PROYECTO SISTEMAS INFORMÁTICOS

CURSO 2011-2012

INTERFAZ DE USO DE CONTADORES HARDWARE MULTIPLATAFORMA

GUILLERMO MARTÍNEZ FERNÁNDEZ

SERGIO SÁNCHEZ GORDO

SOFÍA DRONDA MERINO

DIRIGIDO POR:

PROF. CARLOS GARCÍA SÁNCHEZ

PROF. JUAN CARLOS SÁEZ ALCAIDE

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

Resumen

La finalidad del proyecto “Interfaz de uso de Contadores Hardware multiplataforma” consiste en crear una herramienta multiplataforma que consiga monitorizar el rendimiento de un programa haciendo uso de los contadores hardware integrados en el procesador.

La portabilidad a otras arquitecturas así como a otros sistemas operativos o versiones de kernel será lo que marque la diferencia con el resto de herramientas del mismo propósito.

Se ha diseñado una herramienta modular para facilitar dicha portabilidad a otras arquitecturas con distintos contadores hardware. Además, las dependencias en el sistema operativo han sido eliminadas, buscando la abstracción en módulos que son cargados según la plataforma objetivo.

Abstract

The aim of this project consists on develop a multi-platform tool that allows performance monitoring by means of the use of hardware counters available in modern processors.

The portability to other architectures, as well as other operative systems and kernel versions, is the main difference behind other similar solutions.

A modular tool has been designed to facilitate its portability to other systems with different hardware counters set. Moreover, the operative system dependencies have been is isolated by kernel modules using which avoid any target platform dependencies.

LISTA DE PALABRAS

AGRADECIMIENTOS

Queremos dar nuestro más sincero agradecimiento a nuestros directores de proyecto, Carlos García y Juan Carlos Sáez, por invertir tanto tiempo en nosotros y por su paciencia.

También nos gustaría dar las gracias a nuestros familiares por el apoyo recibido, no sólo en estos años universitarios, sino durante toda nuestra vida.

Y por último y no menos importante, a nuestros amigos. Por esos momentos de distracción y sus ánimos incondicionales.

Sin ninguno de ellos esto no hubiera sido posible.

A todos ellos, muchas gracias.

ÍNDICE

CAPÍTULO 1

INTRODUCCIÓN

Este proyecto de fin de carrera aborda el uso de contadores hardware para evaluar el rendimiento de una aplicación.

La motivación principal es conseguir que cualquier sistema operativo, arquitectura o versión de kernel puedan hacer uso de los contadores tanto a nivel de usuario como de sistema.

En este capítulo hablaremos de los contadores hardware de rendimiento . Explicaremos algunas herramientas de monitorización ya existentes. Expondremos los principales motivos que nos han llevado al desarrollo de este proyecto. Por último resumiremos el contenido de la memoria del proyecto.

1.1 CONTADORES HW DE RENDIMIENTO

Los contadores fueron introducidos en los procesadores para la monitorización del rendimiento de la CPU. Son contadores específicos que ocupan un área despreciable dentro del chip. Estos contadores permiten seleccionar los parámetros de rendimiento del procesador para ser medidos y monitorizados. La información obtenida puede ser usada para ajustar el sistema y observar el rendimiento.

1.2 HERRAMIENTAS DE MONITORIZACIÓN

En la actualidad, ya existen varias herramientas que realizan la tarea de monitorizar los contadores y obtener sus medidas.

El gran inconveniente de las herramientas actuales es la dependencia del SO, de las arquitecturas, etc. Por otro lado, encontramos que la mayoría de las herramientas sólo permiten el funcionamiento únicamente en modo usuario, no encontrándose en la actualidad la posibilidad de interacciones a bajo nivel, como por ejemplo con el planificador del sistema operativo.

Entre las herramientas disponibles en la actualidad destacamos *cputrack*[1], sólo disponible para el sistema Solaris; y *pcm-power*[2], para las arquitecturas de Intel. A continuación, analizaremos estas herramientas:

CPUTRACK:

Solaris 8 OE contiene una serie de APIs que están disponibles como librerías compartidas para programar los contadores hardware. Además, existen herramientas útiles como son *cpustat y cputrack* para acceder a estos contadores del microprocesador a través de la línea de comandos.

La utilidad de *cputrack* es informar sobre los contadores de rendimiento de la CPU devolviendo la medición de un evento de un proceso. Los contadores suspenden la cuenta cuando se produce un cambio de contexto; y se restaura cuando el proceso es devuelto a la CPU. Para lanzar la ejecución de esta aplicación no es necesario ser administrador.

Además, esta herramienta contiene gran cantidad de aspectos configurables. La elección del evento a medir, el tiempo de muestreo o el volcado a un fichero son algunos de ellos.

[1]Ver su implementación en http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/cmd/cpc/common/cputrack.c

[2]Información y descarga de la herramienta en http://software.intel.com/en-us/articles/intel-performance-counter-monitor-a-better-way-to-measure-cpu-utilization#intro

Ejemplo de salida de *cputrack*:

```
# cputrack -T 0 -fve -c Cycle_cnt,Instr_cnt sh -c date
time      pid     lwp     event           pic0    pic1
0.008     9526    1       init_lwp        0       0
0.021     9526    1       fork                            # 9527
0.023     9527    1       init_lwp        0       0
0.025     9527    1       fini_lwp        93760   60136
0.025     9527    1       exec            93760   60136
0.000     9527    1       exec                            # 'date'
0.033     9527    1       init_lwp        0       0
0.041     9527    1       fini_lwp        787164  435118
0.041     9527    1       exit            787164  435118
0.044     9526    1       fini_lwp        1085444 542027
0.044     9526    1       exit            1085444 542027
```

PCM-POWER:

Los procesadores Intel ya proporcionan la capacidad de monitorizar los eventos del rendimiento dentro del procesador. Para obtener con más precisión una imagen de la utilización de la CPU se confía en los datos dinámicos que se obtienen llamando a las Unidades de Monitorización del Rendimiento (PMU) implementado en los procesadores. Estas características avanzadas están disponibles en los siguientes procesadores Intel® Xeon® 5500, 5600, 7500, E5, E7 y Core i7.

Se han puesto en marcha un conjunto de rutinas con una interfaz de alto nivel que se puede llamar desde el usuario proporcionando diversos parámetros de rendimiento de la CPU en tiempo real. Una característica interesante es su uso para *core* y para las unidades de gestión *uncore* del procesador Intel. El *uncore* es la parte del procesador que contiene el controlador de memoria integrado. En general, las siguientes mediciones son compatibles:

- **Core:** Instrucciones retiradas, transcurridos nticks de reloj del núcleo, accesos y fallos de calle L2 y L3.
- **Uncore:** Lee y escribe bytes desde el controlador de memoria y el tráfico de datos transmitidos por el procesador.

Gracias a la capa de abstracción que proporciona la biblioteca, es muy fácil de supervisar las métricas del procesador dentro de su aplicación. Antes de su uso, los contadores de rendimiento deben ser inicializados. Luego, el estado del contador puede ser capturado antes y después de la sección de código de interés. Diferentes rutinas capturan los contadores de los cores, tomas de corriente o el sistema completo, y almacenan su estado en las estructuras de datos correspondientes. Otras rutinas adicionales ofrecen la posibilidad de calcular la métrica basada en estos estados.

La utilidad *pcm-power* esta incorporada en el procesador Intel® Xeon® serie E5, PCM versión 2.0.

El fragmento de código siguiente muestra un ejemplo de su uso:

```
PCM * m = PCM::getInstance();
// program counters, and on a failure just exit
if (m->program() != PCM::Success) return;
SystemCounterState before_sstate = getSystemCounterState();
      [run your code here]
SystemCounterState after_sstate = getSystemCounterState();
cout << "Instructions per clock:" << getIPC(before_sstate,after_sstate)
     << "L3 cache hit ratio:" <<
getL3CacheHitRatio(before_sstate,after_sstate)
     << "Bytes read:"<< getBytesReadFromMC(before_sstate,after_sstate)
     << [and so on]…
```

1.3 MOTIVACIÓN

La motivación principal del proyecto es desarrollar una herramienta de monitorización de los contadores hardware de rendimiento que elimine los requisitos de arquitectura, sistema operativo o versión de kernel que limitan la portabilidad en las herramientas actuales.

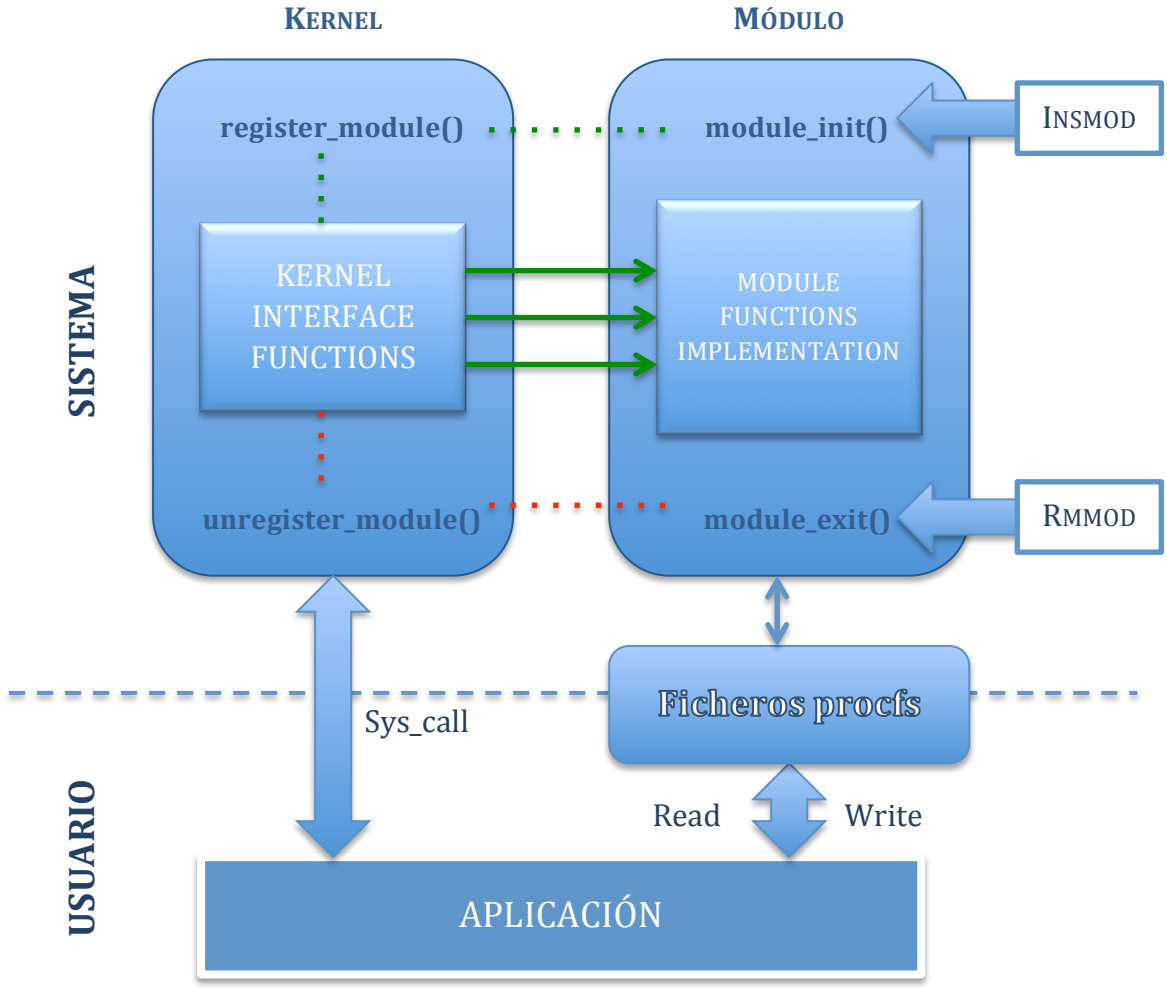
Esta portabilidad es posible aislando al máximo las modificaciones a realizar en el kernel del sistema operativo, trasladando la programación de los contadores hardware a módulos independientes. El diseño modular posibilita la portabilidad, siendo necesaria la definición de una interfaz que enlace el modulo con el kernel para hacer posible la manipulación de los contadores.

Por ultimo, se ha creado un herramienta programada en C llamada ***pmctrack*** inspirada en *cputrack*. Permite tanto monitorizar a nivel de usuario el rendimiento de un programa como configurar los registros eligiendo entre una gran variedad de eventos. Algunos de estos eventos pueden ser la medición de fallos de cache, los accesos a la misma, las instrucciones por ciclo, los fallos de TLB, etc.

Gracias a la modularidad de este proyecto, evitamos la dependencia del kernel del sistema operativo. Esta división es importante para facilitar el desarrollo, la depuración y el mantenimiento. La interfaz de uso es similar que la herramienta de Solaris *cputrack*. Por otro lado se ha validado nuestra herramienta comparando resultados de rendimiento con *cputrack* (ver capítulo 6.1 Validación).

A día de hoy, la herramienta *pmctrack* puede ser utilizada en las siguientes arquitecturas: Xeon, Atom, Core™2 Duo y AMD. Sin embargo, la extensión a otras arquitecturas o a las que aparezcan en el futuro es sencillo gracias al diseño modular empleado.

La figura adjunta resume gráficamente el trabajo desarrollado de este proyecto.



1.4 RESUMEN DEL CONTENIDO

A lo largo de la memoria profundizaremos en el desarrollo del proyecto. La división de temas está estructurada de la siguiente manera:

- **Capítulo 2:** Modificaciones realizadas en el kernel de GNU-Linux para dar soporte a la interacción con los contadores hardware. Detalle de diversos aspectos sobre la creación y liberación de procesos así como la labor del planificador de tareas.
- **Capítulo 3:** Exposición del funcionamiento de los módulos del kernel. Creación, estructuración interna, instalación y desinstalación en el kernel. La mayor parte de codificación recae en esta sección por lo que se ha analizado en detalle la creación de un módulo para Intel® Core™2 Duo.
- **Capítulo 4:** En este capítulo se introduce el sistema de ficheros procfs que servirán de interfaz en los contadores hardware, el módulo del kernel y la herramienta desarrollada.

- **Capítulo 5:** Funcionamiento e implementación de la herramienta desarrollada *pmctrack*.
- **Capítulo 6:** Comprobación de resultados entre *cputrack* y la herramienta desarrollada *pmctrack*. Validación de la herramienta pmctrack, mediante la comparación entre esta y cputrack. Resultados experimentales usando la herramienta pmctrack sobre algunos benchmarck del SPEC2006. Ejecución de *pmctrack* sobre un procesador AMD.
- Conclusiones finales del proyecto y apéndices con información relevante sobre los contadores hardware.

CAPÍTULO 2

MODIFICACIÓN DEL KERNEL GNU-LINUX

La estructuración del kernel de cualquier sistema operativo es muy compleja. En este capítulo trataremos algunos conceptos importantes como las estructuras internas de los procesos y la información que portan, su creación y su liberación de memoria así como su planificación. Se explicará también la relación que existe entre procesos padres e hijos, debido a que es relevante para nuestra herramienta desarrollada.

Por último, una vez entendido el funcionamiento general del kernel, se explicará la integración de ciertas funciones que permitirán la manipulación de los contadores hardware por parte del sistema.

2.1. ESTRUCTURA DEL KERNEL

El kernel es un complejo software que constituye la parte más importante de cualquier sistema operativo. Es el principal responsable de facilitar a los distintos programas acceso seguro al hardware de la computadora. Además, se encarga de la gestión de recursos a través de servicios de llamada al sistema.

En el caso de Unix el lenguaje utilizado casi en su totalidad es C, a excepción del manejador de interrupciones que está escrito en lenguaje ensamblador. Opera como asignador de recursos para cualquier proceso que necesite hacer uso de las facilidades de cómputo. Entre las tareas que realiza podemos destacar:

- Creación de procesos, asignación de tiempos de atención y sincronización.
- Asignación del procesador a los procesos que lo requieren.
- Administración de espacio en el sistema de archivos, que incluye: acceso, protección y administración de usuarios; comunicación entre usuarios y entre procesos, y manipulación de E/S y administración de periféricos.
- Supervisión de la transmisión de datos entre la memoria principal y los dispositivos periféricos.

Consta de dos partes principales: la sección de control de procesos y la de control de dispositivos. La primera asigna recursos, programas, procesos y atiende sus requerimientos de servicio; la segunda, supervisa la transferencia de datos entre la memoria principal y los dispositivos del ordenador.

Introducir modificaciones en el kernel requiere una comprensión detallada de algunos conceptos como la información asociada a un proceso y su almacenamiento, el orden de atención de los mismos y su creación o liberación de los recursos del sistema. Este capítulo ahonda en estos conceptos con el objetivo de aclarar las modificaciones que han de introducirse para soportar la interacción con los contadores hardware.

2.2 PROCESOS

Antes de comenzar, sería conveniente remarcar la diferencia entre dos conceptos parecidos pero no iguales: programa y proceso. Un programa es una colección de instrucciones que el procesador interpreta y ejecuta. Los programas se almacenan de modo permanente en memoria secundaria. Posteriormente, se mueven a memoria principal para poder ser ejecutados. Por otro lado, un proceso es un programa en ejecución. Como consecuencia de ello, el sistema operativo le asigna recursos como memoria, dispositivos, archivos, CPU, etc.

En Linux, cada proceso del sistema tiene descriptor asociado que lo identifica como único. Este descriptor es un estructura llamada **`task_struct`** que almacena toda la información relacionada con el proceso como aspectos de planificación,

identificadores, relación con otros procesos, memoria utilizada por el proceso o los archivos abiertos por el mismo.

Para este proyecto es importante conocer los posibles estados en los que se puede encontrar un proceso a lo largo de su vida. El campo de la estructura `task_struct` que se encarga de portar esta información es `exit_state`. En él se describe la situación en que se encuentra el proceso al haber dejado de ejecutarse en la CPU.

- **TASK_RUNNING:** El proceso es ejecutable y por tanto se encuentra en la Cola de Prioridad, ya sea en ejecución, o esperando su turno. El valor por defecto de esta constante es 0.
- **TASK_INTERRUPTIBLE:** El proceso está suspendido esperando a que se cumpla alguna condición. Cuando esto ocurra, el proceso será despertado y puesto en estado TASK_RUNNING. Sin embargo, también puede ser despertado de forma prematura si recibe alguna señal (por ejemplo, SIGTERM). Los procesos interactivos suelen estar suspendidos en este estado.
- **TASK_UNINTERRUPTIBLE:** Igual que el caso anterior, salvo que el proceso no será despertado si recibe alguna señal.
- **TASK_TRACED:** El proceso está siendo depurado.
- **TASK_STOPPED:** La ejecución del proceso ha sido detenida. En este caso el proceso ya no se ejecuta ni puede ser elegido por el planificador para ejecución. Esto ocurre cuando la tarea recibe las señales SIGSTOP, SIGSTP, SIGTTIN o SIGOUT, o bien, si se recibe cualquier señal mientras está siendo depurada.
- **EXIT_ZOMBIE:** El proceso ha acabado, pero su descriptor sigue en memoria por si el padre necesita alguna información. Dicho descriptor será eliminado cuando el padre ejecute la llamada al sistema `sys_wait`.
- **EXIT_DEAD:** El proceso ha acabado y se ha eliminado su descriptor.

El mecanismo más empleado para cambiar el estado de un proceso es utilizando las macros `set_task_state(task, state)` y `set_current_state(state)`. La diferencia es que la segunda sólo cambia el estado del proceso *current*, que es aquel que se encuentra actualmente en ejecución.

Las transiciones entre estados de los procesos quedan reflejadas en la figura 2.1. Pueden observarse funciones como `fork()`, `do_exit()`, `schedule()` o `context_switch()` que se llaman en la creación, liberación, planificación y cambio de contexto de un proceso respectivamente. Se hablará de cada una de ellas más adelante.

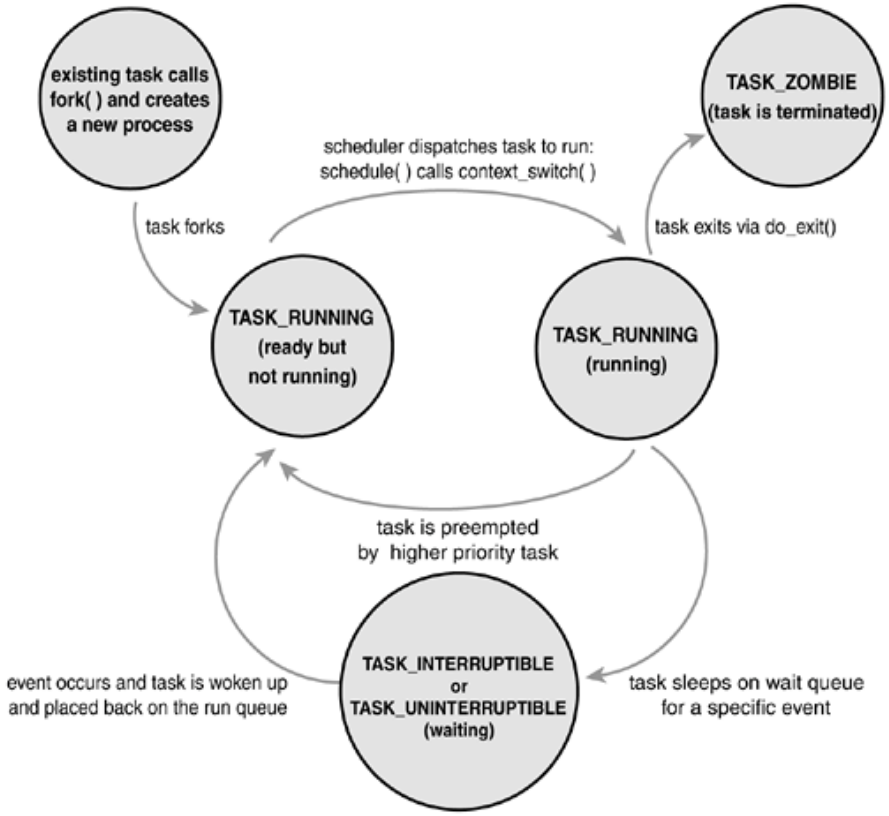


Figura 2.1: *Estados de los procesos*

2.2.1 CREACIÓN DE PROCESOS: FORK Y CLONE

Los procesos en Linux tienen una estructura jerárquica, es decir, un proceso padre puede crear un nuevo proceso hijo y así sucesivamente. La forma en que un proceso crea a otro es mediante una llamada al sistema: *fork* o *clone*.

Ambas llamadas al sistema tienen la misma funcionalidad, pero distintas características:

fork: En el momento de la llamada a *fork* el proceso hijo:

- Es una copia exacta del padre excepto el PID.
- Tiene las mismas variables y ficheros abiertos.
- Las variables son independientes (padre e hijo tienen distintas memorias).
- Los ficheros son compartidos (heredan el descriptor).

clone: Permite especificar qué queremos que compartan padre e hijo.

- Espacio de direccionamiento.
- Información de control del sistema de archivos (*file system*).
- Descriptores de archivos abiertos.

- Gestores de señales o PID.

