anterior.

- Por otro, identificamos el código del driver en sí. Puesto que lo enlazamos al núcleo mediante un módulo de carga dinámica, y tras la decisión inicial de diseño de realizarlo como un dispositivo de caracteres, su arquitectura está prácticamente tasada y predefinida: las operaciones que soportorá su estructura `file_operations` serán las habituales y básicas `read()`, `write()`, `open()` y `release()`, por lo que, fundamentalmente, el grueso del código del driver corresponderá al contenido que tengamos que dar a estas funciones. Existen algunas funciones auxiliares adicionales que son convenientes para evitar la ejecución de un driver completamente dependiente de la arquitectura, como es éste, en otros procesadores o entornos.

- Por último, para que una aplicación de usuario pueda, de una forma sencilla, acceder a los performance counters del Pentium 4 (inicializarlos, leerlos y detenerlos de una forma ordenada y segura), tendremos que abstraer la complejidad de los mecanismos diseñados por Intel para tal fin, y proveer al usuario de una interfaz o API sencilla y de rápido uso.

La clave principal de esta abstracción, que a su vez es el soporte de la ligazón entre la parte del espacio del kernel y la del espacio de usuario que acabamos de ver, es la utilización de 18 dispositivos `/dev/p4pcxx` (desde `/dev/p4pc0` hasta `/dev/p4pc17`), de igual *major* (que se define en el código del driver) y cuyos *minors* se indician mediante el resultado de la operación aritmética [dirección de memoria del contador] módulo [18], operación con la que obtenemos 18 posibles valores (del 0 al 17) con los que identificar, de una parte, el dispositivo `/dev/p4pcxx` correspondiente, mientras que, de otra, podremos indiciar en el código del driver un array de 18 elementos en el que cada uno de éstos represente un contador y su contexto.

Pasamos a continuación a estudiar cada una de estas tres partes.

4.2 Las modificaciones al kernel

Para evitar innecesarios o poco relevantes ficheros adicionales, se ha preferido no añadir un *header* específico que soportara las modificaciones efectuadas al kernel. Este soporte habría cubierto tan sólo el número de contadores (utilizando una definición como, por ejemplo, `MAX_COUNTERS`) o las direcciones de los CCCRs, datos ambos que incluímos directamente en el correspondiente código y que, por sencillo y corto, creemos no pierde legibilidad.

La primera modificación a efectuar es en la estructura `task_struct` (`linux/sched.h`), que define a todo descriptor de proceso. De acuerdo al análisis efectuado, tenemos que incluir en la estructura un elemento que nos indique si el proceso que representa está siendo monitorizado. Un contador del número de contadores activos así nos lo confirmará cuando su valor sea mayor de cero.

Por otra parte, añadimos un flag que indentificará con cada bit activo los contadores en funcionamiento. Este flag se indicia mediante la operación [dirección de memoria del CCCR] módulo [18], que casualmente mapea el primer CCCR en el elemento 0 del array.

El código, tras añadir al final de la estructura los nuevos elementos, queda de la siguiente forma (se remarcan las modificaciones o adiciones, mientras que mantenemos atenuado parte del código original para mejor identificación de la situación de las nuevas sentencias):

```
struct task_struct {
      /*
       * offsets of these are hardcoded elsewhere - touch with care
       */
      volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
      unsigned long flags;    /* per process flags, defined below */
      int sigpending;

...

      spinlock_t alloc_lock;

/* journalling filesystem info */
      void *journal_info;

/* p4pc nuevo codigo */
      int nr_active_counters;     /* Cuantos contadores activos? */
      u32 active_cccrs_flag;     /* Que cccrs estan activos -flag-? */
};
```

Las siguientes modificaciones las haremos en la función `schedule()` (`kernel/sched.c`), función que en la versión del kernel que hemos modificado (2.4.21) incluye el código de `context_switch()` que otras versiones (como la inmediatamente anterior, por ejemplo) individualizan como función aparte. Añadimos código al principio y al final de la función para referenciar al proceso saliente y al entrante, respectivamente, y ejecutar, en consecuencia, las correspondientes acciones: comprobar si el proceso saliente está monitorizado, en cuyo caso detenemos los contadores activos (que tendremos que identificar en un bucle `for`) mediante la inactivación del bit “enable” del CCCR, y comprobar si el proceso entrante también lo está, en cuyo caso activamos de nuevo los correspondientes contadores:

```
asmlinkage void schedule(void)
{
      struct schedule_data * sched_data;
      struct task_struct *prev, *next, *p;
      struct list_head *tmp;
      int this_cpu, c;
      int i; u64 msr_content;  /* p4pc nuevo codigo */


      spin_lock_prefetch(&runqueue_lock);

/* p4pc nuevo codigo */
      if(current->nr_active_counters > 0)
            for(i=0; i<18; i++){
                  if(test_bit(i, &current->active_cccrs_flag)){
                        msr_content = rdmsr64((u32)(0x0360 + i));
                        msr_content = msr_content & 0xfffffffffffefff;
                        wrmsr64((u32)(0x360 + i), msr_content);
                  }
            }

      BUG_ON(!current->active_mm);
need_resched_back:

...

      switch_to(prev, next, prev);
      __schedule_tail(prev);

same_process:
      reacquire_kernel_lock(current);
      if (current->need_resched)
            goto need_resched_back;

/* p4pc nuevo codigo */
      if(current->nr_active_counters > 0)
            for(i=0; i<18; i++){
                  if(test_bit(i, &current->active_cccrs_flag)){
                        msr_content = rdmsr64((u32)(0x0360 + i));
                        msr_content = msr_content | 0x0000000000001000;
                        wrmsr64((u32)(0x360 + i), msr_content);
                  }
            }

      return;
}
```

Debemos realizar algunas modificaciones adicionales. En `do_fork()` (`kernel/fork.c`), incluímos código para inicializar los nuevos elementos de `task_struct`:

```
int do_fork(unsigned long clone_flags, unsigned long stack_start,
          struct pt_regs *regs, unsigned long stack_size)
{
      int retval;
      struct task_struct *p;
      struct completion vfork;
```

```
...

      p->counter = (current->counter + 1) >> 1;
      current->counter >>= 1;
      if (!current->counter)
            current->need_resched = 1;

/* p4pc nuevo codigo */
      p->nr_active_counters = 0;
      p->active_cccrs_flag = 0x00000000;

      /*
       * Ok, add it to the run-queues and make it
       * visible to the rest of the system.
       *
       * Let it rip!
       */
      retval = p->pid;
      p->tgid = retval;
      INIT_LIST_HEAD(&p->thread_group);

...

bad_fork_free:
      free_task_struct(p);
      goto fork_out;
}
```

Asímismo, en `do_exit()` (`kernel/exit.c`), y para cumplir el último de los requisitos identificados anteriormente, debemos añadir el código correspondiente. Puesto que, de una forma habitual, un proceso monitorizado no debería llegar a esta función con los contadores activos, utilizamos la correspondiente directiva del compilador a través de la macro `unlikely()` para optimizar en lo posible el código:

```
NORET_TYPE void do_exit(long code)
{
      int i; u64 msr_content;  /* p4pc nuevo codigo */

      struct task_struct *tsk = current;

      if (in_interrupt())
            panic("Aiee, killing interrupt handler!");
      if (!tsk->pid)
            panic("Attempted to kill the idle task!");
      if (tsk->pid == 1)
            panic("Attempted to kill init!");

/* p4pc nuevo codigo */
      if(unlikely(tsk->nr_active_counters > 0))
            for(i=0; i<18; i++){
                  if(test_bit(i, &tsk->active_cccrs_flag)){
                        msr_content = rdmsr64((u32)(0x0360 + i));
                        msr_content = msr_content & 0xfffffffffffefff;
                        wrmsr64((u32)(0x360 + i), msr_content);
                  }
            }

      tsk->flags |= PF_EXITING;
      del_timer_sync(&tsk->real_timer);
```

```
...

      goto fake_volatile;
}
```

Por último, se han añadido a `asm/msr.h` las dos funciones siguientes, que realizan tareas semejantes a las ya definidas originalmente como macros en este fichero pero que, en lugar de manejar datos de 32 bits, lo hacen con datos de 64 bits, lo que nos simplifica el código para leer y escribir los CCCRs:

```
/* p4pc nuevo codigo, auxiliar */
static __inline u64 rdmsr64(u_int msr)
{
u64 rv;
__asm __volatile("rdmsr" : "=A" (rv) : "c" (msr));
return rv;
}

static __inline void wrmsr64(u_int msr, u64 newval)
{
__asm __volatile("wrmsr" : : "A" (newval), "c" (msr));
}
```

4.3 El device driver

En el software que hemos creado, el código del driver lo encontramos en el fichero `p4pc_dev.c`.