En resumen, cuando se hace un *fork*, se crea un nuevo descriptor de proceso a partir del `task_struct` del proceso padre. Al hijo se le asigna un PID propio y se le copian las variables del proceso padre. Sin embargo, en la llamada al sistema *clone* el `task_struct` del proceso padre se copia y se deja tal cual, por lo que el hijo tendrá el mismo PID y obtendrá (físicamente) las mismas variables que el proceso padre. El proceso hijo creado es una copia del padre (mismas instrucciones, misma memoria). Lo normal es que a continuación el hijo ejecute una llamada al sistema *exec*. En cuanto al valor devuelto por *fork*, depende de varias opciones:

- Si se produce algún error en la ejecución, el valor devuelto es -1.
- Si no se produce ningún error y nos encontramos en el proceso hijo, se devuelve el valor 0.
- Si no se produce ningún error y nos encontramos en el proceso padre, el valor devuelto es el PID asignado al proceso hijo.

Ambas llamadas al sistema terminan ejecutando **`do_fork()`** que es la función principal del archivo `fork.c`. A continuación, se explicará a grandes rasgos su funcionamiento:

- Invocar a `copy_process` para crear un proceso, generando un nuevo `task_struct` y sus recursos. Una vez creado el descriptor del proceso hijo, se le asignará una CPU llamando a `sched_fork`.
- Obtener el identificador del proceso hijo que se creó en la función `copy_process`.
- Devolver el identificador del proceso hijo.

2.2.2 FINALIZACIÓN DE PROCESOS: EXIT

La llamada al sistema *exit* es la primera en la *sys_call_table* y es la función que termina con la ejecución de un proceso. Esta función se encarga de retirar los recursos que están siendo utilizados por el proceso, así como dejarlo preparado para su posterior eliminación. Cuando un proceso termina debe de comunicar a su padre su finalización por medio de la señal SIGCHLD, de forma que una vez el padre haya sido informado y realice un *wait*, el hijo sea totalmente eliminado, borrándolo de la tabla de procesos. Para reflejar este hecho, se denomina al estado transitorio entre la comunicación de la finalización y la eliminación total como estado *zombie*.

Además de notificar al padre, *exit* se encarga de liberar recursos como la memoria tomada por el proceso, así como el sistema de ficheros y las entradas en el mismo y los registros de estado del procesador. Si dicho proceso no tuviera padre, ya que éste hubiera concluido antes que él, se eliminaría directamente del planificador sin comunicar su estado de finalización.

Interiormente, *exit* realiza una llamada a la función **`do_exit()`**. Ésta es la función principal del archivo `exit.c` y es la que realmente se encarga de la liberación de los recursos. Como se ha comentado anteriormente, `do_exit()` puede realizar una de las siguientes tareas según el estado del padre:

- Si el padre existe, se invoca a la función `exit_notify`, la cual notifica al padre la terminación de uno de sus hijos quedándose éste en estado *zombie*.
- En el caso de que el padre haya terminado sin realizar ningún *wait* para esperar a sus procesos hijos, se llamaría a la función `release_task`. Ésta es la que se encarga de liberar por completo el proceso del sistema.

2.2.3 ESPERAR POR UN PROCESO: WAIT

La función *wait* suspende la ejecución del proceso actual hasta que un proceso hijo haya terminado, o hasta que se produzca una señal cuya acción es interrumpir el proceso actual llamando al manejador correspondiente. Cuando un proceso hijo finaliza, quedándose en estado *zombie,* y el padre realiza una llamada a la función *wait*, se captura su estado de terminación. Posteriormente, se elimina de la tabla de procesos y se informa al padre.

Se puede esperar por un hijo mediante una familia de funciones que tienen como objetivo principal esperar hasta que el estado de los hijos lanzados cambie para retornar la información de dicho proceso. El cambio de estado de los hijos por los que esperar viene determinado por las opciones de la llamada al sistema.

Las llamadas tiene esta sintaxis:

```
pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options)
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options)
```

La función *waitpid* suspende la ejecución del proceso en curso hasta que un hijo especificado por el argumento `pid` termine. Sus argumentos de entrada son:

- **`int *status:`** Puntero donde debe el hijo devuelve su información el hijo.
- **`pid_t pid:`** Un valor tal que:
  - **< -1:** Esperar por cualquier proceso cuyo Process Group ID sea igual al valor absoluto de pid.
  - **-1:** Esperar por cualquier hijo.
  - **0:** Esperar por cualquier hijo cuyo Process Group ID sea igual al del proceso llamador.
  - **> 0:** Esperar por el hijo cuyo PID sea igual al valor de pid.
- **`int options:`** Una combinación de los siguientes flags:

- **WEXITED:** Espera por hijos que hayan terminado.
- **WSTOPPED:** Espera por hijos que hayan sido parados por recibir una señal.
- **WNOHANG:** Comprueba si el hijo ha terminado sin bloquear el proceso que está en ejecución.
- **WNOWAIT:** Dejar al hijo sin modificar ni marcar en la tabla de procesos, tal que una posterior llamada se comportaría como si no hubiésemos hecho *wait* por dicho hijo.
- **WUNTRACED:** No esperar si el hijo está parado, a no ser que esté siendo trazado.
- **WCONTINUED:** Volver si un hijo ha continuado ejecutándose tras mandarle la señal SIGCONT (desde la versión 2.6.10).

Los flags WUNTRACED y WCONTINUED solo son efectivos si SA_NOCLDSTOP no ha sido establecido para la señal SIGCHLD.

Si *status* no es NULL, *wait* o *waitpid* almacena la información de estado en la memoria apuntada por *status*. Ambos devuelven el identificador del proceso hijo que ha finalizado. En caso de error devuelven -1. Y en el caso de utilizar *waitpid* con la opción WNOHANG y ningún hijo esté disponible, se devuelve 0.

Por otro lado, la llamada *waitid* es una función análoga a las otras dos con la diferencia de que ésta proporciona un mayor control sobre qué cambios de estado en el hijo deben ser esperados. En este caso, si la función tiene éxito la información del proceso hijo se devuelve en la variable `infop`.

Devuelve 0 si tuvo éxito o si se utilizó WNOHANG y no había hijos por los que esperar. Si hubo error devuelve -1.

2.3 PLANIFICADOR DE TAREAS

El planificador es un componente funcional muy importante de los sistemas operativos multitarea y multiproceso, y es esencial en los sistemas operativos de tiempo real. Su función consiste en repartir el tiempo disponible de un microprocesador entre todos los procesos que están disponibles para su ejecución.

Linux usa un algoritmo razonablemente simple para planificar las prioridades y seleccionar el proceso siguiente. Cuando ha elegido un nuevo proceso para ejecutar, el planificador salva el estado del proceso en curso, los registros específicos del procesador y otros contextos en la estructura de datos `task_struct`. Luego restaura el estado del nuevo proceso (que también es específico a un procesador) para ejecutarlo y da control del sistema a ese proceso. Para que el planificador asigne el tiempo de la CPU justamente entre los procesos ejecutables en el sistema, el planificador mantiene

cierta información en la estructura `task_struct` de cada proceso. La información más relevante es la siguiente:

- **policy:** Esta es la política de planificación que se aplicará a este proceso. Hay dos tipos de procesos en Linux, normales y de tiempo real. Los procesos de tiempo real tienen una prioridad más alta que los otros. Si hay un proceso de tiempo real listo para ejecutarse, siempre se ejecutará primero. Pueden tener dos tipos de políticas: ”Round Robin” (en círculo) y FIFO (el primero en llegar es el primero en salir). En la planificación ”Round Robin”, cada proceso de tiempo real ejecutable se ejecuta por turnos, y en la planificación FIFO cada proceso ejecutable se ejecuta en el orden que están en la cola de ejecución y el orden no se cambia nunca.
- **priority:** Esta es la prioridad estática que el planificador dará a este proceso. También es la cantidad de tiempo que se permitirá ejecutar a este proceso una vez que sea su turno de ejecución. La prioridad de un proceso puede ser modificada.
- **rt_priority:** Linux soporta procesos de tiempo real. Éstos tienen una prioridad más alta que todos los demás procesos en el sistema. Este campo permite al planificador darle a cada proceso de tiempo real una prioridad relativa. La prioridad del proceso se puede alterar mediante llamadas de sistema.
- **time_slice:** Es el llamado quantum o rodaja de tiempo. Representa la cantidad de tiempo que se permite ejecutar este proceso en la CPU. Se decrementa a cada paso de reloj. Su valor inicial es asignado según la prioridad.

La planificación en Linux se lleva a cabo en la función **`schedule()`** del fichero `kernel/sched.c`. El principal objetivo es el cambio de contexto, seleccionando un proceso de la lista de ejecutables y asignándolo a una CPU. Un cambio de contexto puede ocurrir por varios motivos:

- Una señal bloqueante.
- Finalización del proceso en ejecución.
- El *quatum* o rodaja de tiempo asignada al proceso actual se consuma.
- Se despierte un proceso cuya prioridad es mayor que la del proceso actual.

Se definen dos punteros de descriptores de procesos: `prev` y `next`. El primero representa el proceso que va a dejar la CPU, el *current*, y el segundo, el que va a ser elegido para entrar a ejecutarse. La función que se encarga de decidir cuál es el siguiente proceso que pasará a ejecutarse es `pick_next_task()`. Una vez seleccionado el proceso de mayor prioridad de la *runqueue* correspondiente, se procede al cambio de contexto mediante la llamada a otra función, `context_switch()`. En el caso de que `next` y `prev` coincidieran no se produciría ningún cambio de contexto.

Otra de las funciones que es importante comentar debido a su relación directa con el proyecto es **`scheduler_tick()`**. Se invoca en cada “tick” de reloj y se encarga

de disminuir el contador de la fracción de tiempo del proceso actual, y comprobar si el *quantum* se ha agotado. Además realiza balanceado para asegurar que las colas de ejecución de las diferentes CPUs contienen aproximadamente los mismos procesos ejecutables.

2.4 MODIFICACIONES

Al utilizar la modulación, la mayor parte del código será implementado en los distintos módulos de las diferentes arquitecturas. Definiendo una interfaz en el kernel se consigue enlazar éste con los módulos, introduciendo las llamadas a las funciones en partes muy concretas del kernel y definiéndolas en el módulo correspondiente. Dadas las escasas modificaciones realizadas y la similitud de los sistemas operativos a la hora de crear, planificar y finalizar procesos, la integración en otros sistemas no es muy diferente de la realizada.

La idea principal de la herramienta que se ha desarrollado es conseguir que un proceso padre monitorice a su hijo, siendo éste el programa deseado por el usuario. Para ello el hijo tiene que ser capaz de activar y desactivar los contadores y almacenar sus valores, y el proceso padre, poder leer dichos valores y monitorizarlos. Esto es posible añadiendo nuevos campos en el descriptor de proceso o estructura `task_struct`.

- **`int prof_enable:`** Inicialmente toma valor 0, es decir, cuando este proceso tome el control de la CPU los contadores están desactivados. En el caso de querer activar los contadores este valor tendrá que ser modificado a 1.
- **`void *pmc:`** Al ser una estructura que está definida en el módulo por ser dependiente de arquitectura, dentro del `task_struct` será un puntero de tipo `void`. Esto es así porque al arrancar el sistema operativo éste no conoce su situación exacta hasta que el módulo no es instalado. La estructura se llama `pmon_prof_t` (ver subcapítulo 3.2 Implementación del módulo). En ella se almacenará la información necesaria para hacer posible la monitorización de los contadores.

Conocidos los cambios realizados en el descriptor de procesos, la forma en que el kernel o el usuario puede acceder a ellas es a través de las funciones de la interfaz mencionada anteriormente. La estructura de dicha interfaz está definida en el archivo `mc_experiment.h` de la siguiente manera:

```
typedef struct __pmc_mc{
      void* (*init_pmon_prof_t);
      int   (*pmc_config_read);
      int   (*pmc_config_write);
      int   (*monitor_pmcs_read);
      int   (*monitor_pmcs_write);
      int   (*pmc_enable_read);
      int   (*pmc_enable_write);
      void  (*amp_save_callback);
      void  (*amp_restore_callback);
      void  (*amp_pmon_tick);
      void  (*free_pmon_prof);
} pmc_mc;
```

Las funciones de lectura y escritura están relacionadas con los ficheros del /proc, por lo que tiene un tratamiento diferente. Este tema será explicado en el cuarto capítulo.

La integración del resto de las funciones en el kernel serán desarrolladas a continuación. Sin embargo, una descripción más detallada de su implementación se llevará a cabo en el capítulo siguiente.

- **`init_pmon_prof_t:`** Esta función se encarga de reservar memoria e inicializar la nueva estructura `pmon_prof_t` integrada en el descriptor del proceso cuando éste sea creando. Al principio de este capítulo se muestra los pasos a seguir durante la creación de un proceso. Dentro de la función `do_fork` se llama a la función `copy_process`, que recibe como parámetros de entrada la información necesaria del proceso padre que se va a duplicar. Una vez generado el nuevo `task_struct`, o descriptor del proceso hijo, se llama a `sched_fork`, que realiza la planificación del nuevo proceso asignándole una CPU. Este es el momento más adecuado para invocar esta función. Además, justo después de la llamada, también se inicializa la variable `prof_enable` a cero.

```
static struct task_struct *copy_process(unsigned long clone_flags,
                                unsigned long stack_start,
                                struct pt_regs *regs,
                                unsigned long stack_size,
                                int __user *child_tidptr,
                                struct pid *pid,
                                int trace)
{
      struct task_struct *p;
      ...

      p = dup_task_struct(current);
      ...

      /* Performance monitoring counter */
      p->pmc = init_pmon_prof_t(p);
      p->prof_enabled = 0;

      /* Perform scheduler related setup. Assign this task to a CPU. */
      sched_fork(p, clone_flags);
      ...

      return p;
}
```

- **amp_save_callback:** Cuando se realiza un cambio de contexto y el proceso actual tiene activado los contadores (`p->prof_enable = 1`) es necesario salvar el valor de los contadores y detenerlos ya que el proceso va a ser expulsado de la CPU. Esta es la finalidad de `amp_save_callback`. Como ya sabemos, la función que se encarga de la planificación es el `schedule`, así que aquí es donde se incluirá la llamada a la función, justo antes del cambio de contexto (`context_switch`).
- **amp_restore_callback:** Esta función realiza la tarea opuesta a la anterior. En este caso, el proceso que ha sido elegido para entrar al procesador es el que tiene activado los contadores. Por lo tanto, la función se encarga de restaurar los contadores y activarlos para que vuelvan a contar los eventos correspondientes. La situación en el kernel es similar a la de `amp_save_callback`.

```
asmlinkage void __sched schedule(void)
{
      struct task_struct *prev, *next;
      unsigned long *switch_count;
      struct rq *rq;
      int cpu;

need_resched:
      preempt_disable();            // Se desactiva el cambio de contexto
      cpu = smp_processor_id();     // Se obtiene el ID de la CPU actual
      rq = cpu_rq(cpu);             // Cola de procesos asignada a la CPU
      rcu_note_context_switch(cpu);
      prev = rq->curr;              // prev guarda el proceso actual
      ...

      next = pick_next_task(rq);    // next es el nuevo proceso a ejecutar
      ...

      /* Performance monitoring counter */
      if(prev->prof_enabled) amp_save_callback(prev->pmc, cpu);
      if(next->prof_enabled) amp_restore_callback(next->pmc, cpu);

      context_switch(rq, prev, next); // Realiza el cambio de contexto
      ...
}
```

- **amp_pmon_tick:** En el caso de que la herramienta quisiera leer los contadores y el proceso no haya dejado la CPU por cualquier motivo, la lectura que se realizaría sería errónea. Hasta el momento, sólo se guardan los valores de los contadores cuando se realiza un cambio de contexto mediante la función `amp_save_callback`. La solución es salvar esos valores no sólo cuando salga el proceso de la CPU sino también cuando se produzca una interrupción de reloj. El sitio más conveniente de insertar esta función es en la llamada a la función `scheduler_tick` para reducir el contador de la fracción de tiempo del proceso en ejecución.

```
void scheduler_tick(void)
{
      int cpu = smp_processor_id();
      struct rq *rq = cpu_rq(cpu);
      struct task_struct *curr = rq->curr;

      sched_clock_tick();

      /* Performance monitoring counter */
      if(curr->prof_enabled) amp_pmon_tick(curr->pmc, cpu);

      ...
}
```

- **free_pmon_prof:** En la creación de los procesos se realiza una reserva de memoria del kernel en la función `init_pmon_prof_t` para almacenar toda la información relacionada con los contadores. Esa memoria tiene que ser liberada cuando el proceso haya finalizado. En el capítulo anterior (2.1.1 Finalización de procesos), se habla de la función `do_exit` que es invocada cada vez que un proceso llegue a su fin. Una vez llamada dicha función, quien se encarga de la liberación de recursos es `release_task`. Es aquí donde se insertará la función `free_pmon_prof` que se detallará más adelante.

Estas son las modificaciones mas relevantes del kernel. Aparte de las descritas en este capítulo, existen otras funciones que sin ellas no sería posible la comunicación del kernel con el módulo a través de la interfaz anterior. Al instalarse o desinstalarse el módulo, hay que comunicarle al kernel que las funciones insertadas se encuentran definidas dentro del módulo. Además, cuando el módulo es instalado también es necesario generar las entradas del */proc* que hacen posible la comunicación de la herramienta de monitorización con el sistema. Estás funciones, que se encuentran definidas en el archivo `mc_experiment.c` se llaman `register_pmc_module` y `unregister_pmc_module`.

Por otro lado, ¿qué pasaría si no hubiera ningún módulo instalado? Las nuevas funciones introducidas en el kernel no estarían implementadas en ningún sitio. Para evitar este problema, se definen también en el `mc_experiment.c` una función *wrapper* por cada una de las funciones de la interfaz. Estos *wrappers* consisten en comprobar si existe un módulo instalado, y por lo tanto la implementación de la función correspondiente, para así poder seguir con la ejecución. En el caso de no existir el módulo, la función no realizaría ninguna tarea. De esta manera se evitan que se produzcan fallos críticos en el sistema.

CAPÍTULO 3

MÓDULOS

La arquitectura del kernel de Linux permite cargar y descargar extensiones al sistema operativo en forma de módulos sin tener que detener o reiniciar el sistema. Esto implica una serie de ventajas:

- Facilita la depuración del código.
- Disminuye el requerimiento de memoria del sistema.
- Evita la reconstrucción un nuevo kernel monolítico cada vez que se quiera añadir un nuevo dispositivo.
- Y todo ello desencadena una disminución considerable de tiempo y una mayor flexibilidad a la hora de trabajar.

A demás de las ventajas que conlleva la utilización de módulos, también permiten uno de los objetivos de este proyecto: conseguir que la herramienta de monitorización de rendimiento sea portable a cualquier arquitectura.

Su compilación es considerablemente más rápida y la depuración en tiempo real es mucho más eficaz e intuitiva.

A lo largo del capítulo, se explicará la forma de instalar y desinstalar un módulo mediante línea de comandos. Su implementación estará enfocada a una arquitectura basada en el procesador Intel® Core™2 Duo. La parte desarrollada para los procesadores AMD es análoga exceptuando el modo configuración ya que es dependiente del tipo y el número de contadores de cada arquitectura.

3.1 INSTALACIÓN Y DESINSTALACIÓN

Una vez compilado el módulo y generado el archivo *.ko*, instalarlo y desinstalarlo no demora mucho tiempo. Los comandos utilizados para la gestión de los módulos son los siguientes:

- **insmod:** Seguido del paquete generado tras la compilación del módulo (.ko), inserta el módulo en el kernel.
- **rmmod:** Seguido en este caso del nombre de un módulo previamente instalado, lo elimina del kernel.
- **lsmod:** Lista todos los módulos instalados en el kernel actual.

El módulo debe cumplir unos requisitos estructurales. Se utilizan unas macros definidas en el archivo *<linux/init.h>*. Éstas son ***module_init()*** y ***module_exit()***. Ambas reciben por parámetro una función que inicializará o liberará los recursos del módulo.

Por otra parte, podemos incorporar información como el autor, la licencia, la tabla de dispositivos, una descripción, etc. Basta con importar el archivo *<linux/module.h>* que define las macros correspondientes.

3.2 IMPLEMENTACIÓN

Cómo vimos en el capítulo anterior, al instalar nuestro módulo se crean las entradas del */proc* y se establece el enlace de las funciones declaradas en la interfaz con las del módulo. Otra función añadida es la configuración inicial de los contadores hardware, tanto los fijo como los configurables, asignado unos eventos por defecto.

- **Contadores fijos:** En la arquitectura del Core™2 Duo son tres, y los eventos asignados son fijos y no pueden ser modificados por el usuario ("instrucciones por ciclo", "ciclos de reloj" y "ciclos de bus").
- **Contadores configurables:** Son dos y en este caso el usuario sí que puede asignarle cualquier evento según lo que se desee contar. Por defecto, los eventos de estos contadores son "fallos de la caché L2" y "accesos de la caché L2", coincidiendo este último con los aciertos de la caché L1.

Dentro del módulo se definen las estructuras utilizadas para hacer posible la monitorización de un proceso. La primera que vamos a describir es ***pmon_prof_t***. Esta estructura ha sido la más complicada de desarrollar, sobretodo a la hora de integrarla en el descriptor de proceso (campo `pmc` del `task_struct`).

```
typedef struct{
      atomic64_t ll_acum[MAX_LL_EXPS];
      struct task_struct *parent_tsk;
      struct task_struct *child_tsk;
      int ticks_counter;
      unsigned int pmc_ticks_counter;
      uint_t ticks_last_pmc_read;
      int ticks_warm_up;
      unsigned int samples_counter;
      unsigned int switch_out_counter;
      unsigned int switch_in_counter;
} pmon_prof_t;
```

El significado de cada una de la variables es el siguiente:

- **ll_acum:** Vector de *MAX_LL_EXPS* elementos, tantos como contadores de rendimiento posea la arquitectura. Guarda el valor de los contadores cada vez que haya una interrupción de reloj o un cambio de contexto.

  MAX_LL_EXPS es una macro definida para cada una de las arquitecturas que representa el número de contadores fijos y configurables. Para el Core™2 Duo es de valor 5 (3 contadores fijos y 2 configurables).

- **parent_task:** Como hemos visto en el capítulo anterior, tras la ejecución de un *fork* se genera un proceso hijo como copia idéntica del proceso en ejecución, el proceso padre. Esta variable es un puntero que, en el caso de pertenecer al proceso hijo, hará referencia a su padre; en cualquier otro caso tomará un valor nulo.

- **child_task:** Análogo a la variable *parent_task*, en el caso de pertenecer al proceso padre, ésta hace referencia al descriptor de proceso del hijo; en cualquier otro caso tomará un valor nulo.

- **ticks_counter:** Número de ticks del contador transferidos a lo largo de la vida del proceso asociado.

- **pmc_ticks_counter:** Acumulador de ticks necesarios para realizar la lectura de los contadores. Cuando llegue al valor permitido, se leerán los valores y se pondrá de nuevo a 0.

- **ticks_warm_up:** Periodo por defecto de preparación previa al conteo real de los contadores.

- **samples_counter:** Número de intervalos de muestreo.

- **switch_out_counter:** Número de veces que el proceso ha sido expulsado de la CPU.

- **switch_in_counter:** Número de veces que el proceso haya tomado la CPU desde la cola de prioridad.