El driver se estructura alrededor de un array `counters_structure[]` de 18 elementos (uno por contador) del tipo `struct counter_config`. Esta estructura contiene los registros que definen si está activo, y cómo está actuando, un determinado contador, y se define de la siguiente forma:

```
struct counter_config{
      __u32 escr_bits;        /* Vector de bits del escr */
      __u32 cccr_bits;        /* Vector de bits del cccr */
      __u16 escr_addr;        /* Direccion del escr */
      __u16 cccr_addr;        /* Direccion del cccr */
      __u16 counter_addr;     /* Direccion del contador */
      __s64 account;          /* Cuenta atras hasta overflow */
      _Bool pebs_on;          /* Precise Event Based Sample? */
      __u64 result;           /* Resultado a devolver a esp. usuario */
      _Bool active;           /* ¿Esta el contador activo? */
      pid_t process;          /* Proceso que inicio el contador */
      struct semaphore sem;   /* Semaforo para posible concurrencia */
};
```

Los registros de esta estructura recogen el contenido de los campos de bits con los que configurar los correspondientes MSRs, las direcciones de éstos, un registro para almacenar la lectura del contador, el pid del proceso propietario y otras informaciones que se explican por sí solas. Los identificadores y comentarios son, creo, suficientemente descriptivos.

Tendremos, por lo tanto, un elemento no nulo en el array `counters_structure[]` por cada contador que una aplicación de usuario active, permitiendo nuestro código que estén activos, simultáneamente, los 18 contadores disponibles en el Pentium 4.

Como ya hemos anticipado, este array se indicia mediante [dirección memoria contador] módulo [18].

Veremos enseguida la actuación para cada una de las posibles llamadas a las funciones definidas en la estructura `file_operations` del dispositivo, pero previamente estudiaremos las funciones de inicio y salida del módulo.

module_init():

El inicio del módulo se ha implementado en la función `p4pc_dev_init()`. Comprobamos, en primer lugar, que estamos utilizando un sistema de un único procesador. Aunque el driver correría en un sistema SMP, se dejaría al usuario el cuidado en la gestión de esta situación, lo que no parece del todo adecuado.

Pasamos a confirmar que corremos bajo un procesador Pentium 4 llamando a la función `pentium4_confirm()`. Seguidamente comprobamos que la versión del kernel es 2.4 o superior.

Si estas comprobaciones no tienen éxito salimos, ya que el driver es completamente dependiente de la arquitectura y modelo del procesador.

Si el entorno es el correcto, registramos el dispositivo mediante `register_chrdev()`, estableciendo su major, su nombre y la dirección de su estructura `file_operations`.

module_exit():

Esta función, implementada en nuestro driver como `p4pc_dev_exit()`, se limita a descargar de forma ordenada el dispositivo con la correspondiente función `unregister_chrdev()`.

open():

Implementada como `p4pc_dev_open()`, su primera acción es averiguar el *minor* del dispositivo abierto por la aplicación de usuario (a través del *inode* que se le pasa en la llamada `open()` del espacio de usuario), con lo que identificamos directamente el contador a considerar.

Acto seguido, se comprueba si el contador ya está activo comprobando el booleano `active` de su estructura. Si lo está, comprobamos si el proceso propietario es el mismo, en cuyo caso no hacemos nada. Por el contrario, si un proceso diferente intenta abrir este contador, retornamos error.

```
if((counters_structure[o_minor].active == TRUE)
&& (current->pid != counters_structure[o_minor].process)){
      printkf("P4PC_ERROR_K:  Contador en uso por otro proceso\n");
      return -EBUSY;    /* ...salimos indicando dispositivo ocupado */
}
```

Si el dispositivo contador no está activo, no procedemos a activarlo en `open()`, sino que esperamos a `write()` para hacerlo. Con esto evitamos, como veremos en la función `read()`, que algún proceso mal diseñado pueda leer de un contador que todavía no ha sido inicializado propiamente con `write()`. No obstante, ya tomamos nota del pid del proceso propietario:

```
if(counters_structure[o_minor].active == FALSE){
            counters_structure[o_minor].process = current->pid;
}
```

Para terminar, y de acuerdo a los estándares, devolvemos 0 si todo ha ido bien.

release():

Llamada en última instancia por un `close()` en código de usuario, esta función la implementamos en el driver como `p4pc_dev_release()`.

Al igual que en el caso anterior, inicialmente identificamos el *minor*. Si el contador correspondiente a este minor está activo pero pertenece a otro proceso, se rechaza la acción devolviendo error.

```
if((counters_structure[c_minor].active == TRUE)
&& (current->pid != counters_structure[c_minor].process)){
      printkf("P4PC_ERROR_K:  Contador en uso por otro proceso\n");
      return -EBUSY;    /* ...salimos indicando dispositivo ocupado */
}
```

Si el cierre del dispositivo está originado por una llamada a `stop_counter()`, función perteneciente a la API de usuario, tendremos desactivado el bit "enable" del correspondiente CCCR. Lo mismo pasará si algún código mal diseñado pretende cerrar un dispositivo no activo. En cualquier caso, desactivamos el contador poniendo a FALSE el valor de su atributo active:

```
if(!(counters_structure[c_minor].cccr_bits & 0x00001000)){
            counters_structure[c_minor].active = FALSE;
}
```

Antes de salir de la función, reducimos el número de contadores activos y desactivamos el correspondiente bit del flag que recoge los CCCRs activos:

```
/* Codigo para discriminacion por proceso:  actuamos en su task_struct
*/
      current->nr_active_counters--;
      clear_bit((counters_structure[c_minor].cccr_addr% MAX_COUNTERS),
            &current->active_cccrs_flag);
```

Y también devolvemos 0 si todo ha ido bien.

read():

Implementada como `p4pc_dev_read()`.

Tras comprobar que el tamaño de los parámetros pasados por la aplicación de usuario es correcto, averiguamos el *minor* a través del puntero al archivo que se pasa con la llamada de usuario `read()`.

Identificado el contador con el *minor*, comprobamos si está activo. En caso contrario, devolvemos error.