Otra de las estructuras dependiente de arquitectura es ***pmon_config_t***. Su función es la de identificar si las muestras de los contadores son validas o no según el momento

en el que se encuentre el proceso. Por ejemplo, durante el proceso de calentamiento de los contadores, los valores no serían correctos. Inicialmente, esta estructura tomará unos valores por defecto que podrán ser modificados por el usuario a través de una escritura en el fichero *mod_pmc_config* del *procfs* como veremos en el capítulo siguiente.

Anteriormente se dieron a conocer algunas de las funciones más importantes del módulo. Sus objetivos y su situación dentro del kernel ya son conocidas, pero en este capítulo las conoceremos con más detalle.

Al crear un proceso, la reserva de espacio de memoria en el kernel para la estructura `pmon_prof_t` se hace en una variable local dentro de la función `init_pmon_prof_t` mediante la siguiente línea de código:

```
pmon_prof_t *prof = (pmon_prof_t*)kmalloc(sizeof(pmon_prof_t), GFP_KERNEL);
```

En el caso de que existiera algún problema para reservar el espacio indicado, la función `kmalloc` nos devolvería un error que se captaría para informar al usuario. Por el contrario, si el funcionamiento fuera correcto, se inicializarían los campos de la variable `prof, devolviendo` su referencia para que la variable `pmc` insertada en el `task_struct` del proceso (ver subcapítulo 2.3 Modificaciones del kernel) apunte al espacio de memoria reservado para esta variable local.

La memoria del kernel no es infinita, por lo que de la misma forma que se reserva espacio cuando se crea un proceso hay que asegurarse de su liberación a la finalización de mismo. La liberación de la estructura `pmon_prof_t` de un proceso se realiza en la función `free_pmon_prof` que recibe por parámetro el `task_struct` del proceso en cuestión. El código que se encarga de la eliminación completa de la estructura es el siguiente:

```
kfree((pmon_prof_t*)tsk->pmc);
tsk->pmc = NULL;
```

Antes de la liberación, debemos tener cuidado con las referencias entre padre e hijo por medio de los campos `parent_task` y `child_task` definidos más arriba, ya que dejar un puntero asignado a un espacio de memoria inexistente puede dar lugar a un error critico en el sistema.

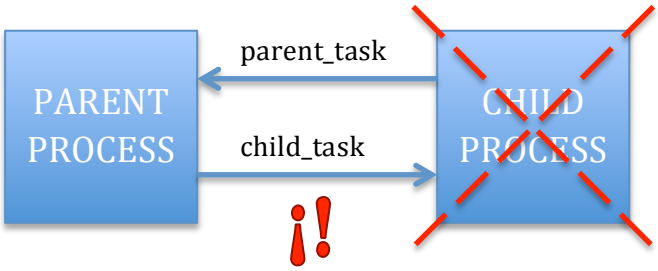


Figura 3.1: *Liberación de un proceso hijo*

La asignación de `parent_task` y `child_task` las realiza la aplicación desarrollada por medio de una escritura en el fichero *mod_monitor_pmcs* del *procfs* (ver subcapítulo 4.3). La idea es que como el proceso hijo conoce a su creador, éste acceda a él a través de `parent_task` y ponga el campo de `child_task` a *null* antes de su liberación.

Por último, se hablará de la función que se encarga de salvar los valores de los contadores para su posterior lectura: `do_count_mc_experiment`. Como ya se dijo en el capitulo anterior, los contadores son guardados cuando se produce un cambio de contexto llamando a `amp_save_callback`, o simplemente en cada interrupción de reloj con `amp_pmon_tick`. Pues bien, ambas funciones, para realizar dicho guardado, utilizan `do_count_mc_experiment`.

Inicialmente, se comprueba si realmente las muestras se quieren guardar o no; es decir, no seria necesario salvar el valor de los contadores si éstos se encuentran en la preparación previa al conteo real o, simplemente, no interese salvarlos en ese preciso instante. Posteriormente, cuando todo indique que los valores de los contadores son correctos y se quieran guardar, se paran los contadores, se realiza una lectura de los mismos y se establece de nuevo su funcionamiento. Por último, se obtienen los valores leídos para acumularlos en el vector de resultados de los contadores `ll_acum` definido en la estructura `pmon_prof_t`.

```
for(i=0; i<cur_ll_set->size; i++){
      low_level_exp* lle = &cur_ll_set->array[i];

      __stop_count(lle);      // Parada de los contadores
      __read_count(lle);      // Lectura de sus valores
      __start_count(lle);     // Reanudacion de los contadores

      /* Obtenemos el ultimo valor leido */
      last_value = __get_last_value(lle);

      rcu_read_lock();
      atomic64_add(last_value, &prof->ll_acum[i]);
      rcu_read_unlock();
}
```

Como se puede observar, la variable `ll_acum` es de tipo `atomic64_t`. Las operaciones atómicas son el método de sincronización entre procesos más eficiente puesto que se realizan a muy bajo nivel. Consiste en ejecutar operaciones sobre un entero en un único paso indivisible, sin admitir interrupciones. Con ello nos aseguramos que el proceso padre no pueda leer esta variable mientras que se esté actualizando su valor. Quedaría bloqueado mientras que la operación atómica `atomic64_add` no haya terminado. La lectura de la variable `ll_acum` y su posterior restauración se realiza a través del fichero *mod_monitor_pmcs* del *procfs* explicado en el capitulo siguiente.

CAPÍTULO 4

SISTEMA DE FICHEROS PROCFS

En Linux existe un mecanismo para que el kernel y los módulos puedan intercambiar información con los procesos en ejecución, el sistema de ficheros *procfs (process filesystem)*. Es un pseudo-sistema compuesto por un conjunto de archivos virtuales, generalmente montados en el directorio */proc,* que se generan cada vez que se inicia el equipo. Estos archivos no existen físicamente en el disco, sino que se encuentran alojados en la memoria principal de la máquina.

En este capítulo abordaremos las entradas en este sistema de ficheros que han sido creadas para hacer posible la comunicación entre la herramienta desarrollada y el módulo.

4.1 GESTIÓN DE FICHEROS DEL PROCFS

La creación de entradas en el */proc* consiste en añadir un nodo virtual al sistema de ficheros, de forma que cuando se lea o se escriba en el fichero asociado a nuestro nodo, se realizarán unas llamadas al sistema que accederán a las funciones de lectura y escritura implementadas dentro del módulo. Dichas funciones tienen los siguientes prototipos:

```
int (*read)( char *page, char **start, off_t off, int count, int *eof, void
*data )
int (*write)( struct file *file, const char __user *buffer, unsigned long
count, void *data )
```

En la llamada de lectura nos encontramos los siguientes parámetros:

- **`char *page:`** Una página de memoria en espacio de usuario que es en la que hay que escribir la información que queramos devolver al usuario.
- **`char **start:`** Se utiliza para poder realizar lecturas de la entrada en varios accesos. Por ejemplo, si necesitásemos devolver más de los 4k que ocupa una página de memoria, tendríamos que devolver los cuatro primeros kilobytes y dejar el puntero **start* en el punto en el que nos quedamos leyendo.
- **`off_t off:`** Es el punto, dentro de la página, a partir del cual debemos empezar a escribir.
- **`int count:`** Es el número máximo de bytes que debemos devolver.
- **`int *eof:`** Devolveremos 1 en caso de que se haya llegado al final del fichero.
- **`void *data:`** Es información extra que se nos pasa desde el nodo de la entrada.

En cambio, los parámetros de la llamada de escritura son:

- **`struct file *file:`** Un puntero a la estructura del fichero.
- **`const char __user *buffer:`** Memoria del espacio de usuario en la que está la información que están escribiendo en la entrada.
- **`unsigned long count:`** Número de bytes que se pretenden escribir.
- **`void *data:`** Al igual que en la función de lectura, esta información se nos pasa siempre desde el nodo de la entrada del */proc*.

En nuestros caso necesitaremos crear tres ficheros distintos en el *procfs* para poder gestionar la monitorización permitiendo a nuestro programa acceder y modificar la información de los contadores de rendimiento. Estos ficheros son creados en la instalación del módulo y su funcionalidad será especificada a continuación.

4.2 ENTRADA MOD_ENABLE_PMCS

El fichero */proc/mod_enable_pmcs* es utilizado en la comunicación entre la herramienta de monitorización y el módulo para que este pueda realizar las acciones indicadas por el usuario.

En concreto, este fichero va a ser usado para establecer la activación y la desactivación de la monitorización de un proceso. Para ello usamos las siguientes funciones de escritura y lectura que detallamos a continuación:

- **Escritura:** La función usada es la siguiente:

```
static int core2_pmc_enable_write(struct file *filp, const char __user
*buff, unsigned long len, void *data)
```

El uso de los parámetros ha sido descrito en la introducción de este capítulo.

El usuario puede escribir "ON" u "OFF" en este */proc*. Dependiendo de ello el sistema realizará las siguientes acciones:

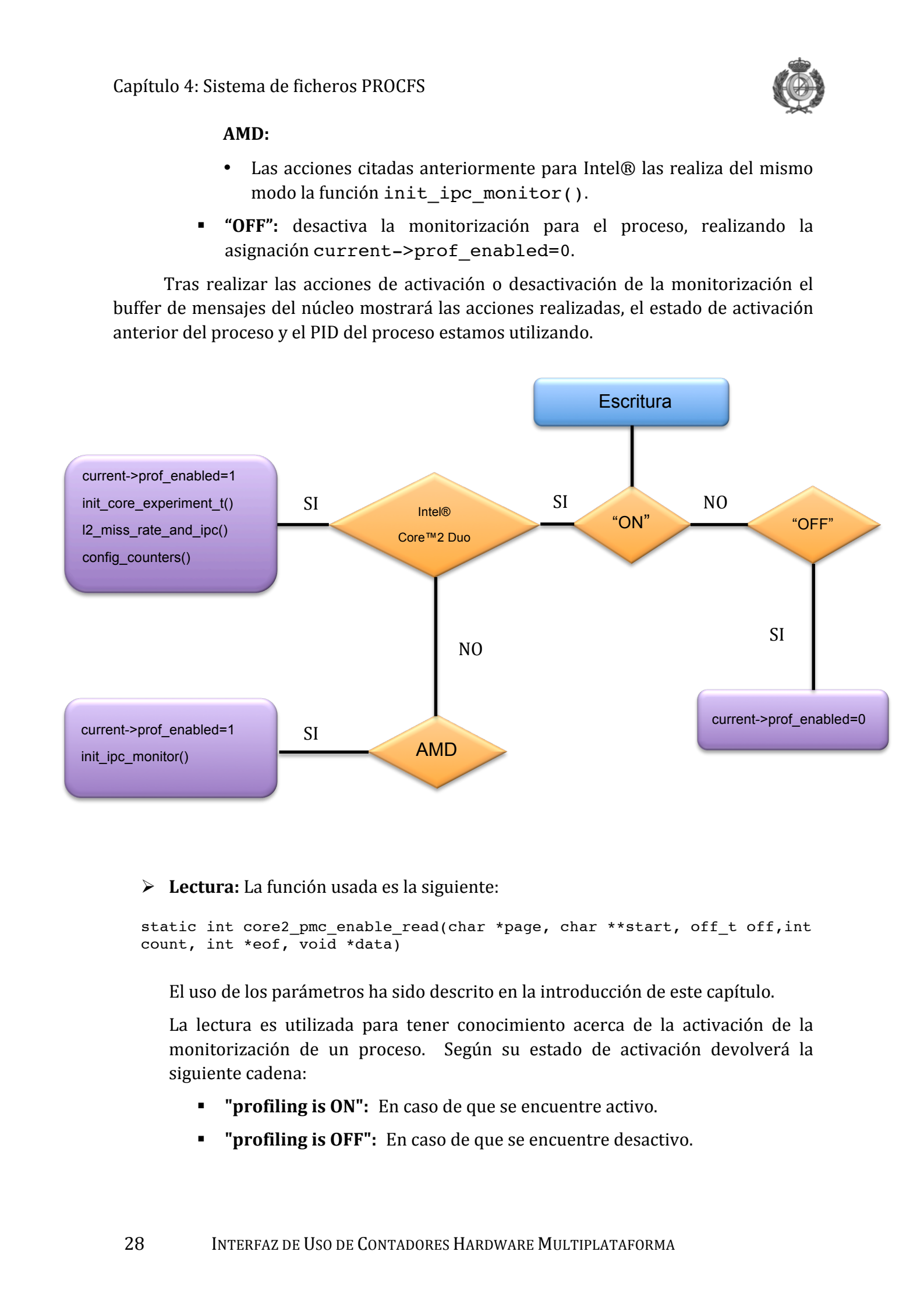
- **"ON":** Activa la monitorización para el proceso realizando la asignación `current->prof_enabled=1`.

  Además se lleva a cabo la configuración de los contadores realizándose una inicialización y configuración de los contadores para cada procesador lógico. En el caso de que el usuario haya establecido una configuración anteriormente, se utilizará esta. Si no ha sido establecida ninguna configuración, los contadores utilizarán una configuración predefinida.

  Dependiendo de la arquitectura usada serán utilizadas las siguientes funciones:

  Intel® Core™2 Duo:

  - init_core_experiment_t(): inicializa los contadores.
  - l2_miss_rate_and_ipc(): establece la configuración de los contadores fijos y la configuración predefinida en los contadores variables en caso de no haber especificado anteriormente una configuración para estos.
  - config_counters(): establece la configuración especificada por el usuario en los contadores variables, en caso de que se haya especificado alguna configuración.
  - Todas estas funciones reciben como parámetro de entrada `&mc_exp.core_experiments[]` con la posición de el procesador lógico que se esté configurando.



AMD:

- Las acciones citadas anteriormente para Intel® las realiza del mismo modo la función `init_ipc_monitor()`.

▪ **"OFF":** desactiva la monitorización para el proceso, realizando la asignación `current->prof_enabled=0`.

Tras realizar las acciones de activación o desactivación de la monitorización el buffer de mensajes del núcleo mostrará las acciones realizadas, el estado de activación anterior del proceso y el PID del proceso estamos utilizando.

➢ **Lectura:** La función usada es la siguiente:

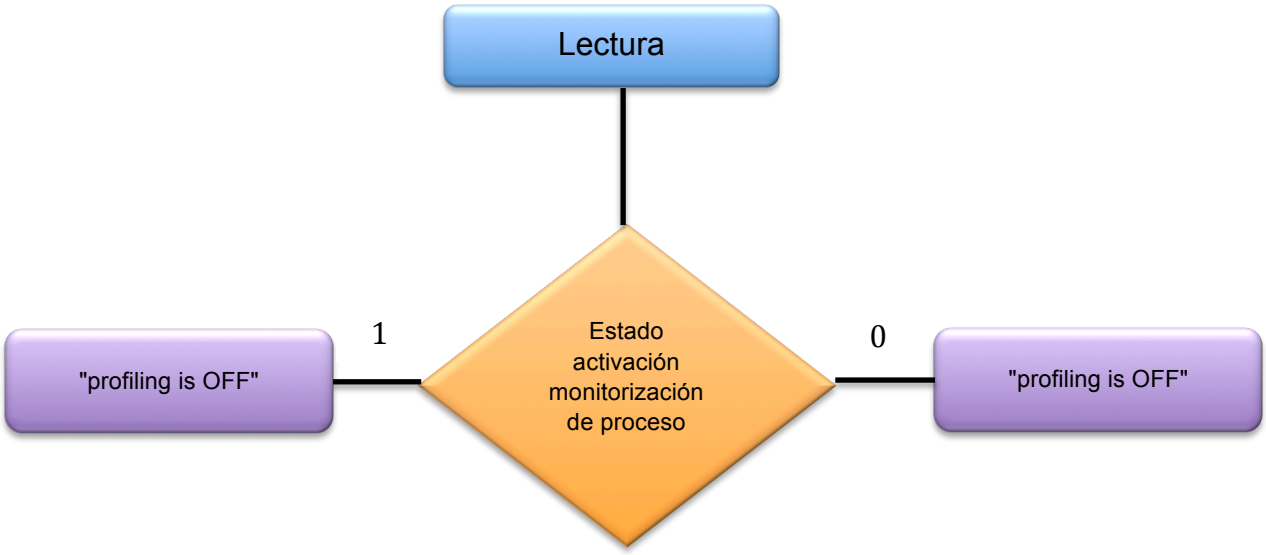
```
static int core2_pmc_enable_read(char *page, char **start, off_t off,int
count, int *eof, void *data)
```

El uso de los parámetros ha sido descrito en la introducción de este capítulo.

La lectura es utilizada para tener conocimiento acerca de la activación de la monitorización de un proceso. Según su estado de activación devolverá la siguiente cadena:

▪ **"profiling is ON":** En caso de que se encuentre activo.

▪ **"profiling is OFF":** En caso de que se encuentre desactivo.



4.3 ENTRADA MOD_PMC_CONFIG

El fichero */proc/mod_pmc_config* es utilizado en la comunicación entre la herramienta de monitorización y el módulo para que este pueda realizar las acciones indicadas por el usuario.

En concreto, este fichero va a ser usado para establecer la configuración tanto general como específica de los contadores hardware. Para ello usamos las siguientes funciones que detallamos a continuación de lectura y escritura del */proc:*

- **Escritura:** La función usada es la siguiente:

```
static int core2_pmc_config_write(struct file *filp, const char __user
*buff, unsigned long len, void* data)
```

El uso de los parámetros ha sido descrito en la introducción de este capítulo.

Con la escritura se configuran los contadores. Existen dos tipos de configuraciones:

- **Configuración general:** este tipo de configuración afecta a todos los contadores hardware y nos permite configurar los siguientes aspectos:
  - Número de ticks entre cada.
    `core2_pmon_config.pmon_nticks`
  - Seleccionar el periodo de calentamiento. Este es un tiempo en el que no se medirán los contadores. Se utiliza para evitar monitorizar operaciones no correspondientes exclusivamente al proceso a monitorizar.
    `core2_pmon_config.pmon_init_warm_up_period`
  - Establecer el periodo en el cuál no se medirán los contadores tras una

migración.
`core2_pmon_config.pmon_migr_warm_up_period`

- Establecer el número de ticks consecutivos desde el último cambio de contexto para empezar a considerar el valor que ofrecen los contadores.
`core2_pmon_config.pmon_min_pmc_cons_tick`

Esta configuración se realiza escribiendo en este fichero el aspecto que se quiere modificar, seguido de un espacio y el valor deseado.

El buffer de mensajes del núcleo nos ofrecerá información sobre que aspecto hemos configurado.

Otro aspecto a configurar es la selección de los contadores a mostrar. La configuración se llevará a cabo través de la máscara binaria `ctr_mask` que tendrá valor 1 en cada posición del contador que se desee mostrar y 0 en caso de no mostrarse. La máscara se configura a través de las siguientes escrituras:

- **`"init_mask":`** A `ctr_mask` se le asigna valor 0.
- **`"select_all_pics"`** `ctr_mask` tomará tantas posiciones con valor 1 como número de contadores existan para la arquitectura utilizada.
- **“pic$_x$”:** selecciona un contador concreto a mostrar, siendo x el número del contador. En este caso se realizará una operación `OR` para poner un 1 en la posición de acuerdo al contador que se desea mostrar.

- **Configuración específica:** únicamente afecta a un contador.

  Se realiza a través de la llamada a la siguiente función:

  ```
  config_core2_counters(const char __user *buff)
  ```

  El parámetro `buff` es un puntero a la cadena de caracteres que contiene la información de la configuración de un contador.

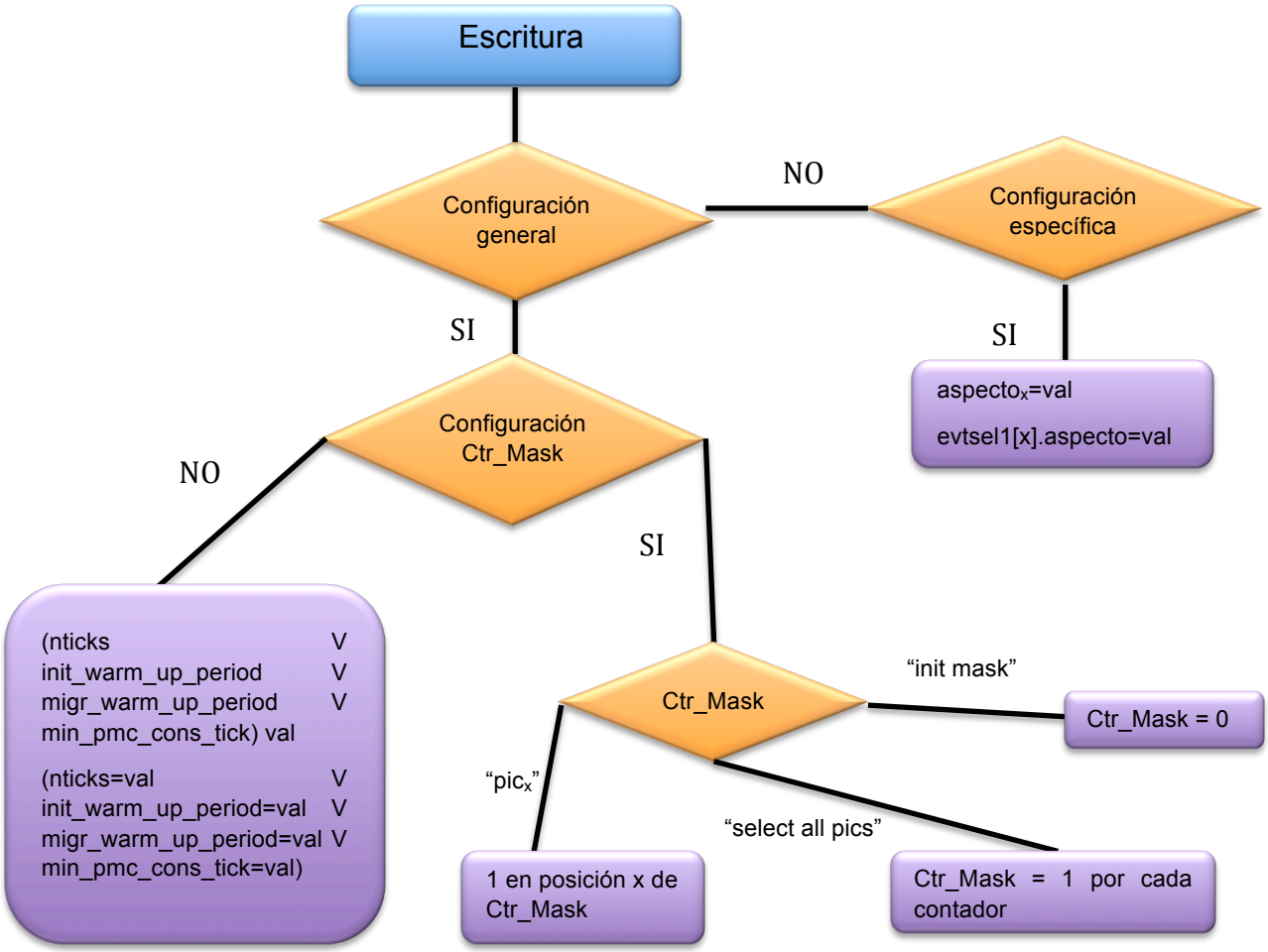
  Los aspectos a configurar son los siguientes: evento a medir, máscara unitaria del evento, modo usuario, modo sistema operativo, contador de máscara, inversor y detección de margen. El uso de estos aspectos está definido en la sección que trata sobre el modo configuración de la herramienta y en los apéndices.

  La forma en la que se realiza la configuración de los contadores es la siguiente:

  1. Escribir en el `/proc/mod_pmc_config` el aspecto a configurar de un contador elegido y el valor que se desea, de la forma: `count`$_x$`=val`, `umask`$_x$`=val`, `usr`$_x$`=val`, `os`$_x$`=val` , `cmask`$_x$`=val`, `inv`$_x$`=val` o `edge`$_x$`=val`; refiriéndose al evento, máscara unitaria, modo usurario, modo sistema operativo, contador de máscara, inversor o detección de margen, respectivamente; x

indica el contador a configurar, siendo este un valor entero; y `val` el valor deseado.

2. Guardar el aspecto configurado en posición correspondiente al número de contador en el vector `evtsel1[]` (variable global del módulo de tipo `evtsel_msr`). El tipo de este vector tiene los campos correspondientes a cada aspecto configurable del contador. Así la configuración de cada contador queda almacenada para así poder acceder a la información en el momento de establecer la configuración de un contador.

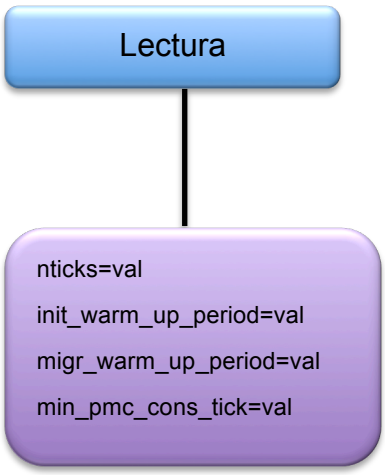


➢ **Lectura:** La función usada es la siguiente:

```
static int core2_pmc_config_read(char* page, char** start, off_t off,int
count, int* eof, void* data)
```

El uso de los parámetros ha sido descrito en la introducción de este capítulo.

La lectura muestra los valores de los aspectos de configuración general: nticks, periodo de calentamiento, periodo de migración y número de ticks para el cambio de contexto.



4.4 ENTRADA MOD_MONITOR_PMCS

El fichero */proc/mod_monitor_pmcs* es utilizado en la comunicación entre la herramienta de monitorización y el módulo para que este pueda realizar las acciones indicadas por el usuario.

En concreto, este fichero va a ser usado para la muestra de resultados. Para ello usamos las siguientes funciones que detallamos a continuación de lectura y escritura del */proc:*

- **Escritura:** La función usada es la siguiente:

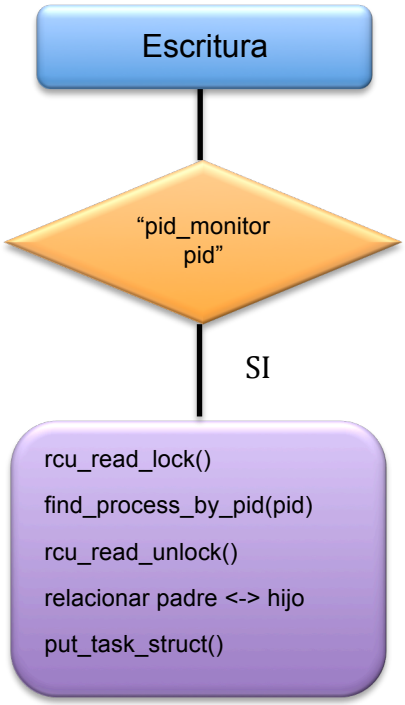
```
static int core2_monitor_pmcs_write(struct file *filp, const char __user
*buff, unsigned long len, void* data)
```

El uso de los parámetros ha sido descrito en la introducción de este capítulo.

La escritura establece una relación entre el proceso a monitorizar y el proceso que muestra los resultados, y así no perder la referencia. Para lograrlo, se llevarán a cabo las siguientes acciones:

1. El proceso de lectura de contadores escribe en el */proc:* "`pid_monitor pid`", donde `pid` va a ser el PID del proceso a monitorizar.
2. Se realiza un bloqueo mediante la llamada a `rcu_read_lock()`.

3. Durante este bloqueo se obtiene el `task_struct` del proceso a monitorizar por medio de su PID correspondiente llamando a la función `find_process_by_pid(pid)`.
4. Libera el cerrojo.
5. Mediante una variable `pr` de tipo `pmon_prof_t*` se relacionan respectivamente el proceso a monitorizar y el proceso de lectura.
6. Una vez realizada la relación podremos liberar la estructura anteriormente buscada mediante `put_task_struct()`, con el `task_struct` buscado como parámetro de entrada.



- **Lectura**: La función usada es la siguiente:

```
static int core2_monitor_pmcs_read(char* page, char** start, off_t
off,int count, int* eof, void* data)
```

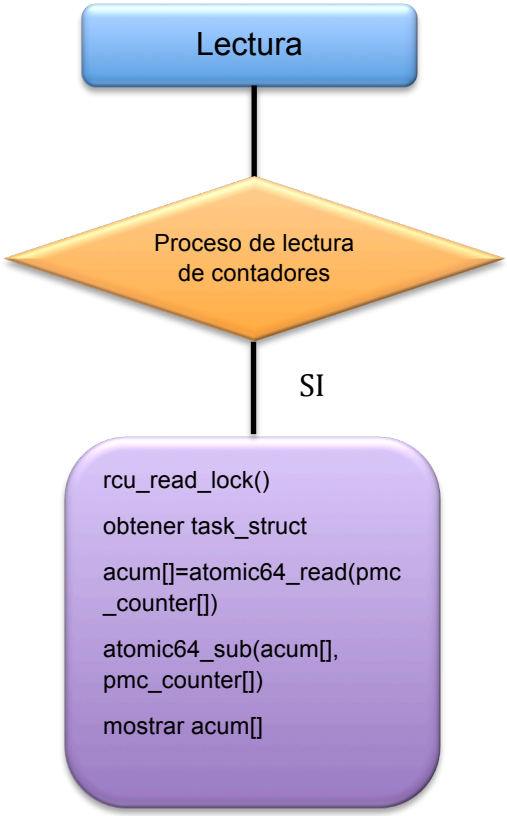
El uso de los parámetros ha sido descrito en la introducción de este capítulo.

La lectura va a ser utilizada para obtener el valor de los contadores. El funcionamiento consiste en realizar los siguientes pasos:

1. En caso de existir un proceso de lectura contadores, se produce un bloqueo mediante `rcu_read_lock()`.
2. Durante este bloqueo se obtiene el `task_struct` del proceso a monitorizar.

3. Se recorren los contadores leyéndolos para conocer sus valores con el uso de la función `atomic64_read()` (con un contador del proceso monitorizado como parámetro de entrada).
4. Almacena el valor del contador hardware en un acumulador. Este será mostrado posteriormente.
5. Resta al contador de proceso el acumulador. Así se consigue el valor adecuado al realizar la siguiente medición. La operación de resta se lleva a cabo mediante la función `atomic64_sub()` (con el acumulador y el contador del proceso como parámetros de entrada).
6. Se muestra el valor de los contadores a través de los datos almacenados en el acumulador.

Es necesario el uso de funciones que permitan seguir utilizando los contadores a la par que se realizan operaciones con ellos. Esta sincronización es posible gracias a las operaciones atómicas.



CAPÍTULO 5

HERRAMIENTA MULTIPLATAFORMA

Hemos desarrollado *pmctrack,* una herramienta multiplataforma programada en lenguaje C, que permite conocer el valor de los contadores hardware que éstos proporcionan en la medición de diferentes eventos.

La herramienta necesita comunicarse con el kernel para establecer el proceso a monitorizar, configurar el evento que queremos conocer y mostrar los valores que proporcionan los contadores. La conexión con el kernel se establece a través de las funciones situadas en modulo para la escritura y lectura de diferentes entradas del /proc.

Para poder conocer los valores que toman los contadores en la monitorización es necesario establecer una relación entre el proceso que va a mostrar el valor de los contadores y el proceso que se desea monitorizar. Para lograrlo el proceso que va a mostrar los contadores crea el proceso a monitorizar como hijo suyo.

En este capítulo se detalla la implementación de nuestra herramienta desarrollada *pmctrack,* que permitirá al usuario configurar los contadores hardware sobre cualquier arquitectura y monitorizar un programa deseado.

5.1 MANEJO

El modo de uso de *pmctrack* mediante línea de comando es el siguiente:

```
pmctrack [opciones] [programa a monitorizar [argumentos]]
```

- **Opciones:** *pmctrack* permite el uso de varias opciones para configurar el uso de la herramienta, no es necesario el uso de estas opciones ya que *pmctrack* tiene una configuración predefinida que detallaremos a lo largo de éste capítulo. Podemos establecer varias opciones a la vez. Según lo que el usuario desee puede elegir entre las siguientes opciones:
  - Ayuda (-h): muestra las instrucciones de uso de *pmctrack*.
  - Establecer salida (-o): permite guardar los resultados de las mediciones en un archivo especificado por el usuario. Se puede especificar la ruta del archivo. En caso de no existir este archivo, *pmctrack* lo crea. Si no se usa esta opción los resultados serán mostrados a través de la salida estándar para mostrar los resultados. Su eso es:

    ```
    -o archivo
    ```

  - Configurar los contadores (-*c*): permite establecer la configuración de los contadores variables, si no se usa esta opción *pmctrack* usará la última configuración establecida o una configuración predefinida en caso de que no se haya establecido ninguna configuración anteriormente. En el capitulo modo configuración entraremos en detalle sobre esta opción.
  - Tiempo entre mediciones (-T): permite elegir el tiempo (en segundos) entre cada medición realizada por los contadores. Si no se usa esta opción se usará el tiempo predefinido (1 segundo). Su uso es:

    ```
    -T segundos
    ```

  - Contadores a mostrar (-C): permite elegir que contadores mostrar. Los contadores a mostrar se establecen escribiendo "pic", seguido por el número de contador (sin espacio), para establecer varios contadores a mostrar cada pic_x debe estar separado por una coma (sin espacio). Se pueden establecer tantos *pic* como contadores existan. Si no se usa ésta opción mostrará todos los contadores. Su uso es:

    ```
    -C pic_x,...,pic_n
    ```

- **Programa a monitorizar:** Tras establecer las opciones deseadas se especifica el nombre del proceso a monitorizar y sus argumentos, en caso de que sean necesarios. Si únicamente se desea establecer la configuración de los contadores no es necesario poner ningún proceso a monitorizar. Entraremos en detalle en el apartado sobre el modo monitorización.

5.2 MODO MONITORIZACIÓN

La idea básica es conseguir que mediante la llamada `fork()`, el proceso hijo, una vez halla habilitado los contadores, pase a ejecutar el programa indicado por el usuario por medio de `execvp()`, mientras que el proceso padre se encargará de su monitorización. La ejecución de `execvp` simplemente realiza un cambio de ejecución de programa manteniendo el mismo descriptor de proceso. Esto es importante puesto que la manera que tiene el padre para monitorizar el programa indicado por el usuario es a través del descriptor del proceso hijo.

Al elegir este modo, se llama a la siguiente función:

```
void monitoring_counters(int seg,int npics,int optind,char *argv[])
```

Los parámetros de entrada representan los siguientes aspectos:

- **`int seg:`** Indica el tiempo (en segundos) que transcurrirá entre cada medición de los contadores.
- **`int npics:`** Número de contadores a mostrar por pantalla.
- **`int optind:`** Se utiliza para conocer la posición en `*argv[]` del nombre del proceso a monitorizar.
- **`char *argv[]:`** Todo los argumentos de entrada de *pmctrack.*

La función es llamada cuando en la ejecución de *pmctrack* se especifica el nombre de un proceso y sus parámetros de entrada, en caso de ser utilizados, como últimos argumentos en la línea de comandos.

El modo de monitorización consta de tres pasos principales: inicialización de las señales, el proceso hijo, y el proceso padre.

- **Inicialización de las señales:** como el proceso a monitorizar se ejecutará como hijo de *pmctrack*, necesitaremos conocer en todo momento el estado del proceso hijo y poder interrumpirlo si se desea. Para ello será necesario preparar los manejadores de señales. Vamos a utilizar las señales SIGINT, SIGCHLD y SIGALRM. SIGINT se usa para comprobar si se ha mandando una interrupción desde la terminal y así detener la medición de contadores y parar el proceso que se está monitorizando. SIGCHLD se utilizará para conocer el estado del proceso que estamos monitorizando, mediante esta señal podremos saber si el proceso

hijo a terminado su ejecución. Por último SIGALRM, se utiliza para llevar el reloj de alarma que, como veremos mas adelante, se usa en el proceso padre.

- **Proceso hijo:** El proceso hijo va a ser el programa que queremos monitorizar, para poder tener referencia a él desde *pmctrack* y así mostrar sus contadores, se realiza la llamada `fork()`. Una vez está en ejecución el proceso hijo este activa la monitorización de los contadores comunicándose con el módulo a través del */proc/mod_enable_pmcs* y tras esto con a través de `execvp()` ejecuta el proceso a monitorizar, el nombre del proceso a monitorizar y sus argumentos, en caso de que tenga, los sabemos gracias a `optind` y `*argv[]` que hemos pasado por parámetro
- **Proceso padre:** el proceso padre se va a encargar de mostrar el valor de los contadores, ya sea por la salida estándar o por el archivo donde el usuario puede haber proporcionado para volcar el valor de los contadores. Primero se comunica, a través del */proc/mod_monitor_pmcs,* con el módulo indicando el PID del proceso que va a monitorizar. Luego entra en un bucle donde se lee el valor de los contadores cada cierto numero de segundos indicados por la variable `seg` pasada por parámetro, para llevar a cabo esta medición cada cierto tiempo se usan las llamadas `alarm(seg)` seguido de `pause()`, para así parar el proceso de lectura de contadores `seg` segundos y que no ocurra ningún problema si se fuerza una interrupción. El bucle termina cuando el proceso hijo finalice, sea el motivo que sea.

5.3 MODO CONFIGURACIÓN

El modo configuración (opción `–c` de *pmctrack*) permite establecer una configuración deseada por el usuario para los contadores variables o configurables. El funcionamiento de la configuración consiste en una comunicación de *pmctrack* con el módulo a través del */proc* y así este poder establecer la configuración seleccionada por el usuario.

Podemos configurar los eventos a medir, proporcionando una máscara en caso de ser necesario; activar/desactivar modo usuario, activar/desactivar modo sistema operativo, activar/desactivar la detección de margen, invertir el resultado y especificar el contador de máscara.

- **Evento a medir:** es un código hexadecimal que determina el evento que va a ser medido
- **Máscara unitaria (UMASK):** valor hexadecimal complementario al evento que se utiliza para seleccionar una condición dentro del evento a medir.
- **Modo usuario (USR):** Con valor 1 activa la cuenta de contadores cuando el procesador está operando en modo privilegiado 1, 2 o 3. Con valor 0 está desactivado.

- **Modo sistema operativo (os):** Con valor 1 activa la cuenta de contadores cuando el procesador está operando en modo privilegiado 0. Con valor 0 está desactivado.
- **Detección de margen:** permite (cuando está activado, valor 1) la detección de margen de el evento seleccionado, es decir, permite al software, no sólo medir la fracción de tiempo gastado en un estado particular, sino también la duración media de permanencia en dicho estado.
- **Invertir (`INV`):** Cuando está activado (valor 1) invierte el resultado de la comparación que indica el campo Counter-Mask.
- **Contador de máscara (`CMASK`):** Cuando este campo es distinto de cero, el procesador lógico compara el valor de esta máscara con el recuento de situaciones detectadas por los eventos durante un solo ciclo. Si el recuento es mayor o igual, el contador se incrementa en uno. De lo contrario se queda igual.

  Si el campo es cero, entonces el contador se incrementa cada ciclo por causa del número de eventos asociados con múltiples ocurrencias.

La función llamada por *pmctrack* para la configuración de contadores es la siguiente:

```
void config_counters(char* arg)
```

El parámetro de entrada `arg` es la cadena de que el usuario ha introducido para establecer la configuración de los contadores.

La función es llamada cuando el usuario ha escrito la opción –c en la línea de comandos al ejecutar *pmctrack.*

La función `config_counters` es muy sencilla. Se comunica con el módulo a través de la escritura en */proc.* El módulo se encargará de otorgar la configuración establecida por el usuario a los contadores leyendo la información guardada en el citado */proc.* Además, divide la cadena de entrada en *tokens*, cada *token* se compondrá de un aspecto a configurar de un contador elegido (evento, umask…) y el valor otorgado por el usuario para ese aspecto.

La cadena `arg` (pasada por parámetro a la función) es dividida en *tokens* mediante el uso de la función `strtok`, la cual divide la cadena de entrada en cada ocurrencia de ",". Estas divisiones que indican los parámetros a configurar en los contadores son escritos en */proc/mod_pmc_config* para así transmitir los parámetros de configuración al módulo.

A continuación mostraremos las principales características en la configuración para las arquitecturas Intel® Core2™ Duo y AMD.

- **Configuración en Intel® Core2™ Duo**: En la arquitectura Intel® Core2™ Duo existen 5 contadores, de los cuales 3 son fijos y 2 son variables. Únicamente se podrán configurar los contadores variables, siendo estos los contadores 3 y 4.

  En caso de que el usuario no establezca ningún configuración para alguno de los contadores variables, estos medirán eventos predefinidos, los cuales son: Fallos de cache el contador 3; e instrucciones de carga el contador 4.

  De forma predefinida se encuentra activado el modo usuario y desactivados el resto de campos.

  La forma en la sintaxis para configurar el contador 3 y 4 es la siguiente:

  ```
  -c
  count3=hex,umask3=hex,usr3=0|1,os3=0|1,cmask3=hex,inv3=0|1,edge3=0|1,
  count4=hex,umask4=hex,usr4=0|1,os4=0|1,cmask4=hex,inv4=0|1,edge4=0|1.
  ```

  Donde $count_x$ indica el evento seleccionado para el contador *x*, *umaskx* indica la máscara unitaria para el contador *x*, y os_x, $cmask_x$, inv_x y $edge_x$ indica el estado de activación de los demás campos para el contador *x*.

  Recomendamos usar únicamente los campos para el evento y su máscara (en caso de no especificar una máscara se le asignará la máscara 0x00) y dejar el resto de campos sin modificar. A continuación mostramos un ejemplo completo y su sintaxis idéntica usando únicamente los campos necesarios y recomendados.

  Un ejemplo para una sintaxis completa es:

  ```
  -c
  count3=0xc0,umask3=0x00,usr3=1,os3=0,cmask3=0,inv3=0,edge3=0,count4=0
  xc4,umask4=0x01,usr4=1,os4=0,cmask4=0,inv4=0,edge4=0
  ```

  Recomendamos el uso de la siguiente sintaxis para un caso como el anterior:

  ```
  -c count3=0xc0,count4=0xc4,umask4=0x01
  ```

  En los apéndices se puede encontrar más información acerca de la arquitectura Core2 Duo y su configuración.

- **Configuración en AMD**: A diferencia de la arquitectura Intel® Core2™ Duo. AMD dispone de 4 contadores, pero en este caso todos son configurables.

  En caso de que el usuario no establezca ninguna configuración para alguno de los contadores variables, estos medirán unos eventos predefinidos, los cuales son: Instrucciones retiradas, en el contador 1; Ciclos de reloj, en el contador 2; Accesos a L3, en el contador 3; y Fallos de L3, en el contador 4.

  De forma predefinida se encuentran desactivados todos los demás campos.

  La forma en la sintaxis para configurar los contadores es la misma que para Intel® Core2™ Duo.

Recomendamos usar únicamente los campos para el evento y su máscara (en caso de no especificar una máscara se le asignará la máscara 0x00) y activar el modo usuario y dejar el resto de campos sin modificar. Los contadores se numeran del 0 al 3.

CAPÍTULO 6

RESULTADOS

En este capítulo se llevará a cabo una comparación de la herramienta cputrack de Solaris con *pmctrack* como demostración de su correcto funcionamiento.

Posteriormente, se escogerá una serie de benchmarcks caracterizados por su tipo de ejecución, pudiendo ser intensivos tanto en memoria como en CPU. Los resultados obtenidos mediante la monitorización con *pmctrack* de estos benchmarks se mostrarán en diferentes gráficas según la configuración de los contadores.

Finalmente, se recogerán los resultados obtenidos tras la ejecución en una arquitectura AMD para verificar la portabilidad en la que tanto se ha insistido.

6.1 VALIDACIÓN

Como muestra del correcto funcionamiento de la herramienta desarrollada, se compararán los resultados obtenidos de la ejecución del benchmark *gcc06* con las del *cputrack* de Solaris. Para ello se han seleccionado unos valores representativos como los ciclos por instrucción, fallos de la cache de nivel 2, mediciones de la TLB y otros ejemplos.

Para que los resultados ofrecidos sean consistentes entre los dos sistemas se ha utilizado el compilador gcc versión 4.4.6.

En el gráfico de barras mostrado, se pueden observar algunas medidas generadas por ambas herramientas. Al confrontar las soluciones comprobamos que los eventos analizados son similares. El coeficiente de correlación de las medidas tomadas con *pmctrack y cputrack* es de 99,898% con lo que se puede afirmar que ambas herramientas tienen comportamientos idénticos.

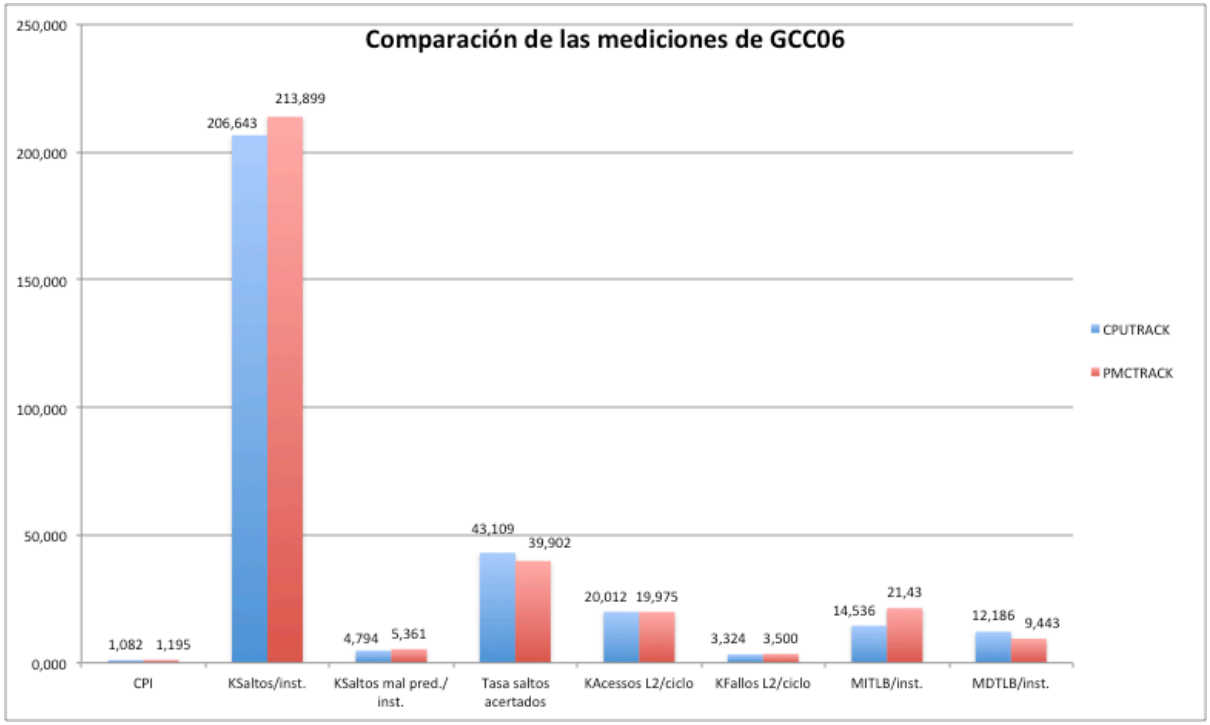


Figura 6.1.*Comparativa pmctrack (Solaris) y pmctrack*

6.2 BENCHMARKS

Se mostrará la solución de la ejecución de una serie de benchmarcks con nuestra herramienta en un procesador Intel Core 2 Duo, de esta manera podemos analizar las características de rendimiento de la CPU que causan estos programas.