Si el contador está activo, y con independencia del propietario del dispositivo (es decir, permitimos que otro proceso lea del contador), procedemos a la lectura del mismo almacenando el resultado en el atributo `result` de la estructura que define al contador o dispositivo:

```
rdmsr(counters_structure[r_minor].counter_addr, low, high);
counters_structure[r_minor].result = (__u64)high;
counters_structure[r_minor].result =
      counters_structure[r_minor].result) << 32;
counters_structure[r_minor].result =
      counters_structure[r_minor].result | (__u64)low;
```

Una vez obtenido el resultado, lo devolvemos al usuario con la correspondiente llamada a `copy_to_user()`:

```
if(copy_to_user(buffer, (char *)&(counters_structure[r_minor].result),
sizeof(counters_structure[r_minor].result)))
      return -EFAULT;
```

Para cumplir el estándar, según el cual la llamada a `read()` por parte de una aplicación de usuario debe devolver el número de bytes leídos, devolvemos, si todo ha sido correcto, el mismo parámetro de cuenta de bytes que se nos ha pasado con la llamada desde el espacio de usuario.

write():

Le corresponde, en nuestro driver, la función `p4pc_dev_write()`, y es la función más amplia y densa del código del dispositivo. No en vano le corresponde la tarea de gestionar toda la información que le pasa la aplicación de usuario, comprobar su corrección y escribir las correspondientes configuraciones de bits en los MSRs destinatarios. Veamos cómo actúa:

Tras controlar que el tamaño de los parámetros que se le pasan es correcto y averiguar el *minor* de la misma forma que la función anterior, retornamos error si el dispositivo ya está en uso por otro proceso

```
if((counters_structure[w_minor].active == TRUE)
&& (current->pid != counters_structure[w_minor].process)){
      printkf("P4PC_ERROR_K:  Contador en uso por otro proceso\n");
      return -EBUSY;    /* ...salimos indicando dispositivo ocupado */
}
```

El siguiente paso es copiar el búffer del espacio de usuario (mensaje que nos envía la función `write()` desde la aplicación) al espacio del kernel, lo que hacemos llamando a `copy_from_user()`.

Conociendo en qué posición del mensaje recién adquirido por el driver se encuentra la dirección de memoria del contador, obtendremos su situación en el array de contadores `counters_structure[]` con la ya comentada operación [dirección memoria contador] módulo [18]:

```
ptr_aux = (char *)&counter_addr_aux;
for(i=0; i<sizeof(counter_addr_aux); i++)
      *(ptr_aux+i) = entry_msg[12+i];
aux = counter_addr_aux % MAX_COUNTERS;
```

Obviamente, y tal y como hemos definido las relaciones entre contadores y dispositivos, en circunstancias normales este índice calculado tiene que coincidir con el minor que hemos obtenido del *file* que se nos pasa como parámetro. Aprovechamos esta circunstancia para establecer un control adicional y confirmar que

la aplicación de usuario está haciendo un uso correcto de las estricta relación que existe entre los dispositivos `/dev/p4pcxx` y los contadores:

```
if(w_minor != (int)aux){
      printkf("P4PC_ERROR_K:  Se ha pasado al driver una pareja");
      printkf("    dispositivo/contador erronea.");
      return -EPERM;
}
```

A partir de este momento, y utilizando como índice del array `counters_structure[]` el calculado previamente, rellenamos completamente la estructura `counter_config` correspondiente que define al contador y su entorno. Para ello, vamos leyendo y traduciendo debidamente la información que hemos copiado del espacio de usuario. Para, como ejemplo, el campo de bits del ESCR haremos:

```
/* Vector de bits escr */
ptr_aux = (char *)&counters_structure[aux].escr_bits;
for(i=0; i<sizeof(counters_structure[aux].escr_bits); i++)
      *(ptr_aux+i) = entry_msg[i];
```

Tras dar valores de esta forma a los campos de bits, a las direcciones de los MSRs, a la magnitud de cuenta atrás (en su caso, puesto que no se utiliza en este TFC), al booleano `pebs_on` y activar el contador colocando a TRUE el atributo `active` (se permite que el proceso propietario sobreescriba una configuración ya activa simplemente continuando el proceso), pasamos a actualizar los correspondientes registros de la `task_struct` del proceso:

```
/* Codigo para discriminacion por proceso:  actuamos en su task_struct
*/
      set_bit((counters_structure[aux].cccr_addr % MAX_COUNTERS),
            &current->active_cccrs_flag);
      current->nr_active_counters++;
```

Inmediatamente después procedemos a la escritura de los tres MSRs involucrados:

```
wrmsr64((__u32)counters_structure[aux].escr_addr,
      (__u64)counters_structure[aux].escr_bits);
wrmsr64((__u32)counters_structure[aux].counter_addr,
      (__u64)counters_structure[aux].account);
wrmsr64((__u32)counters_structure[aux].cccr_addr,
      (__u64)counters_structure[aux].cccr_bits);
```

Vemos que, en primer lugar, escribimos el campo de bits del ESCR. Seguidamente, y caso de existir, escribimos la cuenta atrás (magnitud negativa) en el contador que estamos activando. En nuestro trabajo no utilizaremos esta característica, y utilizaremos el contador como acumulador de ocurrencias, configurándolo, por lo tanto, a 0x0. En último lugar, escribimos el campo de bits del CCCR. El bit “enable” de este vector es el que activará o activará, en definitiva, el hardware en estudio.

Por último, de haber ido todo bien, indicamos esta circunstancia a la aplicación de usuario retornándole la misma magnitud que se nos pasaba como parámetro de cuenta de bytes a escribir.

Otras funciones:

El driver se completa con algunas funciones adicionales:

- `printkf()`: bajo el sistema X, `printk()` no vuelca a la consola del proceso, sino al correspondiente log. Con `printkf()` logramos transferir las salidas también a la consola activa, lo que mejora la interacción con el driver (sobre todo en su desarrollo). En la aplicación final cabría perfectamente su desactivación.

- `pentium4_confirm()`: utilizando la instrucción de ensamblador CPUID averiguamos si el procesador en el que se pretende correr el driver es un Pentium 4. En caso contrario, el driver no se carga.

- `kernel_24_confirm()`: en las primeras etapas de diseño se decidió implementar el driver para las series del kernel 2.4 y posteriores, por lo que el driver contrasta la versión del núcleo sobre la que va a correr. Si es anterior a la mencionada, no se carga.

4.4 En el espacio de usuario

Puesto que, como hemos anticipado, nuestro objetivo para el usuario es abstraer una realidad compleja como es el acceso a los performance counters del Pentium 4, y facilitarle al máximo su tarea, deberemos instrumentalizar una serie de capas que nos permitan pasar de la información y funciones de bajo nivel necesarias para configurar los correspondientes MSRs a una API cómoda y rápida de utilizar.

4.4.1 El nivel más bajo: p4pc_event.h, p4pc_event_catalog.h y p4pc_low_level_lib.c:

Nuestro punto de partida lo encontramos en el fichero `p4pc_event.h`, fichero de cabecera donde definimos o declaramos las estructuras de datos o funciones que necesitaremos para nuestros propósitos.

En un primer núcleo de definiciones comunes, caben destacarse las correspondientes a los errores que pueden darse en la ejecución de las funciones de nuestra API:

```
#define P4PC_EMKNOD  1  /* Error en mknod() (install_p4pc_devices())*/
#define P4PC_ENOMSG  2  /* No existe mensaje para write().
                           ¿Se ha seleccionado evento para el contador? */
#define P4PC_EOPEN   3  /* Error en open(). ¿Esta cargado el
                           modulo p4pc_dev.o? */
#define P4PC_EWRITE  4  /* Error en write() */
#define P4PC_ENODESCR 5 /* No existe descriptor.  ¿Se ha iniciado
                           el contador? */
#define P4PC_EREAD   6  /* Error en read() */
```

También es destacable la declaración del array

```
__u8 p4pc_write_msg_array[MAX_COUNTERS][P4PC_MAXBUF];  /* Array de
      mensajes */
```

Esta tabla de 18 elementos servirá a un proceso para almacenar hasta 18 mensajes de una longitud de 20 bytes, uno por posible contador de destino. Cada elemento recoge la información necesaria para la configuración de un contador, información que se pasaría mediante un `write()` al driver.