El motivo de la elección de los benchmark de SPEC CPU2006 viene motivada por su amplia aceptación en la comunidad científica. Se han escogido un subconjunto de lo

más representativo posible tanto del grupo de CINT2006 como del de CFP2006. El primero contiene benchmarcks que miden y comparan el rendimiento informático intensivo entero. En particular hemos seleccionado:

- **astar06:** es el algoritmo A*. Deriva de una ruta 2D de búsqueda de biblioteca que se utiliza en algún juego. Esta biblioteca implementa tres algoritmos diferentes de búsqueda de la ruta. El primero es el conocido A* para mapas con tipos de terrenos transitables y no transitables. En segundo lugar, se encuentra una modificación de la trayectoria A* para encontrar mapas con diferentes tipos de terreno y la velocidad de movimiento diferente. El tercero es para gráficos. Está formado por las regiones del mapa con relación de vecindad.
- **gcc06:** se basa en la versión 3.2 del compilador GNU-GCC. Genera código para un procesador AMD Opteron. El benchmark se ejecuta como un compilador.
- **mcf06:** es un benchmark que se deriva de MCF, un programa usado para programación de vehículos de un solo depósito en el transporte público. El programa está escrito en C. La versión del benchmark utiliza casi exclusivamente aritmética de enteros.

Y el segundo, compara el rendimiento de computación en coma flotante. Los benchmark que analizaremos son los siguientes:

- **povray06:** POV-Ray es un *ray-tracing*. Esta técnica de procesamiento calcula una imagen de una escena simulando los rayos de luz. En el mundo real, son emitidos desde una fuente de luz y un objeto iluminado. La luz se refleja en los objetos o pasa a través de objetos transparentes.
- **soplex06:** esta basado en SoPlex Version 1.2.1. SoPlex resuelve un programa lineal usando el algoritmo Simplex.

  Este algoritmo resuelve problemas como el siguiente

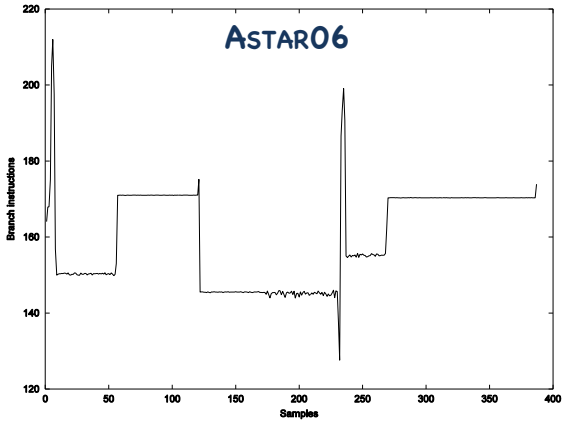
  $$\text{Minimize } c'x:$$
  $$\text{Sujeto a } Ax \leq b,\ x \geq 0.$$

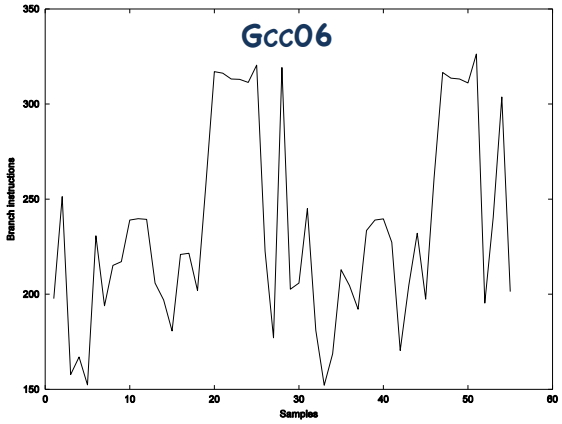
  En la práctica, x puede tener límites superiores y $A(i, .)\ X \leq b(i)$, las limitaciones también podrían ser mayor o igual a limitaciones o restricciones de igualdad.
- **sphinx06:** esta basado en un programa de reconocimiento de voz llamado sphinx-3. El código de este programa se incluye en este benchmark desde mediados de 2005.

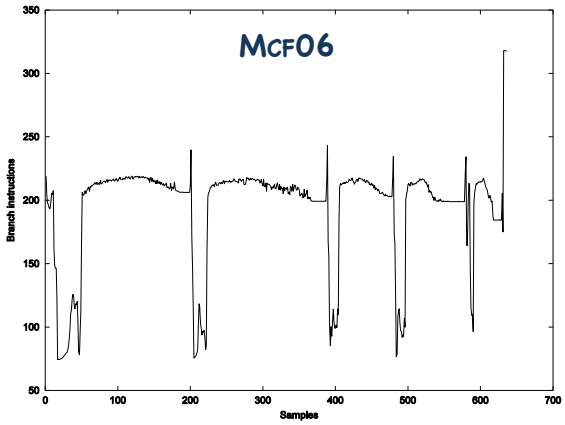
Realizamos tres tipos de pruebas utilizando los benchmark anteriores, mostrándose las gráficas obtenidas en las mediciones:

1. **Branches:** mide los saltos que se producen cuando se ejecutan estos programas. Hay dos tipos:
    a. Instrucciones de salto:

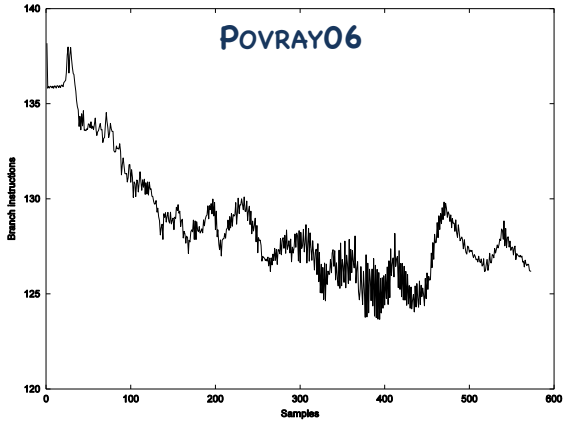
Medición en punto fijo

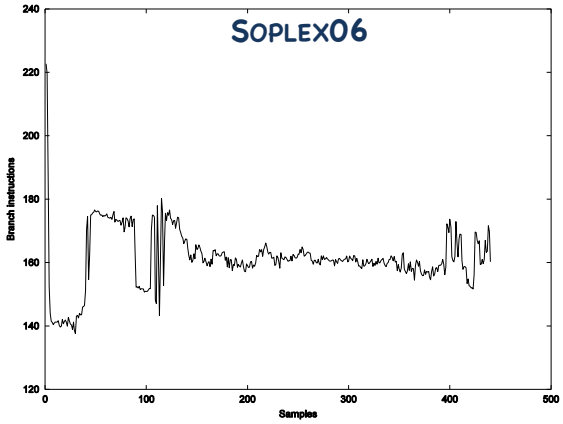


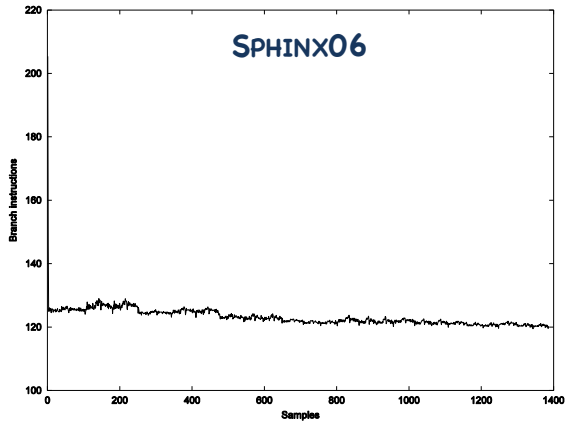




Medición en punto flotante

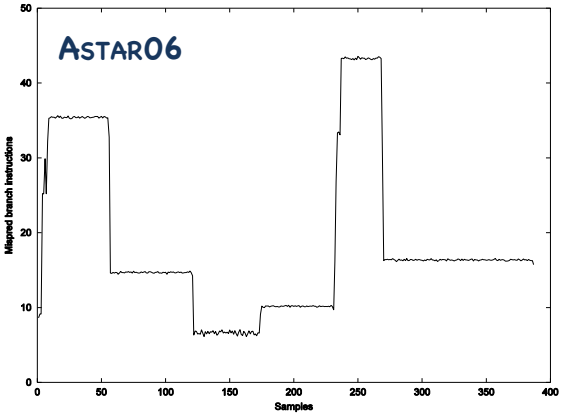


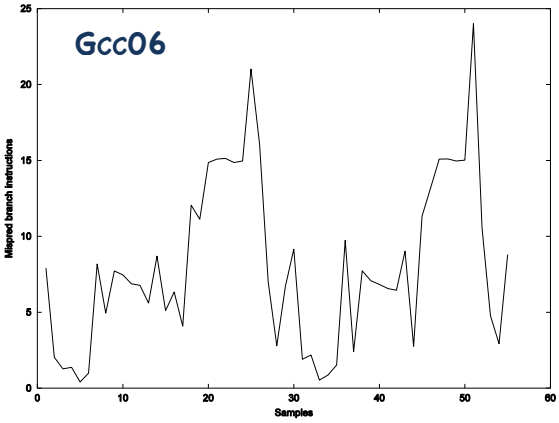


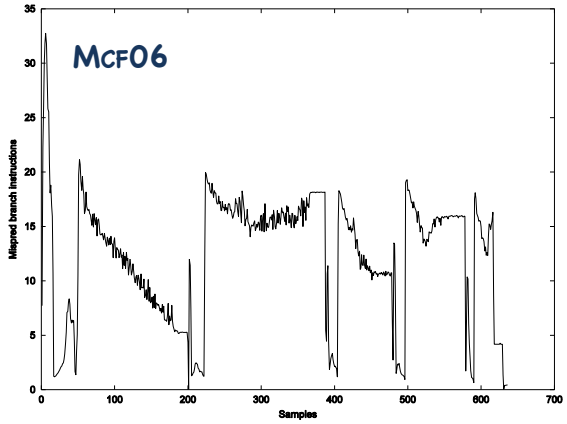


b. Fallo de predicción del salto:

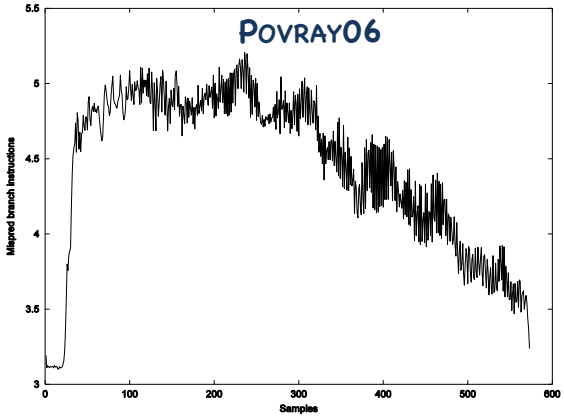
Medición en punto fijo

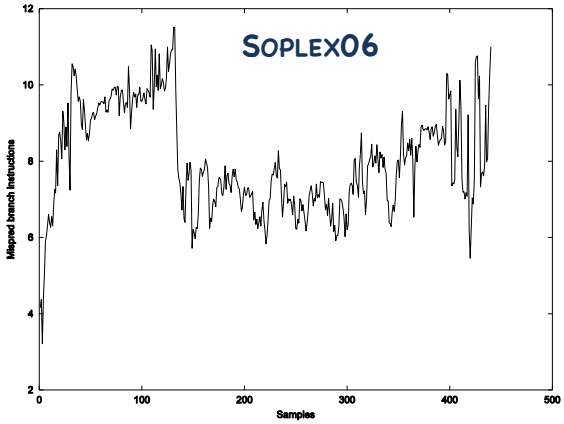


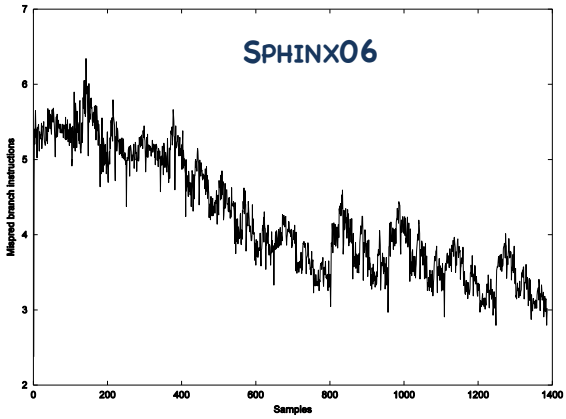




Medición en punto flotante





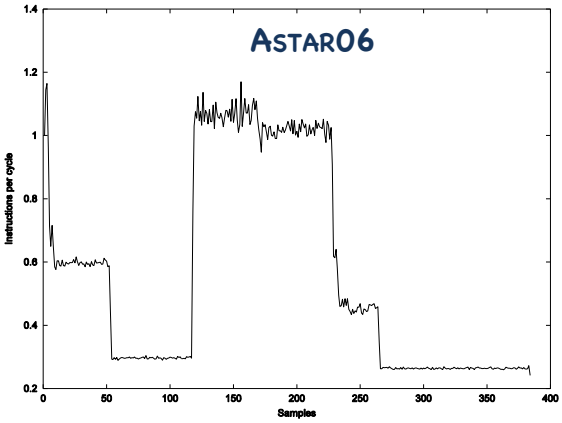


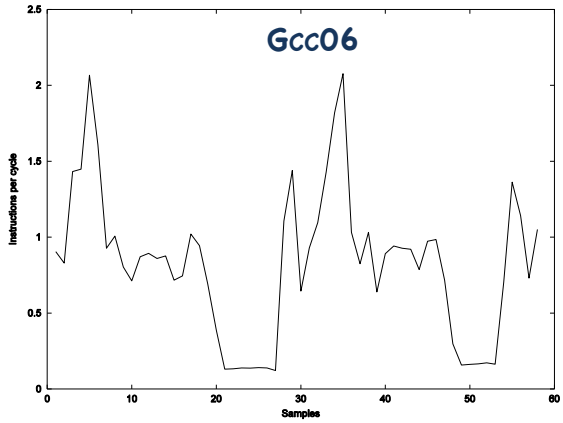
c. Summary: media de las mediciones de los 6 benchmarks probados

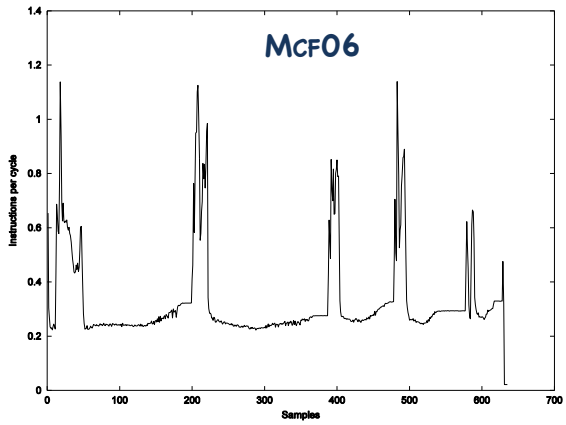
BENCH	SAMPLES	ITLB_MISSES_MINST	DTLB_MISSES_MINST
astar06	387	153,942146	16,422970
gcc06	55	213,898930	5,360540
mcf06	636	176,093584	10,420007
povray06	573	128,726780	4,438545
soplex06	440	161,631327	8,033076
sphinx306	1386	122,997639	4,193240

2. **IPC LLC MR:** mide los fallos, accesos, instrucciones por ciclo,… producidos en cache, vamos a analizar tres tipos:
    d. Instrucciones por ciclo:

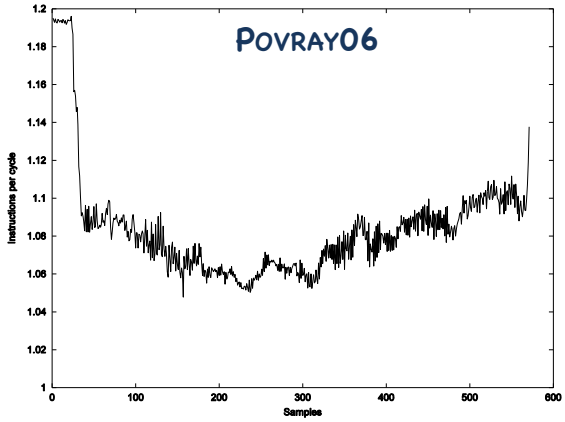
Medición en punto fijo

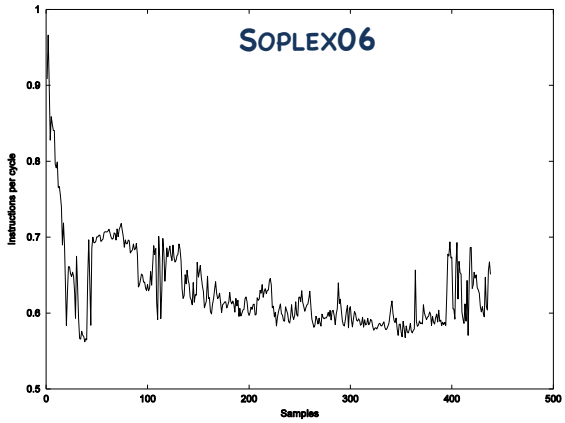


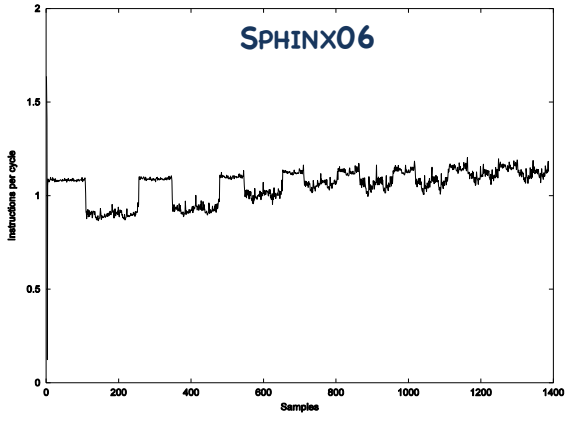




Medición en punto flotante

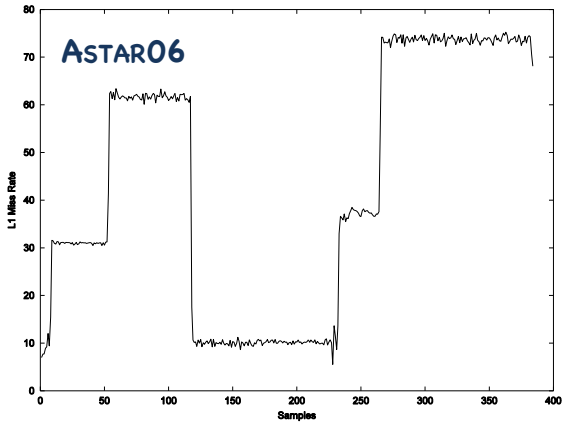


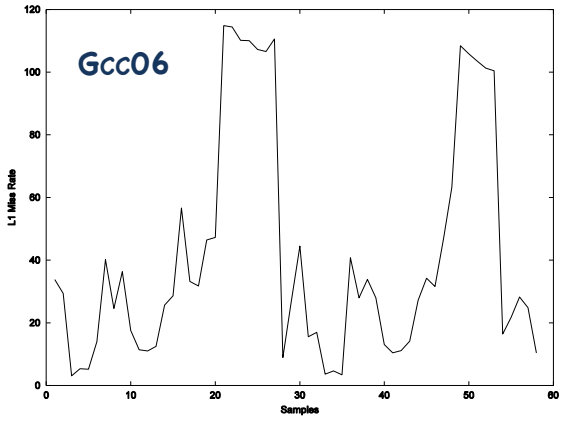


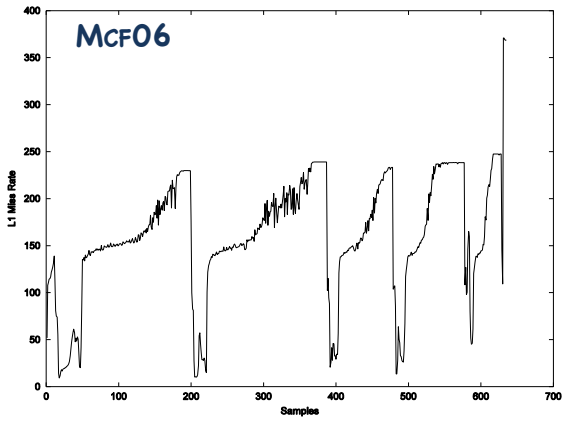


e. Fallos de cache L1:

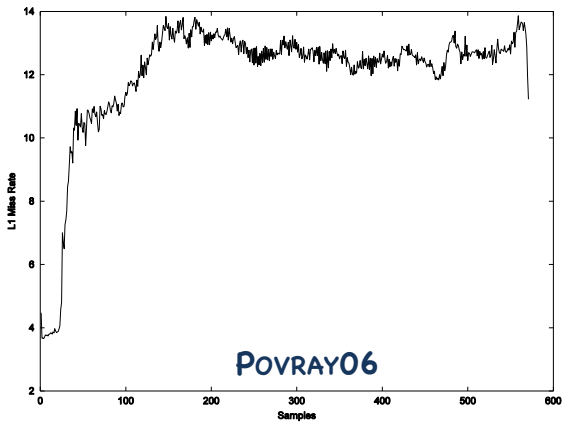
Medición de punto fijo

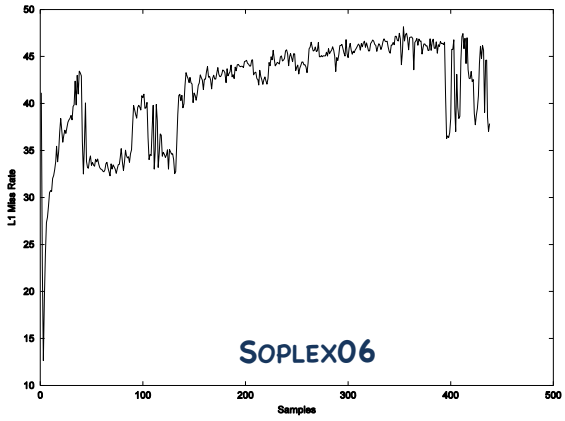


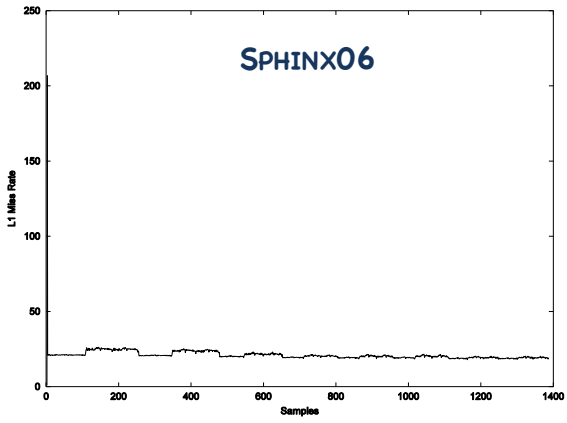




Medición de punto flotante

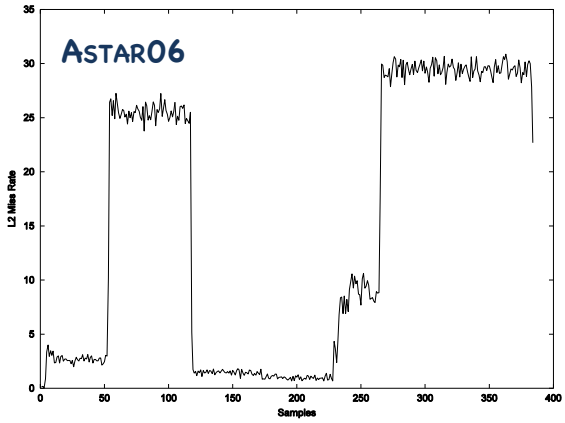


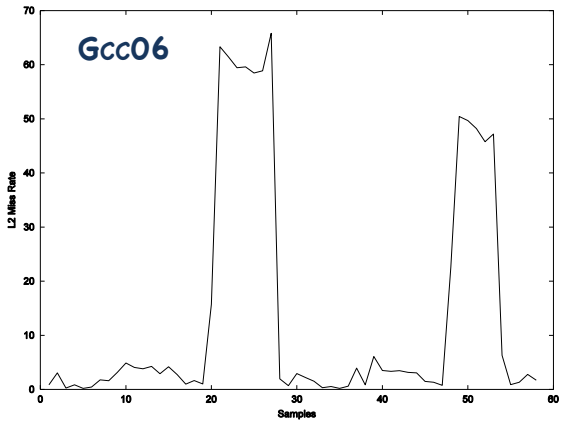


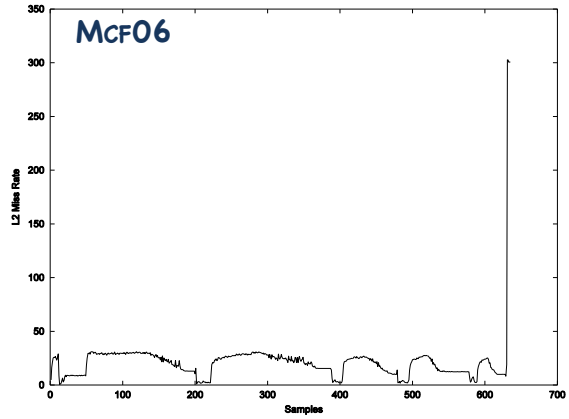


f. Fallos de cache L2:

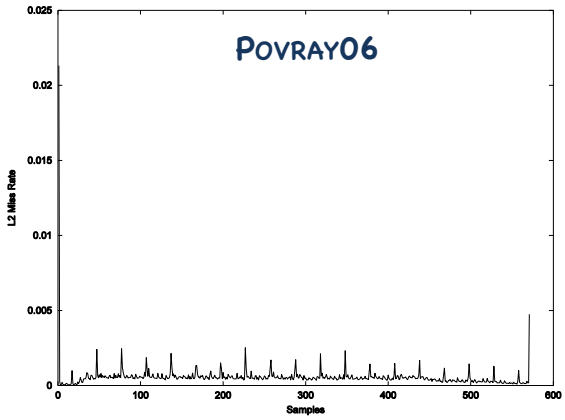
Medición en punto fijo

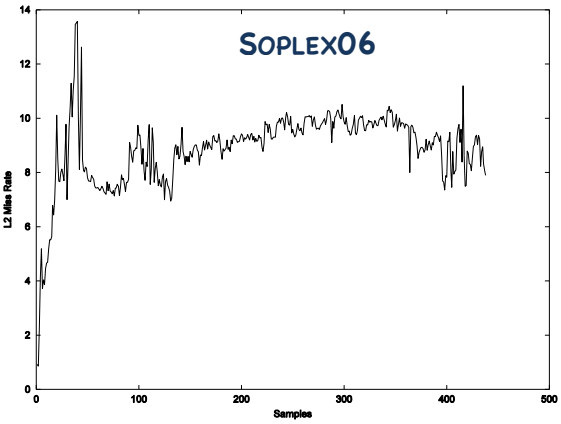


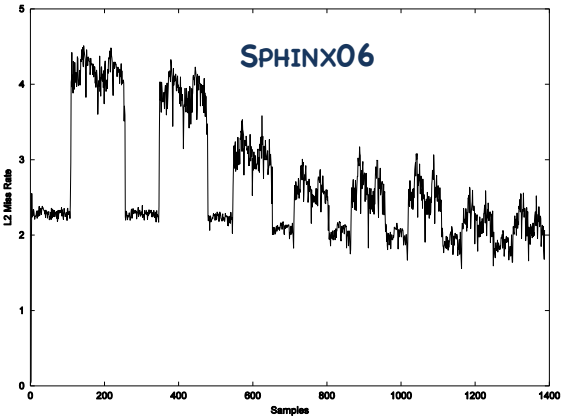




Medición en punto flotante





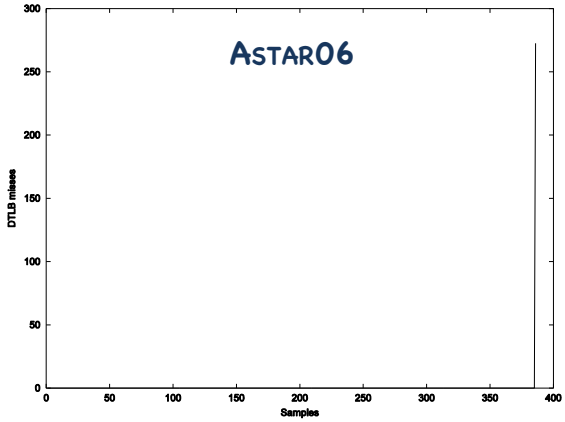


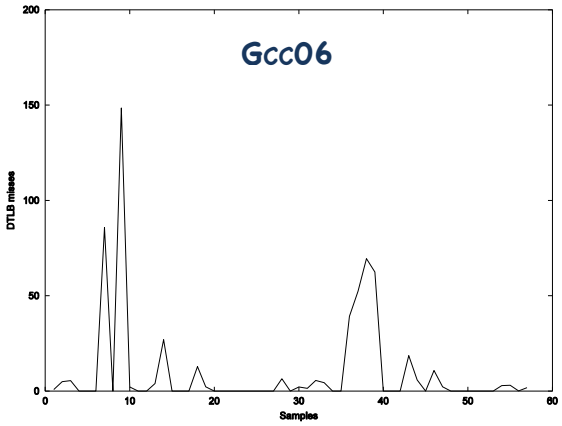
g. Summary: mostramos la media de todos los fallos, accesos, etc. que se producen en la cache.

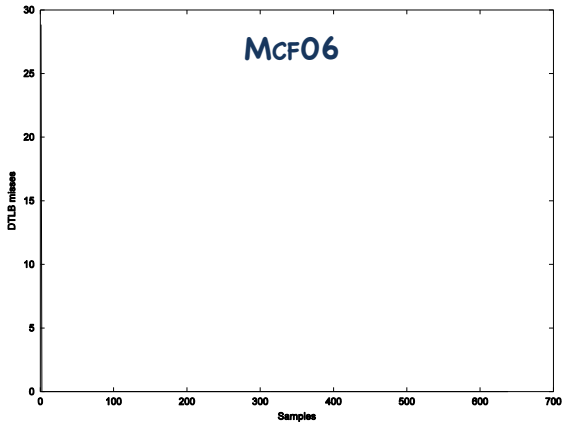
BENCH	SAMPLES	LLC_MISS_KINST	LLC_REQUEST_KINST	LLC_MISS_KCYCLE	LLC_REQUEST_KCYCLE	LLC_MISS_RATE	INSTR_PER_CYCLE
astar06	384	8,277220	28,510953	4,657918	16,044237	29,031719	0,562739
gcc06	58	4,300929	24,546433	3,499931	19,974946	17,521606	0,813762
mcf06	635	16,261347	137,261592	5,324650	44,945232	11,846975	0,327442
povray06	571	0,000519	11,992802	0,000562	12,994329	0,004326	1,083511
soplex06	438	8,776558	41,100785	5,531040	25,901965	21,353748	0,630206
sphinx306	1387	2,615457	20,786852	2,762053	21,951952	12,582266	1,056050

3. **TLB:** medimos los accesos a esta caché, que utiliza hardware de gestión de memoria para mejorar la velocidad de traducción de direcciones virtuales. Analizamos la información tanto de la TLB de instrucciones, ITLB como la de datos, DTLB:
    h. DTLB:

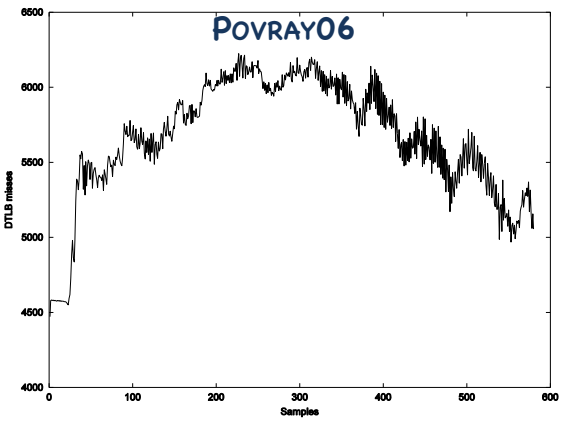
Medición en punto fijo

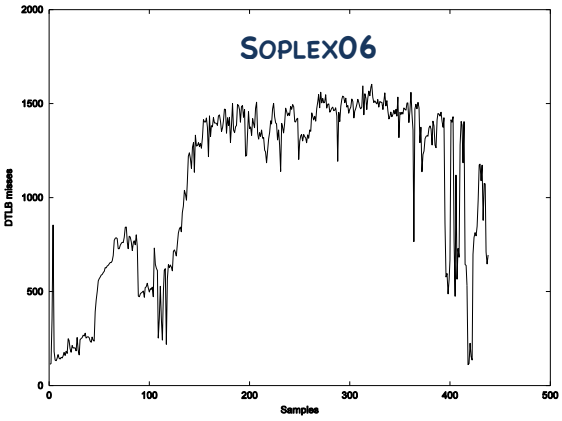


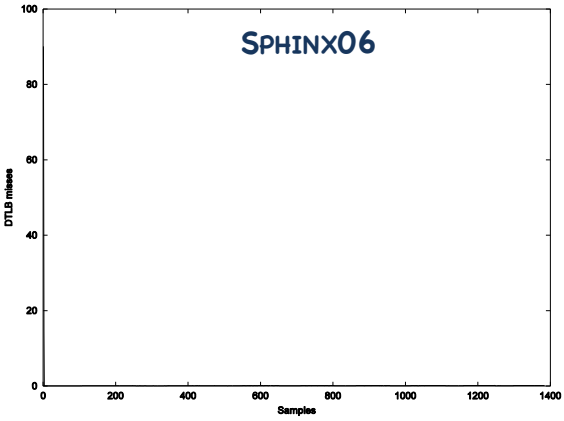




Medición en punto flotante

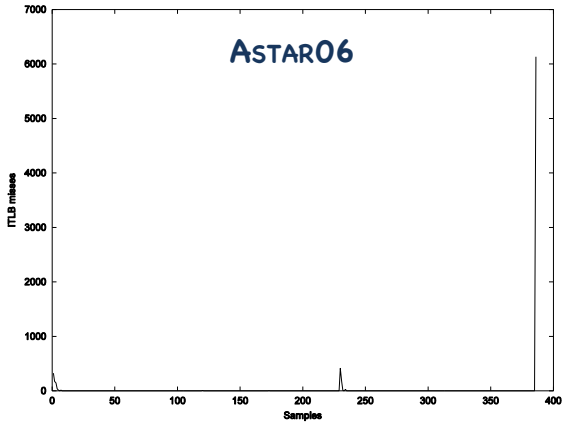


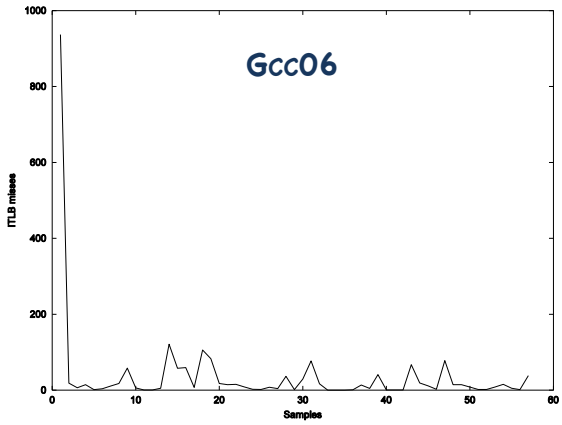


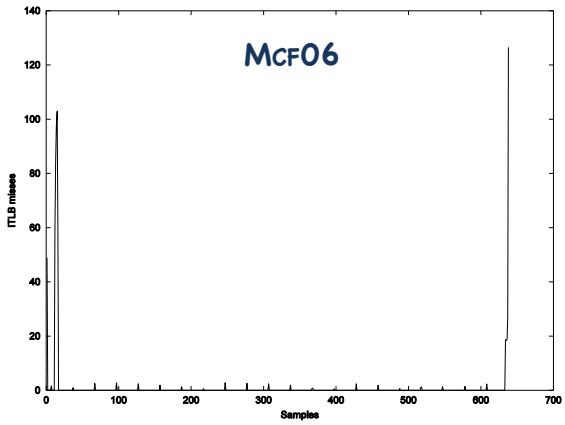


i. ITLB:

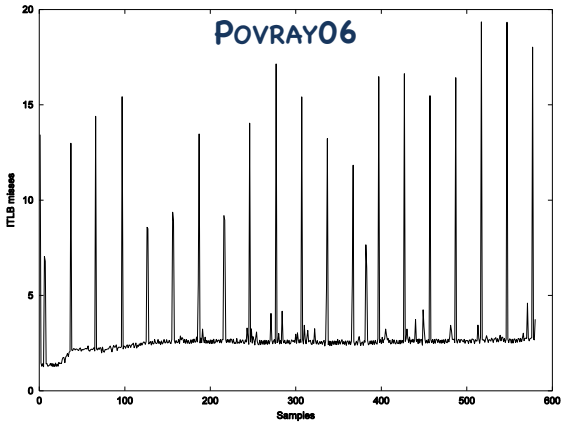
Medición en punto fijo

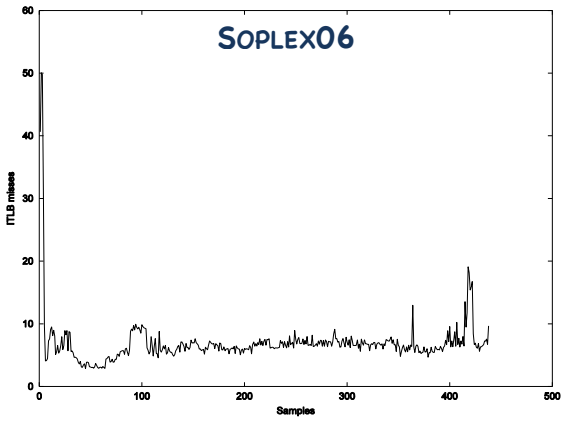


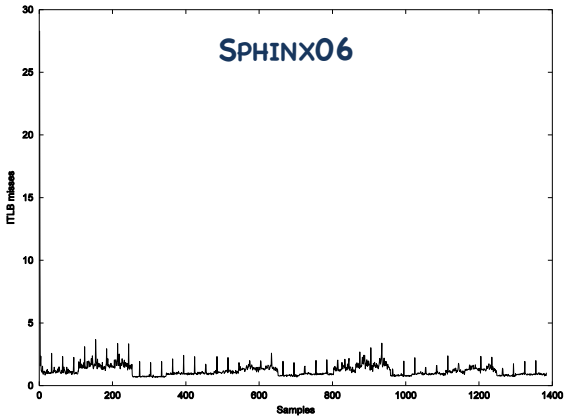




Medición en punto flotante







j. Summary: en este apartado podemos observar las medias que se producen de los seis benchmarks en la TLB:

BENCH	SAMPLES	ITLB_MISSES_MINST	DTLB_MISSES_MINST
astar06	386	4,073250	0,000050
gcc06	57	30,430064	9,443385
mcf06	638	0,757076	0,082404
povray06	580	3,008963	5661,689310
soplex06	438	6,955815	1043,257368
sphinx306	1384	1,194738	0,197447

6.3 EXTENSIÓN DE LA HERRAMIENTA EN AMD

A continuación mostramos un ejemplo de nuestro programa, *pmctrack* ejecutado en una arquitectura AMD, hacemos la medición de la aplicación *burnP6*, con la configuración de los contadores predeterminada. AMD contiene cuatro contadores cada uno de ellos mide determinadas características el contador 1 (*pic0*) calcula las instrucciones retiradas, el contador 2 (*pic1*) los ciclos, el contador 3 (*pic2*) los accesos a L2 y el contador 4 (*pic3*) fallos de cache L2.

nsample	tid	event	pic0	pic1	pic2	pic3
1	1	tick	2183788351	2028242430	12	1
2	1	tick	2326966865	2150329413	23	13
3	1	tick	2380448351	2204670142	6	3
4	1	tick	2380437317	2204431803	1	0
5	1	tick	2379821912	2202796705	10	1
6	1	tick	2380329395	2203938369	8	1
7	1	tick	2386994056	2204434113	0	0
8	1	tick	2380432298	2204352073	2	0
9	1	tick	2380653533	2204878430	0	0
10	1	tick	2380334944	2204368124	3	0
11	1	tick	2380556625	2204871571	0	0
12	1	tick	2380384123	2204345474	1	0
13	1	tick	2380485539	2204707604	0	0
14	1	tick	2380429921	2204484318	1	0
15	1	tick	2380483416	2204697821	0	0
16	1	tick	2380444385	2204339214	1	0
17	1	tick	2380596342	2204728638	0	0
18	1	tick	2380336165	2204114176	2	0
19	1	tick	2386842244	2204448774	2	1
20	1	tick	2380246187	2204295249	5	2
21	1	tick	2380560411	2204807673	0	0
22	1	tick	2380389678	2204512834	2	0
23	1	tick	2380460360	2204759546	0	0

CAPÍTULO 7

CONCLUSIONES Y APORTACIONES

Como producto final del proyecto de sistemas informáticos hemos desarrollado una herramienta de monitorización denominada *pmctrack* que hace uso de los contadores hardware existentes en la mayoría de los procesadores modernos. Dicha herramienta destaca sobre las demás existentes en el mercado por su portabilidad ante diferentes arquitecturas, versiones de kernel y sistemas operativos

Entre las principales características de la herramienta *pmctrack* destacamos:

1.- **Diseño modular:** que permite aislar las dependencias con la plataforma objetivo, el sistema operativo o la versión del kernel. La mayor parte de codificación recae en módulos del kernel lo que facilita el desarrollo, la depuración y portabilidad.
2.- **Es multiplataforma:** en la actualidad, *pmctrack* permite la monitorización de cualquier programa utilizando los contadores hardware integrados en el procesador en sistemas tales como Intel® Core™ 2 Duo, Atom™, Xeon® y AMD.
3.- **Es fiable:** se ha verificado su correcto funcionamiento comparando el rendimiento de un programa con la herramienta de SUN *cputrack* observándose una tasa de correlación de 99.898%.
4.- **Se actualiza fácilmente:** el diseño modular indicado anteriormente permite la creación de nuevos módulos para arquitecturas con especificaciones nuevas, lo que lo hace altamente interesante desde el punto de vista de dar soporte a plataformas objetivo futuras.

Por otro lado nos gustaría indicar que el diseño de la herramienta desarrollada permite hacer uso de los contadores hardware en otros ámbitos que no sean únicamente el modo usuario. Es decir, podemos resaltar la facilidad que existe a día de hoy de dotar de nuevas funcionalidades al planificador del sistema operativo que abre la posibilidad de incorporar nuevas heurísticas de migración de tareas en función del rendimiento de la carga de trabajo en tiempo real, políticas de planificación del *runtime* en función de las fases observadas de una aplicación, etc.

Por último queríamos dejar constancia de algunas de las dificultades encontradas en este proyecto. Entre otras podemos citar la dificultad en la comprensión del funcionamiento del kernel de Linux y discernir sobre que partes eran candidatas a

su modificación, la depuración de los módulos que en muchas ocasiones era una tarea ardua por la falta de herramientas de depuración a bajo nivel; y la compresión de los contadores hardware de la CPU, su uso y puesta en práctica a través de *cputrack*.

APÉNDICES

I. VERSIÓN 2.6.38 DE LINUX

Linux Torvalds anunció el lanzamiento del Linux Kernel 2.6.38, ofreciendo cambios en el *Virtual File System* y un parche para la "agrupación automática de procesos" que mejora el rendimiento. Incluye el soporte para procesadores AMD Fusion, GPUs AMD y NVIDIA, y drivers adicionales para chips *Wi-Fi*. Además mejora el sistema de archivos *Btrfs* y soporte a "transparent huge page" para acelerar las aplicaciones de base de datos y virtualización.

En noviembre del 2009, un parche de 233 líneas del desarrollador Mike Galbrait demostró que podría acelerar notablemente la experiencia del escritorio en Linux, reduciendo la latencia hasta en 60 veces. En esta versión de Linux, el parche cambia la forma en que el planificador asigna tiempo de CPU a cada proceso para que el sistema pueda agrupar todos los procesos con el mismo identificador de sesión.

Por otro lado, los cambios en el sistema virtual de archivos (*VFS*) no solo hacen que las cargas de trabajo *multi-threading* sean escalables, sino que también hace que algunas cargas con procesos únicos sean mucho más rápidos. En esencia, el *dcache VFS* (directorio de caché) y los mecanismo de búsqueda de ruta se han revisado a fin de ser más escalables.

La memoria permitida para procesar aumentará de tamaño de 4KB a 2 MB gracias a las "Transparent Huge Pages", que reducen el número de asignaciones de memoria y aprovechan el mayor rendimiento del hardware. Según Tim Burke, vicepresidente de Ingeniería Linux de Red Hat, el impacto de la inclusión de las *THP* en Linux 2.6.38 es que ofrecen un mejor desempeño en las cargas de trabajo que requiere una gran cantidad de memoria, tales como servidores de *JVM* y base de datos.