Seguidamente son reseñables las definiciones:

```
/* Vinculacion entre contadores y dispositivos, ordenados segun
(counter_addr % MAX_COUNTERS), es decir, (direccion del contador
modulo 18) */
#define BPU_COUNTER0         12
#define BPU_COUNTER1         13
#define BPU_COUNTER2         14
#define BPU_COUNTER3         15
#define MS_COUNTER0          16
#define MS_COUNTER1          17
#define MS_COUNTER2          0
#define MS_COUNTER3          1
#define FLAME_COUNTER0       2
#define FLAME_COUNTER1       3
#define FLAME_COUNTER2       4
#define FLAME_COUNTER3       5
```

```
#define IQ_COUNTER0           6
#define IQ_COUNTER1           7
#define IQ_COUNTER2           8
#define IQ_COUNTER3           9
#define IQ_COUNTER4           10
#define IQ_COUNTER5           11
/* Array auxiliar con los nombres de los dispositivos /dev */
static const char *p4pc_dev_names[MAX_COUNTERS] = {"/dev/p4pc0",
"/dev/p4pc1", "/dev/p4pc2", "/dev/p4pc3", "/dev/p4pc4", "/dev/p4pc5",
"/dev/p4pc6", "/dev/p4pc7", "/dev/p4pc8", "/dev/p4pc9", "/dev/p4pc10",
"/dev/p4pc11",     "/dev/p4pc12",     "/dev/p4pc13",     "/dev/p4pc14",
"/dev/p4pc15", "/dev/p4pc16", "/dev/p4pc17"};
```

Aquí hemos vinculado con cada contador el resultado de la operación [dirección memoria contador] módulo [18] que hemos venido viendo, resultado que convertiremos en el minor del correspondiente dispositivo. Por ejemplo: nuestro objetivo de analizar los fallos de lectura en la caché L2 sólo puede implementarse en los contadores BPU_COUNTER0 o BPU_COUNTER1. Como veremos más adelante, para el estudio de este evento hemos elegido el primero de ellos. Indiciando el `array p4pc_dev_names[]` con el valor asignado a BPU_COUNTER0, el dispositivo que se le asigne será el `/dev/p4pc12`, al que, en consecuencia, habrá de asignársele un *minor* de 12.

Tras nuevas definiciones de las direcciones de los MSRs y otras informaciones convenientes, pasamos a definir las estructuras de datos necesarias para nuestros fines.

En su definición intentaremos seguir el formato y estructura que utiliza Intel en su documentación sobre los performance counters del Pentium 4: "*IA-32 Intel Architecture Software Developers Manual, vol. 3: System Programming Guide*", "*IA-32 Intel Architecture Optimization*" y "*Events on Intel Pentium 4 Processors*".

Así, creamos una primera estructura que recoge la información necesaria para identificar el evento a estudiar:

```
struct p4pc_event_id{
      __u8 event_group; /* 6 bits - Clase de eventos */
      __u16 event_mask; /* 2 bytes - Mascara en una clase */
      __u32 escr;       /* Direccion ESCR */
      __u32 counter;    /* Direccion Counter */
      __u32 cccr;       /* Direccion CCCR */
      __u8 cccr_escr;   /* 3 bits - Ordinal 0-7 de ESCR en su grupo */
};
```

La totalidad de la información imprescindible para tal evento queda recogida en las correspondientes tablas de la documentación de Intel, por lo que la implementación de un determinado evento pasaría por las siguientes etapas:

1. Localización del correspondiente grupo de eventos al que pertenece (normalmente, anexo A de la "*IA-32 Intel Architecture Software Developers Manual, vol. 3: System Programming Guide*").

2. Estudio de la máscara que define el evento de entre los diferentes bits posiblemente definidos para cada grupo de eventos.

3. Estudio del resto de información necesaria (direcciones de los registros e identificación del ESCR en su propio grupo).

4. Implementación de la correspondiente estructura integrando la información anteriormente descrita.

Además de esta estructura de identificación, un contador necesita un contexto para saber cómo realizar la monitorización del evento. Modelamos este contexto en cuatro estructuras adicionales:

- `p4pc_usr_ost_t`: recoge información relativa al nivel de privilegios e hilos a estudiar.

- `p4pc_tag`: información sobre etiquetado de µops.

- `p4pc_filter`: información sobre si se realiza, y de qué forma, el filtrado de los resultados de un evento.

- `p4pc_ovf`: información sobre el tratamiento de los desbordamientos de los contadores.

Por último, recogemos en una **estructura resumen**, que denominamos **`event_count_config`**, las cinco estructuras anteriores, más un booleano que nos indicaría si procede o no el muestreo preciso de eventos:

```
struct p4pc_event_count_config{
      struct p4pc_event_id *event_id;
      struct p4pc_usr_os_t *usr_os_t;
      struct p4pc_tag *tag;
      struct p4pc_filter *filter;
      struct p4pc_ovf *ovf;
      _Bool pebs_on;    /* 1 bit - Activa precise event based sampl. */
};
```

El fichero de cabecera `p4pc_event.h` termina con la declaración de todas las funciones, tanto de bajo como de alto nivel, que se utilizarán en la correspondiente API, y que en su momento veremos.

Directamente vinculado a este nivel de información, se ha creído conveniente resumir en un fichero la implementación de las posibles instancias de los eventos a estudiar, incluyendo en nuestro caso los que son objeto de este TFC. De esta forma, en `p4pc_event_catalog.h` encontramos:

- Configuraciones básicas de las estructuras "de contexto" que acabamos de ver: `p4pc_usr_os_t`, `p4pc_tag`, `p4pc_filter` y `p4pc_ovf`. Estas configuraciones básicas nos permiten el recuento de eventos tanto de hilos de usuario como de SO, en ausencia de *Hyper-Threading*, sin etiquetado de µops, sin filtrado y sin tratamiento de desbordamientos.

- Identificaciones de los eventos "fallos de lectura en caché L2" y "accesos de lectura a caché L2", mediante instancias de sendas estructuras `p4pc_event_id`. Para ello, hemos tenido que definir previamente los

correspondientes flags del grupo de eventos BSQ_cache_reference, grupo al que pertenecen los eventos en estudio.

- Ejemplo adicional de definición de otro grupo de eventos, retired_mispred_branch_type, que no utilizaremos.

Para terminar con este primer nivel de abstracción, debemos habilitar un mecanismo que convierta la información que acabamos de presentar al formato de bajo nivel que será utilizado en los registros MSRs. Esta conversión la efectuamos mediante la función `p4pc_ecc_to_wm()`, que traduce la información contenida en una estructura `p4pc_event_count_config` en el mensaje de longitud 20 bytes a transmitir por `write()` al driver. La función se define en el fichero `p4pc_low_level_lib.c`.

En este último fichero de código se implementan, también, las funciones `deactivate_counter()` y `activate_counter()`, que desactivan y activan, respectivamente, el bit "enable" del campo de bits correspondiente al CCCR de un determinado mensaje. En particular, estas funciones se utilizan en la función de la API `stop_counter()`.

4.4.2 El siguiente nivel de abstracción: p4pc_api.c:

Las funciones que componen la API de nuestra aplicación las encontramos en el fichero `p4pc_api.c`. Vamos a revisarlas individualmente.