ADICIONES IMPORTANTES A LINUX 2.6.38:

- **Compresión LZO** y *snapshots* de sólo lectura en *Btrfs*: *Btrfs* añade soporte para el algoritmo de compresión LZO, como alternativa a *zlib*. También se añade soporte para marcar un *snapshot* como sólo lectura y la característica *"force mounting"* que hará que el código base sea más tolerante a fallos.
- **Transparent huge pages:** Esta alternativa al API basado en el sistema de archivos aprovecha de las ventajas de rendimiento de los procesadores modernos con mayor cache. El código se utiliza por defecto cuando sea aplicable, pero puede ser configurado para ser utilizado siempre o sólo cuando se solicite. El aumento será especialmente notable con el incremento en cargas de trabajo intensivas de datos, como bases de datos o sesiones KVM, que con frecuencia accedan a direcciones virtuales.

- **Protocolo de malla B.A.T.M.A.N.:** *Better Approach To Mobile Ad-hoc Networking* es un protocolo de ruteo proactivo para Redes *Mesh Ad-hoc* Inalámbricas, incluyendo las redes *ad-hoc* móviles (en inglés MANETs). El protocolo mantiene proactivamente información sobre la existencia de todos los nodos en la malla, que son accesibles a través de enlaces de comunicación de uno o múltiples saltos. La estrategia de *B.A.T.M.A.N.* es determinar para cada destino en la malla un vecino de un salto, el cual puede ser utilizado como mejor Gateway para comunicarse con el nodo de destino. En estas redes, cada nodo de enrutamiento participa en los datos de reenvío para otros nodos de forma dinámica y se dice que es útil para situaciones de emergencia como desastres naturales.
- **Soporte para AMD *Fusion:*** Esta versión incluye el soporte para los nuevos *APUs Fusión* basados en una o más instancias del núcleo "*Bobcat*".
- **Límites de memoria sucia:** Esta función controla los límites de *Dirty Pages* (Buffer de páginas que contienen modificaciones que no se han escrito en el disco) de cada controlador de memoria *cgroups*.

Linux 2.6.38 también ofrece una variedad habitual variedad de ajustes, adiciones y correcciones de errores, incluyendo mejoras en el núcleo, el planificador (scheduler), gestión de memoria, manejo de bloques, sistemas de archivos, creación de redes, criptografía, virtualización, seguridad y trazado.

II. CONTADORES HW DE RENDIMIENTO EN INTEL®

Los contadores HW fueron introducidos por Intel® en los procesadores Pentium para la monitorización del rendimiento. Son contadores específicos denominados MSR ocupan un área despreciable dentro del chip. Estos contadores permiten seleccionar los parámetros de rendimiento del procesador para ser monitoreados y medidos. La información obtenida puede ser usada para ajustar el sistema y observar el rendimiento.

El mecanismo de monitorización del rendimiento definido para los procesadores Pentium e Intel® Xeon™ no es de tipo arquitectónico. Por otro lado, las familias de procesadores actuales (Intel® Core™, Atom™, familia Nethalem™) tienen dos clases de capacidades para la monitorización de rendimiento:

- La primera, soporta eventos usando conteo. Estos eventos son no-arquitectónicos y varían de un modelo de procesador a otro.
- La segunda clase admite el mismo sistema de conteo y toma de muestras de usos. Es de tipo arquitectónico y su comportamiento es consistente por medio de implementaciones del procesador.

Existen varias versiones de monitorización dependiendo de cuál sea la arquitectura del procesador.

- Intel® Core™ Solo y Duo -> Versión 1
- Intel® Core™ 2 Duo y siguientes -> Versión 2
- Intel® Atom™ -> Versiones 2 y 3 (Incluyendo 2 contadores de rendimiento de propósito general: IA32_PMC0 y IA32_PMC1)
- Intel® i7 -> Versiones 2 y 3 (Incluyendo 4 contadores de rendimiento de propósito general: IA32_PMC0 - IA32_PMC3)

Versión 1:

La configuración de los eventos de monitorización de rendimiento arquitectónicos consiste en programar los registros de selección de eventos de rendimiento (*MSRs IA32_PERFEVTSELx*). El resultado de un evento queda almacenado en un contador (*MSR IA32_PMCx*). Cada contador está emparejado con su registro correspondiente.

La monitorización del rendimiento arquitectónico proporciona un mecanismo CPUID para enumerar la siguiente información:

- El número de contadores disponibles en un procesador lógico.
- El número de bits soportados por cada contador *IA32_PMCx.*
- El número de eventos soportados en un procesador lógico.

El software puede usar CPUID para descubrir si se dispone o no de la monitorización del rendimiento. En la implementación inicial, el software puede determinar cuántos pares de *MSRs IA32_PERFEVTSELx/IA32_PMCx* son soportados por cada *core* (ancho de bits del PCM) y el número de eventos disponibles.

La instalación de la monitorización de rendimiento incluye un conjunto de contadores y de registros de selección de eventos. Estos MSRs tienen las siguientes propiedades:

- Los contadores *IA32_PMCx* comienzan en la dirección 0C1H y ocupa un bloque contiguo igual al espacio de dirección del MSR.
- Los registros de selección de eventos *IA32_PERFEVTSELx* empiezan en la dirección 186H y ocupa un bloque contiguo igual al espacio de dirección del MSR. Cada registro está emparejado con su correspondiente contador en el bloque de dirección 0C1H.
- El ancho de bits de un contador MSR *IA32_PMCx* se define usando el *CPUID.0AH:EAX[23:16]*. Estos son el número de bits válidos para una operación de lectura. En operaciones de escritura, los 32 bits menos significativos del MSR deben estar escritos con algún valor, y los más significativos representan la extensión de signo desde el valor del bit 31.

  La disposición del campo de bits de los registros *IA32_PERFEVTSELx* está definida arquitecturalmente. En la siguiente figura vemos como está diseñado:

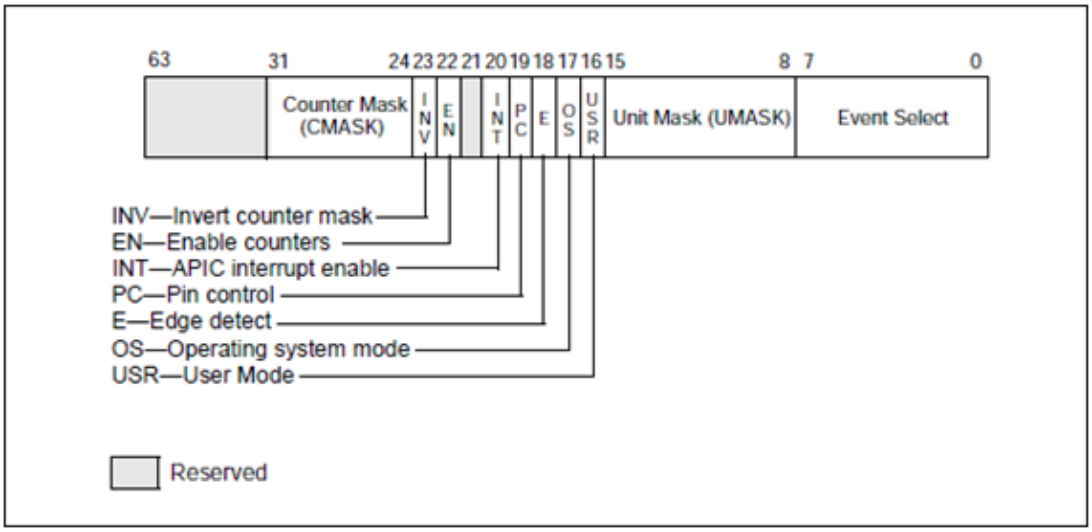


Figura II.1. *Disposición de MSRs IA32_PERFEVTSELx.*

- **Campo Event Select -** Selecciona la unidad lógica de eventos usada para detectar las condiciones de la micro-arquitectura. El conjunto de valores para este campo está definido arquitectónicamente; cada valor corresponde a una unidad lógica de eventos. El número de eventos arquitectónicos se consulta con CPUID.0AH: EAX. Un procesador puede soportar sólo un subconjunto de valores predefinidos.
- **Campo Unit Mask (UMASK) –** Estos bits habilitan la condición que la unidad lógica del evento seleccionado detecta. Los valores UMASK válidos para cada

unidad lógica son específicos. Para cada evento su correspondiente valor UMASK define un estado micro-arquitectónico específico.

Un estado micro-arquitectónico predefinido asociado con un evento arquitectónico puede no ser aplicable a un procesador dado. El procesador soporta un solo subconjunto de eventos arquitectónicos predefinidos.

- **Indicador USR (user mode) –** Especifica que sólo se cuenta el estado micro-arquitectónico seleccionado cuando el procesador lógico está operando como modo privilegiado 1, 2 o 3. Este indicador puede ser usado con el indicador OS.
- **Indicador OS (operating system mode) –** Especifica que sólo se cuenta el estado micro-arquitectónico seleccionado cuando el procesador lógico está operando como modo privilegiado 0. Este indicador puede ser usado con el indicador USR.
- **Indicador E (edge detect) –** Permite (cuando está activado) la detección de bordes de un estado micro-arquitectónico seleccionado. El procesador lógico cuenta el número de transiciones correctas o incorrectas para cualquier estado que pueda ser expresado por los otros campos.

  Este mecanismo permite al software, no sólo medir la fracción de tiempo gastado en un estado particular, sino también la duración media de permanencia en dicho estado (por ejemplo, el tiempo usado en esperar a que una interrupción sea atendida).
- **Indicador PC (pin control) –** Cuando se establece y se produce un evento, el procesador lógico cambia los pines PMi e incrementa el contador. Si no, cambia los pines si se produce un desbordamiento del contador.
- **Indicador INT (APIC interrupt enable) –** Cuando está activado, el procesador lógico genera una excepción a través de su APIC local en caso de desbordamiento del contador.
- **Indicador EN (enable counters) –** Si está activo, el conteo del rendimiento está habilitado en el contador de monitorización de rendimiento correspondiente. En caso contrario, el contador estará deshabilitado. La unidad lógica de eventos para un UMASK debe ser deshabilitada (indicador EN a '0') antes de escribir en el *IA32_PMCx.*
- **Indicador INV (invert) –** Cuando está activado invierte el resultado de la comparación que indica el campo Counter-Mask.
- **Campo Counter-Mask (CMASK) –** Cuando este campo es distinto de cero, el procesador lógico compara el valor de esta máscara con el recuento de situaciones detectadas por los eventos durante un solo ciclo. Si el recuento es mayor o igual, el contador se incrementa en uno. De lo contrario se queda igual.

  Si el campo es cero, entonces el contador se incrementa cada ciclo por causa del número de eventos asociados con múltiples ocurrencias (por ejemplo, 2 o más instrucciones retiradas por ciclo; o el número de ocupaciones en la cola del bus).

Versión 2:

Esta versión tiene varias características mejoradas de la anterior, y son las siguientes:

- **Registro de contador de rendimiento de función fija y registro de control asociado –** Tres de los eventos de rendimiento arquitectónicos se cuentan usando tres MSRs de función fija *(IA32_FIXED_CTR0, IA32_FIXED_CTR0 y IA32_FIXED_CTR2)*. Cada uno de los PMC de función fija pueden contar un solo evento de rendimiento arquitectónico.

  La configuración de estos PMCs se realiza escribiendo en los campos de bits correspondientes del MSR (*IA32_FIXED_CTR_CTRL – Figura 2*) situado en la dirección 38DH. A diferencia de la configuración de los eventos de rendimiento de propósito general PMCs (*IA32_PMCx*) a través del campo UMASK (*IA32_PERFEVTSELx*), para configurar o programar *IA32_FIXED_CTR_CTRL* para los PMCs de función fija no se requiere ningún tipo de campo UMASK.

- **Simplificación de la programación de eventos** – La operación más frecuente en la programación de eventos de rendimiento son la activación/desactivación de eventos de conteo y la verificación de los estados de desbordamiento de los contadores. Esta segunda versión ofrece tres arquitecturas MSR:

  a. ***IA32_PERF_GLOBAL_CTRL*** – Da permiso al software para activar o desactivar eventos de conteo para todos o cualquier combinación de PMCs de función fija (*IA32_FIXED_CTRx*) o cualquier PMCs de propósito general a través de un único WRMSR.

  b. ***IA32_PERF_GLOBAL_STATUS*** – Da permiso al software para consultar las condiciones de desbordamiento del contador en cualquier combinación de PMCs de función fija o PMCs de propósito general mediante un único RDMSR.

  c. ***IA32_PERF_GLOBAL_OVF_CTRL*** – Da permiso al software para eliminar condiciones de desbordamiento del contador en cualquier combinación de PMCs de función fija o PMCs de propósito general por medio de un único WRMSR.

La instalación proporcionada por la versión 2 de monitorización del rendimiento se puede consultar en la hoja de 0AH del CPUID examinando el contenido del registro EDX (*CPUID.0AH.EDX*).

- Los bits del 0 al 4 del registro *CPUID.0AH.EDX* indican el número de contadores de rendimiento de función fija disponibles por *core*.
- Los bits del 5 al 12 del registro *CPUID.0AH.EDX* indican el ancho de bits de los contadores de rendimiento de función fija. Los bits de la anchura de estos contadores están reservados y deben ser ceros.

El MSR *IA32_FIXED_CTR_CTRL* incluye varios campos de 4 bits. Cada uno de ellos controla el funcionamiento de un contador de rendimiento de funciones fijas como muestra la siguiente figura.

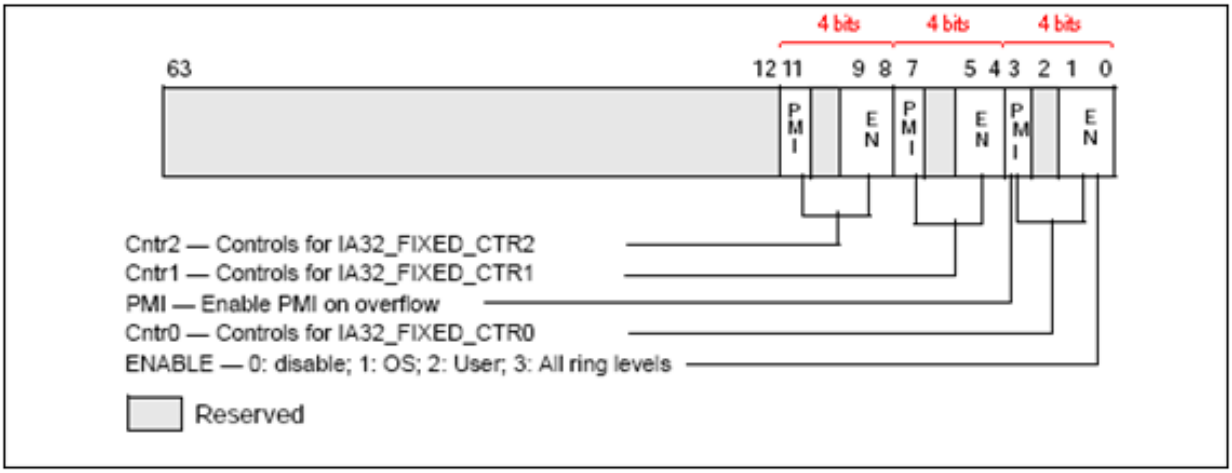


Figura II.2. *Disposición de MSR IA32_FIXED_CTR_CTRL.*

- **Campo Enable:** Cuando el bit 0 o el bit 1 están activos, el rendimiento del conteo está habilitado en la correspondiente función fija del rendimiento de los contadores. Esto es para incrementarse mientras la condición asociada con el evento de rendimiento de la arquitectura ocurrió en el anillo 0 en caso de que se active el bit 0 y en el anillo mayor que 0 si se activa el bit 1. Si escribimos en los dos bits un 0 el contador de rendimiento se para. Escribiendo el valor 11B se activa el contador para incrementarse independientemente de los niveles de privilegio.
- **Campo PMI:** Cuando se habilita, el procesador lógico genera una excepción a través de su APIC local en condición de desbordamiento del respectivo contador fijo.

MSR IA32_PERF_GLOBAL_CTRL proporciona un control de **single-bit** para habilitar los contadores por cada contador de rendimiento. La figura II.3 muestra la disposición de IA32_PERF_GLOBAL_CTRL.

Cada bit activo en IA32_PERF_GLOBAL_CTRL es AND'ed con el bit habilitado para todo nivel de privilegio en el MSRs IA32_PERFEVTSELx o IA32_PERF_FIXED_CTR_CTRL respectivamente para empezar/parar la cuenta de los contadores. El conteo se activa si los resultados AND'ed son verdaderos; el conteo se desactiva cuando el resultado es falso.

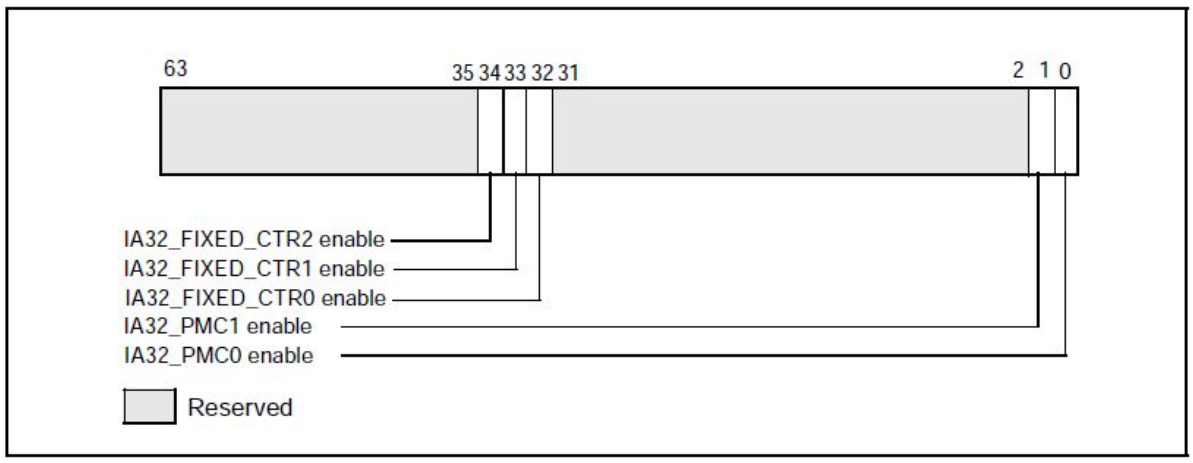


Figura II.3. *Disposición del MSR IA32_PERF_GLOBAL_CTRL.*

Los contadores de rendimiento en función fija soportados por la arquitectura de la versión 2 esta listado en la tabla II.5., la vinculación entre cada contador de rendimiento de función fija para un evento de la arquitectura también se muestra.

MSR IA32_PERF_GLOBAL_STATUS ofrece un único bit de estado para el software para consultar la situación de desbordamiento de cada contador de rendimiento. El MSR también proporciona un bit de estado adicional para indicar las condiciones de desbordamiento cuando los contadores son programados para *precise-event-based sampling* (PEBS). MSR IA32_PERF_GLOBAL_STATUS también da un *sticky bit* para indicar los cambios del estado de la supervisión del rendimiento de hardware. La figura II.4 muestra la disposición del IA32_PERF_GLOBAL_STATUS. Un valor de 1 en los bits 0,1 y del 32 al 34 indica una condición de desbordamiento que ha ocurrido en el contador asociado.

Cuando un contador de rendimiento se configura para PEBS, la condición de desbordamiento en el contador genera una interrupción en el monitoreo del rendimiento señalizando un evento PEBS. En un evento PEBS, los lugares de procesador de datos de registros en la zona del buffer se borra el estado de desbordamiento del contador, y se establece el bit OvfBuffer en IA32_PERF_GLOBAL_STATUS.

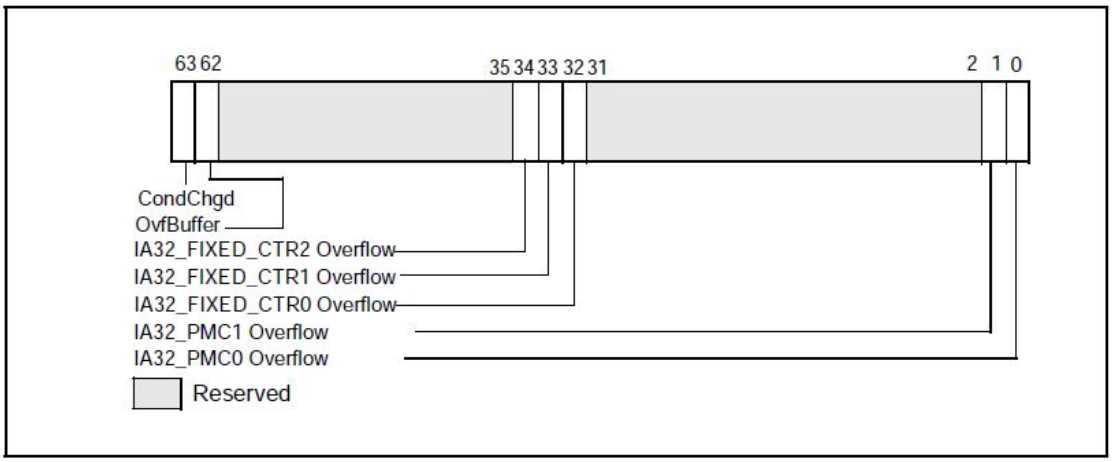


Figura II.4. *Disposición del MSR IA32_PERF_GLOBAL_STATUS.*

MSR IA32_PERF_GLOBAL_OVF_CTL permite al software limpiar el/los indicador/es de desbordamiento de los contadores de propósito general o de función fija a través de una sola WRMSR. El software debería limpiar la indicación de desbordamiento cuando:

- La creación de nuevos valores en el caso de seleccionar y/o en el campo de UMASK para contar o toma de muestras.
- Recarga los valores del contador para continuar con el muestreo.
- Desactiva eventos de conteo o muestreo.

La disposición del IA32_PERF_GLOBAL_OVF_CTL se muestra en la figura II.5:

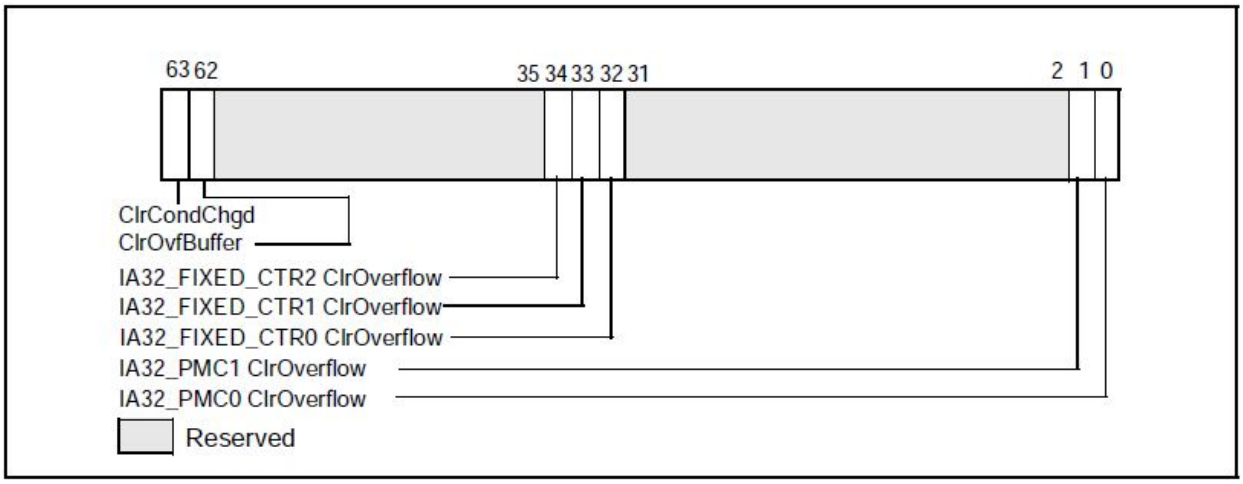


***Figura II.5**. Disposición del MSR IA32_GLOBAL_OVF_CTR.*

Versión 3:

Las comodidades proporcionadas por la arquitectura de monitorización de rendimiento en las versiones 1 y 2 son también apoyadas por la arquitectura de la versión 3. Además la versión 3 ofrece mejoras para soportar un *core* del procesador que comprende más de un procesador lógico, es decir, un *core* de procesador compatible con la tecnología Intel® Hyper-Threading Technology o la capacidad simultanea de *multi-threading*.