Con **`install_p4pc_devices()`** instalamos en `/dev/p4pcxx` los 18 dispositivos. La gestión de errores contempla que ya existieran previamente estos dispositivos, en cuyo caso continuamos sin incidencias. Retorna 0 si la función se ejecuta correctamente y `-P4PC_EMKNOD` si `mknod()` devuelve otro error que no sea el ya comentado:

```
int install_p4pc_devices(void)
{
      int i;
      for(i=0; i< MAX_COUNTERS; i++){
            if((mknod(p4pc_dev_names[i], S_IFCHR | 0664,
MKDEV(P4PC_MAJOR, i)) < 0) && (errno != EEXIST)){
                  return -P4PC_EMKNOD;
            }
      }
      return 0;
}
```

Con la función **`select_event()`**, a la que se le pasa como parámetro un puntero a una estructura `p4pc_event_count_config` (resumen de una configuración de un evento), extraemos el contador a usar para el recuento del evento y, utilizando la función `p4pc_ecc_to_wm()`, obtenemos el mensaje a enviar al driver:

```
void select_event(struct p4pc_event_count_config *e_config)
{
      int counter;
      counter = e_config->event_id->counter % MAX_COUNTERS;
      p4pc_ecc_to_wm(e_config, p4pc_write_msg_array[counter]);
```

```
}
```

`init_counter()`, pasándole como parámetro el contador que queremos activar, efectúa una comprobación de que existe mensaje con la configuración del correspondiente contador a transferir al driver. Si existe, abre el dispositivo oportuno mediante `open()`, para inmediatamente después enviar el mensaje al driver con un `write()`. Si todo ha ido bien, devuelve 0. En caso contrario, si no existe mensaje con el que configurar el hardware (porque, probablemente, no se haya seleccionado un evento), o si hay incidencias en `open()` o en `write()`, devuelve los correspondientes valores negativos:

```
int init_counter(int counter)
{
      if(!p4pc_write_msg_array[counter]){
            return -P4PC_ENOMSG;
      }
      if((fd[counter] = open(p4pc_dev_names[counter], O_RDWR)) < 0){
            return -P4PC_EOPEN;
      }
      if(write(fd[counter], p4pc_write_msg_array[counter],
P4PC_MAXBUF) != P4PC_MAXBUF){
            return -P4PC_EWRITE;
      }
      return 0;
}
```

Para leer un contador activo utilizaremos **`get_counter()`**, a la que se le pasa como parámetro el contador que queremos leer y que devuelve el resultado de la lectura como un entero de 64 bits. Esta función comprueba que se haya inicializado el contador. Si así ha sido, envía un `read()` al driver, cuyo resultado se devuelve. Si no existe el descriptor (probablemente porque no se haya iniciado el contador) o existe algún problema en `write()`, retorna los correspondientes códigos de error:

```
__s64 get_counter(int counter)
{
      __s64 result;

      if(!fd[counter]){
            return -P4PC_ENODESCR;
      }
      if(read(fd[counter],   (char   *)&result,   sizeof(result))   !=
sizeof(result)){
            return -P4PC_EREAD;
      }
      return result;
}
```

Por último, disponemos de la función **`stop_counter()`**, a la que se le pasa el contador a detener. Con esta función enviamos un mensaje al driver que contiene a 0 el bit "enable" del corresponiente CCCR, lo que detiene el proceso de recuento. Para ello, esta función realiza:

1. Comprueba que existe el correspondiente descriptor.

2. Llama a la funcion de bajo nivel `deactivate_counter()`, que se encarga de desactivar el bit "enable" del CCCR del contador pasado como parámetro, desactivación que se realiza en el correspondiente mensaje a pasar al driver.

3. Toma una ultima lectura del contenido del contador a detener, lectura que será el valor devuelto por la función.

4. Escribe el mensaje al driver.

5. Cierra el dispositivo.

6. Anula la entrada en el array de descriptores.

7. Llama a `activate_counter()` para dejar el mensaje en el mismo estado en el que se encontraba.

8. Devuelve la lectura efectuada.

Si no existe el correspondiente descriptor (generalmente, no se habrá inicializado el contador), o si se dan problemas en `read()` o `write()`, retorna los correspondientes errores:

```
__s64 stop_counter(int counter)
{
      __s64 result;

      if(!fd[counter]){
            return -P4PC_ENODESCR;
      }

      deactivate_counter(p4pc_write_msg_array[counter]);

      if(read(fd[counter], (char *)&result, sizeof(result)) !=
sizeof(result)){
            return -P4PC_EREAD;
      }
      if(write(fd[counter], p4pc_write_msg_array[counter],
P4PC_MAXBUF)
      != P4PC_MAXBUF){
            return -P4PC_EWRITE;
      }
      close(fd[counter]);
      fd[counter] = 0;
      activate_counter(p4pc_write_msg_array[counter]);
      return result;
}
```

Estas funciones, que podemos considerar como la interface principal usuario/driver, se complementan con otras funciones "*setter*" que se limitan a ayudar a instanciar las estructuras que ya hemos visto, recibiendo los valores de los correspondientes atributos como parámetros.

Por último, nuestra API en el fichero `p4pc_api.c` se completa con la función **`p4pc_set_BASIC_event_config()`**, función auxiliar que nos resultará útil para

instanciar objetos del tipo `p4pc_event_count_config` con su configuración más básica y que, para recordarla, supone el recuento de todos los hilos (tanto de usuario como de SO), ausencia de *Hyper-Threading*, sin etiquetado de µops, sin filtrado, sin gestión de desbordamientos y sin muestreo preciso. Esta función recibe como parámetros un puntero a la estructura a instanciar y otro a una estructura de identificación de un determinado evento:

```
void p4pc_set_BASIC_event_config(struct p4pc_event_count_config
*e_config, struct p4pc_event_id *e_id)
{
      e_config->event_id = e_id;
      e_config->usr_os_t = &UsrOsNoHT;
      e_config->tag = &NoTag;
      e_config->filter = &NoFilter;
      e_config->ovf = &NoOvf;
      e_config->pebs_on = FALSE;
}
```

4.4.3 Un último nivel: p4pc_TFC_api.h:

El evento relevante para nuestro TFC es el correspondiente a los fallos de lectura en la caché de nivel 2. Intimamente relacionado con éste encontramos el que representa a la totalidad de accesos de lectura a la misma caché.

Como una última capa de abstracción que facilite todavía más la tarea del usuario, definimos las dos macros siguientes en el fichero `p4pc_TFC_api.h`:

```
#define SELECT_2LEVEL_CACHE_READ_MISSES                              \
      do {                                                           \
            p4pc_set_BASIC_event_config(&_2LevelCacheReadMisses,     \
                  &_2LevelCacheReadMisses_id);                       \
            select_event(&_2LevelCacheReadMisses);                   \
      } while(0)

#define SELECT_2LEVEL_CACHE_READ_REFERENCES                          \
      do {                                                           \
            p4pc_set_BASIC_event_config(&_2LevelCacheReadReferences,\
                  &_2LevelCacheReadReferences_id);                   \
            select_event(&_2LevelCacheReadReferences);               \
      } while(0)
```

Estas macros nos permitirán, en la correspondiente aplicación y con tan sólo una línea, seleccionar un evento, un contador y su contexto (básico) completo para su posterior monitorización.

CAPITULO 5

RESUMENES Y CONCLUSIONES

5.1 Introducción

La exposición de la API de alto nivel desarrollada para su utilización directa por una aplicación de usuario puede utilizarse como posible resumen del trabajo efectuado. El software resultante cabe verse, y quizá valorarse, a través de esta exposición.

Ya hemos visto una completa revisión de las funciones que componen esta API en el capítulo 4 (dedicado al estudio del código), apartado 4.4.2. Esta revisión, quizás algo más estructurada, podría servir en su caso como base de un *readme* o pequeño manual a añadir al software.

Pero es difícil encontrar un mejor resumen que una breve aplicación que nos permita tanto poner a prueba la mencionada API como comprobar la corrección de los resultados obtenidos mediante la utilización de algunos algoritmos de resultados previsibles.

Se dedicará el próximo apartado a una aplicación que cumple los requisitos anteriores.

Para terminar, se destinarán las últimas secciones de esta memoria a posibles complementos o mejoras del software que, de una forma u otra, quedaban fuera del alcance de este Trabajo de Fin de Carrera, así como a las pertinentes conclusiones.

5.2 La utilización del producto final: una aplicación de usuario

Transcribimos a continuación, remarcando las sentencias específicas para el acceso a los contadores, el código de una aplicación de prueba que hace uso de la API que hemos visto, tanto de las funciones principales como de alguna de las macros definidas específicamente para este TFC.

Una vez compilado, el programa (que, para no recargar el código, sólo efectúa una gestión de errores básica) activa los contadores oportunos para recoger todas las ocurrencias de fallos de lectura a la caché de segundo nivel y el número total de accesos (fallos y aciertos) de este tipo, obteniendo, a voluntad del usuario, las correspondientes lecturas para la ejecución de un determinado código, que se especifica.

La ejecución del programa debe ser precedida por la compilación del driver (mediante el *makefile* `makefile_dev` que se adjunta) y consiguiente carga del módulo (con el comando `insmod p4pc_dev.o`), así como por la compilación de la aplicación en sí (mediante un simple `make` que actuará sobre el fichero `makefile` que se adjunta con las fuentes).

En el código a analizar explotamos las características de almacenamiento de las matrices en el lenguaje C, que guarda los elementos de una misma fila en celdas contiguas (al contrario que Fortran, que lo hace así con los elementos de una columna) para ver las diferencias en el número de fallos generados en lecturas de la memoria caché al acceder, bien por filas, bien por columnas, a los elementos de una matriz de dimensión DIM x DIM:

```
#include <stdio.h>
#include "p4pc_TFC_api.h"

#define DIM 1024

int main()
{
      int cuenta_reads, cuenta_acces;
      int i, k;
      int matrix[DIM][DIM];

/* Instalamos los dispositivos /dev/p4pcxx */
      install_p4pc_devices();       /* Definida en p4pc_api.c */

/* Seleccionamos eventos a analizar: fallos lectura y accesos en L2 */
      SELECT_2LEVEL_CACHE_READ_MISSES;    /* Def. en p4pc_TFC_api.h */
      SELECT_2LEVEL_CACHE_READ_REFERENCES; /* Def. en p4pc_TFC_api.h*/

/* Iniciamos contadores 0 y 2 del grupo BPU */
      if(init_counter(BPU_COUNTER0) < 0){/* Definida en p4pc_api.c */
            printf("MATRIX_ERROR:  Error inicializando contador 0\n");
            return -1;
      }
      if(init_counter(BPU_COUNTER2) < 0){
            printf("MATRIX_ERROR:  Error inicializando contador 2\n");
            return -1;
      }

/* Empieza codigo 1 a muestrear//////////////////////////////////// */
```

```
      for(i=0; i<DIM; i++)
            for(k=0; k<DIM; k++)
                  matrix[i][k]++;   /* Indices en orden i-k */
/* Termina codigo 1 a muestrear////////////////////////////////// */

/* Paramos -y leemos- contadores */
      /* p4pc_api.c */
      if((cuenta_reads = stop_counter(BPU_COUNTER0)) < 0){
            printf("Error parando contador 0\n");
            return -1;
      }
      if((cuenta_acces = stop_counter(BPU_COUNTER2)) < 0){
            printf("MATRIX_ERROR:  Error parando contador 2\n");
            return -1;
        }
      printf("\nCon la version \"[i][k]\" del codigo se han
producido\n");
      printf("%d fallos de lectura a la cache L2,\n", cuenta_reads);
      printf("y %d accesos de lectura a la misma L2.\n\n",
cuenta_acces);

/* Iniciamos mismo proceso con cambios en el codigo a monitorizar */
      if(init_counter(BPU_COUNTER0) < 0){
            printf("MATRIX_ERROR:  Error inicializando contador 1\n");
            return -1;
      }
      if(init_counter(BPU_COUNTER2) < 0){
        printf("MATRIX_ERROR:  Error inicializando contador 2\n");
        return -1;
        }
/* Empieza codigo 2 a muestrear////////////////////////////////// */
      for(i=0; i<DIM; i++)
            for(k=0; k<DIM; k++)
                  matrix[k][i]++;  /* Cambiado el orden de indices */
/* Termina codigo 2 a muestrear////////////////////////////////// */
      if((cuenta_reads = stop_counter(BPU_COUNTER0)) < 0){
            printf("MATRIX_ERROR:  Error parando contador 0\n");
            return -1;
      }
      if((cuenta_acces = stop_counter(BPU_COUNTER2)) < 0){
        printf("MATRIX_ERROR:  Error parando contador 2\n");
        return -1;
        }
      printf("Con la version \"[k][i]\" del codigo se han
producido\n");
      printf("%d fallos de lectura a la cache L2,\n", cuenta_reads);
      printf("y %d accesos de lectura a la misma L2.\n\n",
cuenta_acces);
}
```

Son de apreciar las pocas líneas de código que una aplicación debe utilizar para tener acceso prácticamente completo a la configuración y lectura de los performance counters.

A continuación se representa una pantalla obtenida tras la ejecución de este programa:

```
[root@Orion TFC]# ./matrix

Con la version "[i][k]" del codigo se han producido
33557 fallos de lectura a la cache L2,
y 345788 accesos de lectura a la misma L2.

Con la version "[k][i]" del codigo se han producido
1053081 fallos de lectura a la cache L2,
y 2927470 accesos de lectura a la misma L2.

[root@Orion TFC]#
```

Vemos que el cambio de posición de los índices para modificar la forma de acceder a los elementos de la matriz (pasando del acceso por filas al acceso por columnas) supone del orden de 30 veces más fallos de lectura en L2, mientras que el número de accesos totales a esta memoria es de más de 8 veces la cantidad obtenida mediante el acceso por filas debido a los fallos generados en las lecturas de la caché de primer nivel.

Las cifras de fallos que obtenemos parecen guardar correlación directa con el número de elementos de la matriz (1.024 x 1.024 = 1.048.576): en el caso del acceso por filas, obtenemos un error por cada 31,25 elementos, que suponen casi 128 bytes y que corresponden a la longitud de la línea de la caché (en realidad, a lo que Intel denomina *sector*, y que son dos líneas de 64 bytes cada una); en el caso del acceso por columnas obtenemos prácticamente un error en cada acceso, puesto que la línea que se carga de la memoria principal sólo contiene el entero buscado y no el que va a requerirse en la siguiente lectura.

De lo anterior, y visto el ajuste de los resultados a lo previsible, parece desprenderse que los contadores actúan y son leídos de forma correcta.

Acabamos de ver que el código muestreado es sencillo y corto. Una aplicación que hiciera uso de la librería para analizar un código extenso o un fichero ejecutable debería ser diseñada contemplando, al menos, dos mecanismos adicionales:

- El uso de fork() o exec() facilitaría una estructura probablemente más racional del recuento de eventos. Además, si el programa a analizar ya se encuentra compilado en forma de ejecutable, sería la única opción posible.

- El establecimiento de un temporizador que permitiera tomar lecturas periódicas del contador permitiría un análisis posterior más efectivo, aunque éste sería óptimo si se dispusiera de un servicio a la interrupción PMI que registrara las instrucciones causantes del evento en revisión.

5.3 Complementos y mejoras del código

Veremos en este epígrafe temas relacionados con la sincronización del código del kernel, consideraciones sobre máquinas multiprocesador y sobre la posible interfaz de usuario.

5.3.1 Mecanismos de sincronización:

La ejecución del código de kernel de Linux, en la versión 2.6, ha acabado siendo *apropiativa*, lo que quiere decir que en cualquier momento, y siempre que no esté actuando bajo un semáforo o cualquier otro mecanismo que asegure la concurrencia, su código puede ser apartado de la ejecución si existe un proceso de mayor prioridad pendiente de ejecución.

Hasta la versión 2.4 el planificador no actuaba si se estaba corriendo código de kernel. Así, en el software que hemos desarrollado bajo la versión 2.4.21, y siempre que lo ejecutemos en cualquier versión 2.4, no parece necesaria la existencia de sistemas de sincronización puesto que el acceso a los datos globales del driver se realiza siempre de una forma *atómica* por naturaleza: el código privilegiado del kernel no puede ser retirado de ejecución, por lo que el acceso sin problemas de concurrencia o sincronización a las estructuras de datos está garantizado.

Pero si se corre el módulo bajo la versión de kernel 2.6, sí cabe la posibilidad de que el planificador se apropie del procesador para dar paso a un proceso de mayor prioridad. Puesto que nuestra aplicación, dirigida a usuarios especializados, correría en entornos de ejecución probablemente controlada y segura, son difíciles de concebir colusiones o problemas de sincronización entre procesos compitiendo por un mismo recurso, aunque técnicamente no es imposible.

Como solución, se ha dotado a la estructura `counter_config` del driver (que se encuentra en `p4pc_dev.c`) de un atributo `struct semaphore sem`, inicializado como de exclusión mutua en `p4pc_dev_init()`. Aunque nuestro código no lo utiliza, es sencillo habilitar un mecanismo de protección de acceso a estas estructuras en las correspondientes funciones mediante `down_interruptible(&counters_structure[i].sem)`, además de con su contrapartida `up(...)`, que encerrarían el código de acceso a los datos a asegurar.

Se ha valorado la inclusión de este mecanismo en el software, pero su propia naturaleza parecía no hacerlo indispensable y, para no recargar más el código, se decidió incluirlo como posible mejora y referenciarlo en estas notas.

5.3.2 Máquinas SMP:

Un entorno multiprocesador origina idéntica problemática a la comentada en el anterior apartado: posible acceso concurrente a datos globales.

Son de aplicación en este caso semejantes consideraciones a las ya tratadas, aunque ahora ahora existe un problema adicional con la multiplicación de registros y contadores, que nuestro software no trata.

Una posible solución vendría dada por la inclusión de un campo de identificación de la correspondiente CPU en la estructura `counter_config`, y su posterior gestión en las

correspondientes secciones del módulo de manera adecuada. En cualquier caso, se considera este tema fuera de los límites del TFC y nos limitamos a no cargar el módulo en presencia de SMP (ver la función `p4pc_dev_init()` en `p4pc_dev.c`), aún cuando en entornos de ejecución controlada, y con presencia de varios procesadores, el driver funcionaría probablemente sin incidencias.

5.3.3 Interfaz de usuario:

De forma consciente no se ha dotado a la aplicación de una interfaz de usuario definida. Dado el preciso objetivo del TFC no era necesaria, y la interfaz de programación que se provee es más que suficiente para nuestros propósitos y, probablemente, para los de cualquier usuario necesitado de sus servicios.

No obstante, hay que reconocer que la inclusión manual de nuevos eventos es una tarea pesada y que en una aplicación para uso frecuente habría que proveer desde el origen, si no todos los eventos definidos por Intel, sí un importante subconjunto de éstos.

Además, una interfaz en modo de texto tampoco parece apropiada para la complejidad que puede alcanzar una configuración completa de varios contadores.

Las estructuras de datos diseñadas pueden ser utilizadas directamente, con algunos complementos, como soporte de una aplicación gráfica para su configuración y uso. Las relaciones que hemos implementado entre estas estructuras entre sí y con el hardware también podrían servir como base para esta aplicación gráfica, aunque en este caso con algunas modificaciones adicionales.

5.4 Conclusiones

Nuestra aplicación permite y/o se caracteriza por:

- Acceso (configuración, lectura y detención) a los *performance counters* del Pentium 4 para cuantificar los fallos de lectura de la caché L2, así como los accesos (suma de fallos y aciertos) a la misma.

- Este acceso se distingue por discriminar de forma efectiva los procesos a monitorizar, sin pérdida de recursos por muestreos genéricos de todos los procesos del sistema.

- El catálogo de eventos disponibles está abierto y preparado para la inclusión de nuevos eventos soportados por el procesador, de forma sencilla y rápida.

- Los eventos se definen y configuran de acuerdo a la literatura de Intel sobre estos contadores.

- El driver y el código soporte de las aplicaciones de usuario permiten una configuración completa del hardware: aunque no lo hemos utilizado en este trabajo, el software provee un marco apropiado para configurar desbordamientos, filtrado y etiquetado de µops.

- Es ampliable para soportar PEBS y servicio a las interrupciones por *overflow*.

- La API que proporciona es sencilla, y permite obtener las funcionalidades buscadas con muy pocas líneas de código en una aplicación de usuario.

- Aunque básica y mejorable, sobre todo en sus aspectos formales, la gestión de errores identifica y permite tratar los más significativos e importantes.

- Las modificaciones del núcleo suponen una sobrecarga despreciable a añadir al código del kernel.

Todo el proyecto se ha desarrollado según la planificación inicialmente prevista, a excepción de un ligero adelanto en las etapas finales que, en la fase de planificación, aparentaban ser más complejas.

Es opinión del autor que los objetivos planteados, incluídos los adicionales, se han cumplido razonablemente.

GLOSARIO

CCCR – *Counter Configuration Control Register*: registro de control de configuración del contador. Con este MSR se configura cómo queremos hacer el recuento del evento seleccionado con el ESCR.

Contador: registro MSR que recoge la cuenta de las ocurrencias del evento en estudio.

CPL – *Current Privilege Level*: nivel actual de privilegios bajo el que corre el procesador. Normalmente, sistema operativo (OS) o usuario (USR).

DS – *Debug Store Save Area*: área de almacenaje de depuración. Búffer en memoria donde se almacena la información que provee la PMI bajo PEBS.

EBS – *Event Based Sampling*: muestreo basado en eventos, que da base al perfil de rendimiento basado en eventos.

ESCR – *Event Selection Control Register*: registro de control de selección de eventos. Es un MSR que permite a la aplicación seleccionar un (o varios) evento a analizar.

Evento: Incidencia relevante para el análisis de un código acaecida en su ejecución, y vinculada al desenvolvimiento del procesador y subsistemas relacionados.

Grupo de eventos: Intel agrupa los diferentes eventos individuales en conjuntos de eventos relacionados (por ejemplo, el grupo de eventos sobre cache contiene los eventos sobre fallos o aciertos de diferentes tipos en las diferentes cachés). Como resultado, a la hora de programar los controladores de los registros de los eventos habrá que indicar el grupo al que pertenece el evento y, dentro de éste, los flags necesarios de los diferentes eventos del grupo a monitorizar.

Métrica: es un cálculo de rendimiento en el que intervienen los resultados de dos o más eventos. Estos últimos son recogidos de sus contadores, mientras que la métrica es calculada.

MSR – *Model Specific Register*: registros del procesador de modelo específico. En general, accesibles sólo por el sistema operativo. Los contadores y registros complementarios son todos MSRs.

PEBS – *Precise Event Based Sampling*: muestreo preciso basado en eventos

Performance counter: Contador de rendimiento (o comportamiento), implementado como unidad de hardware en los actuales procesadores

PMI – *Performance Monitoring Interrupt*: interrupción que puede generarse al desbordarse un contador.

BIBLIOGRAFIA

Algunos documentos (electrónicos o en formatos más tradicionales) han resultado esenciales para este trabajo; otros, de mayor o menor trascendencia, han resultado interesantes o útiles. Se especifican ambas categorías a continuación.

Documentos esenciales

Sobre los *performance counters* del Pentium 4:

Intel Corporation. *IA-32 Intel® Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2004. [*ftp://download.intel.com/design/Pentium4/manuals/25366814.pdf*, octubre 2004].

Intel Corporation. *Events on Intel® Pentium® 4 Processors, Including Hyper-Threading Technology-Enabled Processors*, 2003. [*http://www.intel.com/cd/ids/developer/asmo-na/eng/microprocessors/ia32/pentium4/hyperthreading/51527.htm*, octubre 2004].

Sobre programación en C, del kernel de Linux y de módulos:

Kernighan, B. y **Ritchie, D**. *El lenguaje de programación C, 2ª Edición*. México, Ed. Pearson Educación, 1991.

Love, R. *Linux Kernel Development*. EE.UU, Ed. Sams Publishing, 2004.

Rubini, A. y **Corbet, J**. *Linux Device Drivers, 2nd Edition*. Ed. O'Reilly, 2001. [Accesible mediante *http://www.xml.com/ldd/chapter/book/* , octubre 2004].

Otras fuentes de interés

GCC Developers team. *GCC 3.4.3 Manual*, 2004. [*http://gcc.gnu.org/onlinedocs/gcc-3.4.3/gcc/*, octubre 2004].

Intel Corporation. *IA-32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture*, 2004. [*ftp://download.intel.com/design/Pentium4/manuals/25366514.pdf*, octubre 2004].

Intel Corporation. *IA-32 Intel® Architecture Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z*, 2004. [*ftp://download.intel.com/design/Pentium4/manuals/25366714.pdf*, octubre 2004].

Intel Corporation. *Intel(R) Processor Identification and the CPUID Instruction,* 2003. [*ftp://download.intel.com/design/Xeon/applnots/24161827.pdf*, octubre 2004].

Hennessy, J. y **Patterson, D**. *Arquitectura de Computadores. Un enfoque cuantitativo.* España, Ed. McGraw-Hill, 1993.

Salzman, P. y **Pomerantz, O**. *The Linux Kernel Module Programming Guide, ver. 2.4.0*, 2003. [*http://jamesthornton.com/linux/lkmpg/*, octubre 2004].

Sanchez, F. y **Arango, R**. *El kernel 2.4 de Linux*. España, Ed. Prentice Hall, 2003.

Sprunt, B. *The Performance Monitoring Features of the Pentium 4 Processor*, 2002. [*http://www.eg.bucknell.edu/~bsprunt/emon/brink_abyss/doc/pentium4_emon.pdf*, octubre 2004].

Wall, K. *Programación en Linux con Ejemplos*. Argentina, Ed. Prentice Hall – Pearson Eduación, 2000.

Las páginas web de los programas que se citan en la memoria son:

Abyss: http://www.eg.bucknell.edu/~bsprunt/emon/brink_abyss/brink_abyss.shtm

OProfile: http://oprofile.sourceforge.net/links/

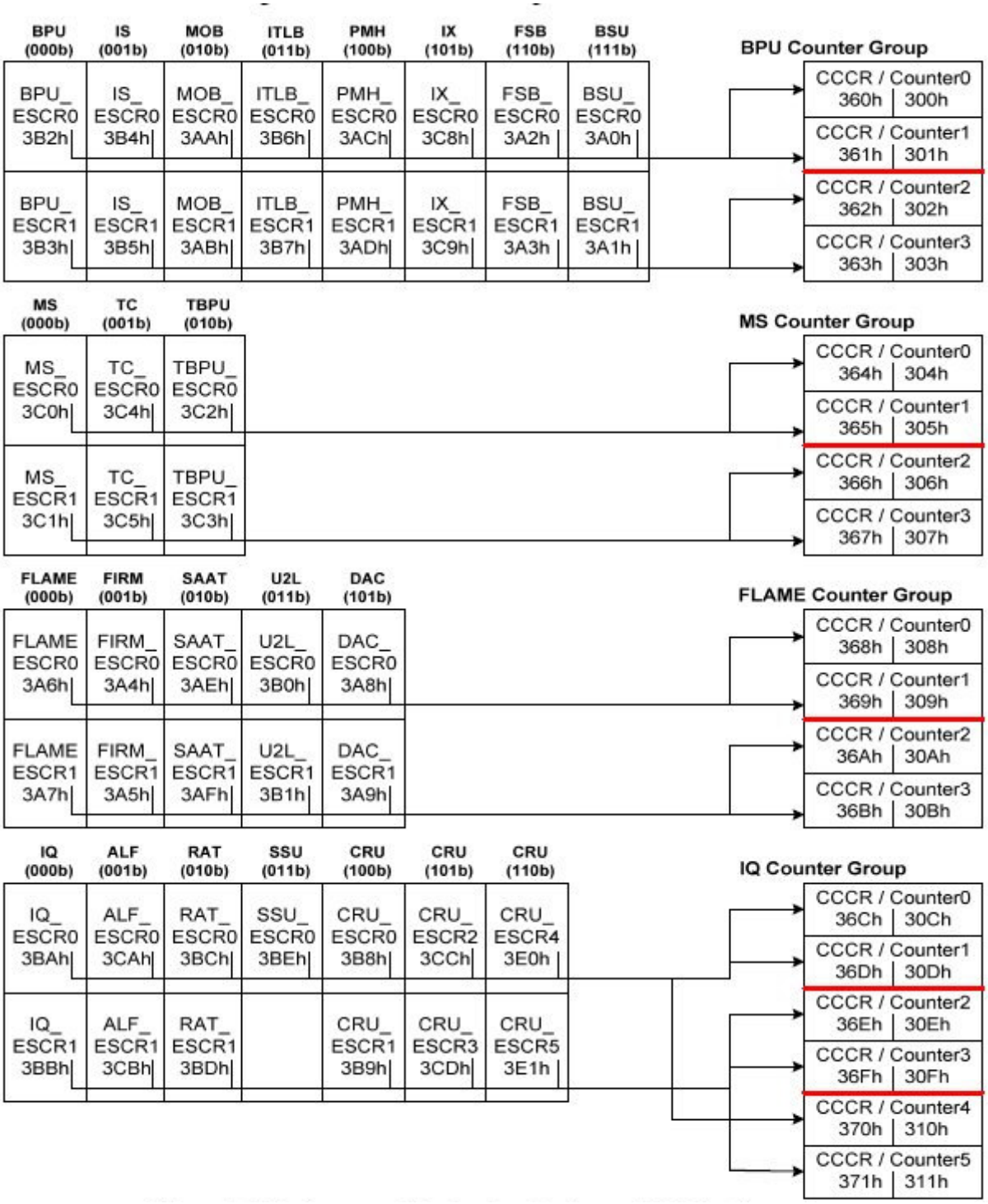
PAPI: http://icl.cs.utk.edu/projects/papi

PCL: http://www.fz-juelich.de/zam/PCL/

VTune: http://developer.intel.com/software/products/vtune/

ANEXO A

TABLA DE CONFIGURACION DE MSRs PARA MONITORIZACION DE RENDIMIENTO



Extraída de: *"Events on Intel Pentium 4 Processors, Including Hyper-Threading Technology-Enabled Processors*".

ANEXO B

PARAMETRIZACION DEL EVENTO BSQ_cache_reference

Event Name	Event Parameters	Parameter Value	Description
BSQ_cache _reference			This event counts cache references (2nd level cache or 3rd level cache) as seen by the bus unit. Specify one or more mask bit to select an access according to the access type (read type includes both load and RFO, write type includes writebacks and evictions) and the access result (hit, misses).
	ESCR restrictions	BSU_CR_ESCR0 BSU_CR_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	0CH	ESCR[31:25]
	ESCR Event Mask	Bit 0: RD_2ndL_HITS 1: RD_2ndL_HITE 2: RD_2ndL_HITM 3: RD_3rdL_HITS 4: RD_3rdL_HITE 5: RD_3rdL_HITM 8: RD_2ndL_MISS 9: RD_3rdL_MISS 10: WR_2ndL_MISS	ESCR[24:9] Read 2nd level cache hit Shared (includes load and RFO). Read 2nd level cache hit Exclusive (includes load and RFO). Read 2nd level cache hit Modified (includes load and RFO). Read 3rd level cache hit Shared (includes load and RFO). Read 3rd level cache hit Exclusive (includes load and RFO). Read 3rd level cache hit Modified (includes load and RFO). Read 2nd level cache miss (includes load and RFO). Read 3rd level cache miss (includes load and RFO). A Writeback lookup from DAC misses the 2nd level cache (unlikely to happen).
	CCCR Select	07H	CCCR[15:13]
	Event Specific Notes		1: The implementation of this event in current Pentium 4 and Xeon processors treats either a load operation or a request for ownership (RFO) request as a "read" type operation. 2: Currently this event causes both over and undercounting by as much as a factor of two due to an erratum. 3: It is possible for a transaction that is started as a prefetch to change the transaction's internal status, making it no longer a prefetch. or change the access result status (hit, miss) as seen by this event.

Extraída de: "*IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*", Anexo 1.