No profundizaremos en esta versión con detalle puesto que no hemos implementado nada relacionada con la tecnología *multi-threading.* Esta es una de las posibles mejoras del proyecto en trabajos futuros.

1.1. Monitorización del rendimiento de procesadores INTEL® CORE™ DUO.

En los procesadores Intel® Core™ Solo e Intel® Core™ Duo, los eventos de monitorización del rendimiento no arquitectónico son programados usando las misma herramientas (ver la Figura II.1) que en los eventos arquitectónicos de rendimiento.

Los eventos de rendimiento no arquitectónicos utilizan los valores de eventos seleccionados que son modelos específicos. Los valores de la máscara de eventos (Umask) también son específicos para las unidades lógicas de eventos. Algunas condiciones de micro-arquitectura detectadas por un valor Umask pueden tener una relación distinta con la topología del procesador. Como resultado, el campo de la unidad de la máscara (por ejemplo, IA32_PERFEVTSERLx[bits 15:8]) debe contener sub-celdas en las que se especifica la información de los cores del procesador.

La disposición de sub-celdas sin el campo Umask debe soportar dos bits de codificación que califica la relación entre la condición y el *core* originario. Este dato se muestra en la tabla II.1. La decodificación de dos bits para la especificación del *core* es solo apoyado por un subconjunto de valores de Umask y por el procesador Intel® Core™ Duo. Cada evento se refiere a un evento especifico del *core*.

IA32_PERFEVTSELx MSRs	
Bit 15:14 Encoding	Description
11B	All cores
10B	Reserved
01B	This core
00B	Reserved

Tabla II.1. *Codificación del Core específico con un Umask no arquitectónico.*

Algunas condiciones de micro-arquitectura permiten la detección específica solo en el límite del procesador físico. Algunos eventos en el bus pertenecen a esta categoría, proporcionan una especificidad entre un procesador de origen físico y otro agente en el bus. La decodificación de una sub-celda para especificar el agente, se muestra en la tabla II.2.

IA32_PERFEVTSELx MSRs	
Bit 13 Encoding	Description
0	This agent
1	Include all agents

Tabla II.2. *Codificación del Agente específico con un Umask no arquitectónico.*

Algunas condiciones de micro-arquitectura permiten la detección típica que incluye o excluye la acción de *prefetch* del hardware. Una codificación de los dos bits debe ser soportado por la calificación de la acción del hardware. Normalmente, esto se aplica solo a algunos eventos L2 o bus. La codificación del sub-campo para la clasificación de *prefetch* del hardware se muestra en la tabla II.3.

IA32_PERFEVTSELx MSRs	
Bit 13:12 Encoding	Description
11B	All inclusive
10B	Reserved
01B	Hardware prefetch only
00B	Exclude hardware prefetch

Tabla II.3. *Codificación de la calificación del prefetch del HW con Umask no arquitectónico.*

Algunos eventos de rendimiento pueden:

a) No apoyar a ninguno de las tres codificaciones de calificación de eventos específicos.
b) Soportar un core y un agente específico simultáneamente.
c) Apoyar una cualificación simultanea del core-especifico y el *prefetch* del hardware.

Un agente específico y una calificación de captación del hardware son mutuamente excluyentes.

Además, algunos eventos L2 permiten calificación que distingue a los estados coherentes de cache. La definición de un sub-campo para un estado de calificación coherente de cache se muestra en la tabla II.4. Si no hay bits en el sub-campo de calificación MESI entonces se establece un evento que requiere el establecimiento de los bits en el sub-campo de calificación MESI, el recuento de evento no se incrementará.

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	Description
Bit 11	Counts modified state
Bit 10	Counts exclusive state

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	Description
Bit 9	Counts shared state
Bit 8	Counts Invalid state

Tabla II.4. *Definición de la calificación MESI con Umask no arquitectónico.*

1.1.1. Contadores de rendimiento en función fija.

Los procesadores basados en micro-arquitectura Intel® Core™ proporcionan tres contadores de rendimiento en función fija. Los bits más allá del ancho del contador fijo están reservados y deben escribirse como ceros. El modelo específico de los contadores de rendimiento en función fija en los procesadores que soportan la Arquitectura Perfmon en la versión 1 ocupa 40 bits de ancho.

Cada contador en función fija está dedicado para contar un evento de monitorización de rendimiento predefinida. Los eventos de monitorización de rendimiento asociados con contadores de función fija y las direcciones de estos contadores están visibles en la tabla II.5.

Event Name	Fixed-Function PMC	PMC Address
INST_RETIRED.ANY	MSR_PERF_FIXED_CTR0/IA32_FIXED_CTR0	309H
CPU_CLK_UNHALTED.CORE	MSR_PERF_FIXED_CTR1//IA32_FIXED_CTR1	30AH
CPU_CLK_UNHALTED.REF	MSR_PERF_FIXED_CTR2//IA32_FIXED_CTR2	30BH

Tabla II.5. *Asociación del rendimiento de los contadores en función fija con eventos de rendimiento de la arquitectura.*

1. **Campo Enable:** Cuando el 0 se activa, el conteo de rendimiento se habilita en el correspondiente contador de rendimiento en función fija para incrementarlo

cuando la condición objetivo asociada con el evento de rendimiento de la arquitectura ocurre, se pone en el anillo 0.

Cuando el bit 1 se establece, el conteo de rendimiento se activa en el correspondiente contador de rendimiento en función fija para incrementar cuando la condición objetivo asociada con el evento de rendimiento de la arquitectura se produce en el anillo superior a 0.

Escribiendo 0 en ambos bits se para el contador de rendimiento. Escribiendo 11B hace que el contador se incremente independientemente de los niveles de privilegio.

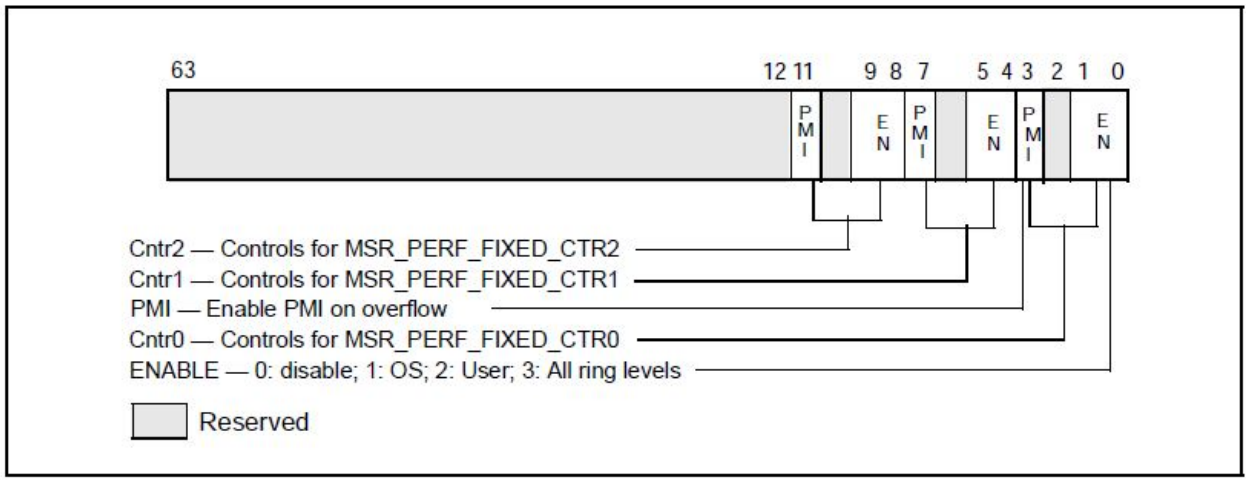


***Figura II.6**. Disposición del MSR MSR_PERF_FIXED_CTR_CTRL.*

2. **Campo PMI:** Cuando se establece, el procesador lógico genera una excepción a través de su APIC local en condición de desbordamiento del respectivo contador de función fija.

1.1.2. Programación para eventos Non-Retirement

Los pasos básicos para programar los contadores de rendimiento y para los eventos de conteo son los siguientes:

1. Seleccionar el evento o eventos que se quieren medir.
2. Para cada evento, seleccionar un ESCR compatible con esos eventos comprobando las restricciones.
3. Seleccionar el contador CCCR donde se va a contar a partir de los valores de ESCR.
4. Establecer los niveles de privilegios que se van a monitorizar en el ESCR.
5. Activar los contadores CCCR de rendimiento para seleccionar el ESCR y el evento del filtro.
6. Establecer el conteo en cascada si se desea.

7. Opcionalmente configurar el CCCR para generar interrupciones PMI cuando el contador desborda. El APIC local debe estar preparado.
8. Comenzar el conteo del evento.

Para todos estos pasos se puede detallar como conseguir los valores adecuados. Obtener los valores con este método es algo tedioso, con lo que se buscó otra forma para obtener estos valores.

1.1.3.Eventos at-retirement

Muchos eventos de rendimiento no arquitectónicos se ven afectados por la naturaleza especulativa de ejecución fuera de orden. Un subconjunto de eventos de rendimiento no arquitectónicos en procesadores basados en micro-arquitectura Intel® Core™ están mejorados con un mecanismo de etiquetado que incluye las aportaciones que se derivan de la ejecución especulativa. Los eventos retirados están activados en procesadores basados en una micro-arquitectura Intel® Core™ que no requiere programar un control especial de MSR, pero está limitado para IA32_PMC0. Ver la tabla II.6 para la lista de eventos disponibles del procesador basado en micro-arquitectura Intel® Core™.

Event Name	UMask	Event Select
ITLB_MISS_RETIRED	00H	C9H
MEM_LOAD_RETIRED.L1D_MISS	01H	CBH
MEM_LOAD_RETIRED.L1D_LINE_MISS	02H	CBH
MEM_LOAD_RETIRED.L2_MISS	04H	CBH
MEM_LOAD_RETIRED.L2_LINE_MISS	08H	CBH
MEM_LOAD_RETIRED.DTLB_MISS	10H	CBH

***Tabla II.6**. Eventos de rendimiento at-retirement para una micro-arquitectura Intel® Core™.*

1.1.4. Precise Event-Based Sampling (PEBS)

Los procesadores basados en micro-arquitectura Intel® Core™ también soportan el muestreo de eventos precisos (Precise Event Based Sampling, PEBS). Esta característica fue introducida por procesadores basados en micro-arquitectura Intel® NetBurst.

PEBS utiliza un mecanismo de depuración y una alarma de supervisión del rendimiento para almacenar un conjunto arquitectónico de la información de estado del procesador. La información proporcionada por el estado arquitectónico de la instrucción ejecutada después de la que causó el evento.

En los casos donde la misma instrucción causa BTS y PEBS para ser activada, PEBS es procesado antes que BTS. El PMI solicita mantenerse hasta que el procesador completa el tratamiento de PEBS y BTS.

Para procesadores basados en micro-arquitectura Intel® Core™, los eventos que soportan muestreo simple son listados en la tabla II.7.

Event Name	UMask	Event Select
INSTR_RETIRED.ANY_P	00H	C0H
X87_OPS_RETIRED.ANY	FEH	C1H
BR_INST_RETIRED.MISPRED	00H	C5H
SIMD_INST_RETIRED.ANY	1FH	C7H
MEM_LOAD_RETIRED.L1D_MISS	01H	CBH
MEM_LOAD_RETIRED.L1D_LINE_MISS	02H	CBH
MEM_LOAD_RETIRED.L2_MISS	04H	CBH
MEM_LOAD_RETIRED.L2_LINE_MISS	08H	CBH
MEM_LOAD_RETIRED.DTLB_MISS	10H	CBH

Tabla II.7. *Eventos de rendimiento PEBS para Intel® Core™ micro-arquitectura.*

Configuración del buffer PEBS:

Para procesadores basados en micro-arquitectura Intel® Core™, PEBS está disponible usando solo IA32_PMC0. Utilizando el procedimiento de seguimiento para establecer el procesador y el contador IA32_PMC0 para PEBS:

1. Establece las utilidades para eventos precisos del buffer. Los valores en la base, el índice, el máximo absoluto, el umbral de interrupción, y el reinicio de celdas de eventos precisos del contador del manejo del buffer DS. En los procesadores basados en la micro-arquitectura Intel® Core™, los registros PEBS consisten en unas entradas de direcciones de 64-bit.
2. Activación del PEBS. Ajusta el campo Enable del PEBS en la etiqueta PMC0 (bit 0) en MSR IA32_PEBS_ENABLE.
3. Configura el contador de rendimiento IA32_PMC0 e IA32_PERFEVTSEL0 para un evento listado en la tabla II.7.

Formato de grabación de PEBS:

Los registros de formato PEBS deben extenderse a través de diferentes implementaciones de procesadores. El MSR IA32_PERF_CAPABILITES define un mecanismo de software para manejar la evolución del formato de los registros PEBS en el procesador que soporta la arquitectura de monitorización del rendimiento, con la versión de ID igual o superior a 2. Los campos de bits relevantes que gobiernan el PEBS son:

- *PEBSTrap [bit 6]:* cuando se establece, el registro PEBS esta como atrapado. Después el contador con PEBS-activo tiene sobrecarga, el registro se graba para el siguiente evento de PEBS-disponible para la terminación del muestreo de las instrucciones causando un evento PEBS. Cuando se limpia, la grabación del PEBS

es como un error. El registro PEBS graba antes del muestreo de la instrucción causando un evento PEBS.

- *PEBSSaveArchRegs [bit 7]:* cuando se active, PEBS se guardara en un registro de arquitectura y un estado de información conforme la codificación del valor del campo PEBSRecordFormat. Un procesador basado en micro-arquitectura Intel® Core, este bit siempre estará a 1.
- *PEBSRecordFormat [bits 11:8]:* la decodificación válida es la siguiente:
    - 0000B: solo registros de propósito general, puntero a instrucción y registros RFLAGS son guardado en cada registro PEBS.

Escribir una Interrupción en la Rutina de Servicio del PEBS:

Las utilidades PEBS comparten el mismo vector y la rutina de servicio de interrupción (llamado el DS ISR) con el muestreo de eventos no precisos basados en el DS ISR.

La rutina de servicio puede preguntar por MSR_PERF_GLOBAL_STATUS para determinar que el/los contador/es causan la condición de desbordamiento. La rutina de servicio debería limpiar el indicador de desbordamiento para escribir en MSR_PERF_GLOBAL_OVF_CTL.

Una comparación de la secuencia de requerimiento del programa PEBS para procesadores basados en la micro-arquitectura Intel® Core™ e Intel® NetBurst listado en la tabla II.8.

	For Processors based on Intel Core microarchitecture	For Processors based on Intel NetBurst microarchitecture
Verify PEBS support of processor/OS	▪ IA32_MISC_ENABLE.EMON_AVAILABE (bit 7) is set. ▪ IA32_MISC_ENABLE.PEBS_UNAVAILABE (bit 12) is clear.	
Ensure counters are in disabled	On initial set up or changing event configurations, write MSR_PERF_GLOBAL_CTRL MSR (0x38F) with 0. On subsequent entries: ▪ Clear all counters if "Counter Freeze on PMI" is not enabled. ▪ If IA32_DebugCTL.Freeze is enabled, counters are automatically disabled. Counters MUST be stopped before writing.[1]	Optional
Disable PEBS.	Clear ENABLE PMC0 bit in IA32_PEBS_ENABLE MSR (0x3F1).	Optional
Check overflow conditions.	Check MSR_PERF_GLOBAL_STATUS MSR (0x 38E) handle any overflow conditions.	Check OVF flag of each CCCR for overflow condition
Clear overflow status.	Clear MSR_PERF_GLOBAL_STATUS MSR (0x 38E) using IA32_PERF_GLOBAL_OVF_CTRL MSR (0x390).	Clear OVF flag of each CCCR.
Write "sample-after" values.	Configure the counter(s) with the sample after value.	
Configure specific counter configuration MSR.	▪ Set local enable bit 22 - 1. ▪ Do NOT set local counter PMI/INT bit, bit 20 - 0. ▪ Event programmed must be PEBS capable.	▪ Set appropriate OVF_PMI bits - 1. ▪ Only CCCR for MSR_IQ_COUNTER4 support PEBS.
Allocate buffer for PEBS states.	Allocate a buffer in memory for the precise information.	
Program the IA32_DS_AREA MSR.	Program the IA32_DS_AREA MSR.	
Configure the PEBS buffer management records.	Configure the PEBS buffer management records in the DS buffer management area.	
Configure/Enable PEBS.	Set Enable PMC0 bit in IA32_PEBS_ENABLE MSR (0x3F1).	Configure MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT and MSR_PEBS_MATRIX_HORZ as needed.
Enable counters.	Set Enable bits in MSR_PERF_GLOBAL_CTRL MSR (0x38F).	Set each CCCR enable bit 12 - 1.

Tabla II.8. *Requisitos para el programa PEBS.*

Reconfiguración de las utilidades de PEBS:

Cuando el software necesita la reconfiguración de las utilidades del PEBS, debería permitir un periodo de reposo entre detener el conteo previo del evento y la creación de un nuevo evento PEBS. El periodo inactivo es permitir que cualquier registro residual del PEBS para completar la captura de sus especificaciones previas de la dirección del buffer (proporcione para IA32_DS_AREA).

III. CONTADORES HW DE RENDIMIENTO EN AMD

Reset: xxxx xxxx xxxx xxxxh. PERF_CTL[3:0] se utiliza para especificar los eventos contados por el [The Performance Event Counter Registers (PERF_CTR[3:0])] MSRC001_00[07:04] y para controlar otros aspectos de sus operaciones. A cada contador de rendimiento le corresponde un registro event-select que controla su operación.

El modo *edge* incrementa el contador cuando ocurre una transición en el evento monitorizado. Si el evento seleccionado cambia sin deshabilitar el contador, un *edge* extra es detectado falsamente cuando el primer evento es un cero y el segundo es un uno. Para evitar la detección de un *edge* falso, se inhabilita el contador cuando ha cambiado el evento y entonces esta activo el contador con escritura en el segundo MSR.

El registro de los contadores de rendimiento puede ser utilizado para rastrear los eventos Northbridge. Estos incluyen todos los eventos controlados por la memoria, *crossbar* e interfaces HyperTransport. Monitorizando eventos Northbridge se debería realizar solo para un core. Si es seleccionado usando uno de los registros Performance Event-Select en algún core del procesador multi-core, entonces el evento de rendimiento se puede seleccionar en el mismo registro Performance Event Select de algún otro core.

Se debe tener cuidado cuando medimos Northbridge u otros eventos no específicos del procesador bajo condiciones donde el procesador puede entrar en modo de parada durante el periodo de medición. Por ejemplo, se puede desear monitorizar el tráfico de la DRAM producido por la actividad de un disco o un adaptador gráfico sobre la DMA. Esto implica correr algún código de monitorización en el procesador, este código será accedido al principio y al final de la medición, o periódicamente en cada muestra de los contadores. Este código detiene el procesador entre cada intervalo de medida. Si no hay nada más en el sistema operativo para ese procesador en ese momento, se detendrá el procesador hasta volver a usarlo. Por lo tanto, el reloj del contador será detenido y se contarán eventos que no interesan. Para prevenir esto, se correrá un proceso de baja prioridad en background que mantendrá al procesador ocupado durante el periodo de interés.

Bits	Description
63:42	Reserved.
41	**HostOnly: host only counter**. Read-write. 1=Events are only counted when the processor is in host mode.
40	**GuestOnly: guest only counter**. Read-write. 1=Events are only counted when the processor is in guest mode.
39:36	Reserved
35:32	**EventSelect[11:8]: performance event select**. Read-write. See EventSelect[7:0].
31:24	**CntMask: counter mask**. Read-write. Controls the number of events counted per clock cycle. 00h The corresponding PERF_CTR[3:0] register is incremented by the number of events occurring in a clock cycle. Maximum number of events in one cycle is 3. 01h-03h When Inv = 0, the corresponding PERF_CTR[3:0] register is incremented by 1, if the number of events occurring in a clock cycle is greater than or equal to the CntMask value. When Inv = 1, the corresponding PERF_CTR[3:0] register is incremented by 1, if the number of events occurring in a clock cycle is less than CntMask value. 04h-FFh Reserved.
23	**Inv: invert counter mask**. Read-write. See CntMask.
22	**En: enable performance counter**. Read-write. 1= Performance event counter is enabled.
21	Reserved
20	**Int: enable APIC interrupt**. Read-write. 1=APIC performance counter LVT interrupt is enabled to generate an interrupt when the performance counter overflows.
19	Reserved.
18	**Edge: edge detect**. Read-write. 0=Level detect. 1=Edge detect.
17	**OS: OS mode**. Read-write. 1=Events are only counted when CPL=0.
16	**User: user mode**. Read-write. 1=Events only counted when CPL>0.
15:8	**UnitMask: event qualification**. Read-write. Each UnitMask bit further specifies or qualifies the event specified by EventSelect. All events selected by UnitMask are simultaneously monitored. Unless otherwise stated, the UnitMask values shown may be combined (logically ORed) to select any desired combination of the sub-events for a given event. In some cases, certain combinations can result in misleading counts, or the UnitMask value is an ordinal rather than a bit mask. These situations are described where applicable, or should be obvious from the event descriptions. For events where no UnitMask table is shown, the UnitMask is not applicable and may be set to zeros.
7:0	**EventSelect[7:0]: event select**. Read-write. This field, along with EventSelect[11:8] above, combine to form the 12-bit event select field, EventSelect[11:0]. EventSelect specifies the event or event duration in a processor unit to be counted by the corresponding PERF_CTR[3:0] register. The events are specified in section 3.14 [Performance Counter Events]. Some events are reserved; when a reserved event is selected, the results are undefined.

La principal diferencia del registro de AMD con el registro de Intel, es que en el primero se amplía el campo EventSelect, utilizando los bit del 7-0 y del 35-32. Estos campos se combinan para formar el campo de evento de selección de 12-bit. EventSelect especifica el evento o la duración del evento en una unidad del procesador para ser contado por el registro PERF_CTR[3:0] correspondiente. Algunos eventos están reservados; cuando esto ocurre se selecciona, los resultados son indefinidos.

BIBLIOGRAFÍA

(1) Performance Analysis and Monitoring Using Hardware Counters:

http://developers.sun.com/solaris/articles/hardware_counters.html

(2) Intel® Performance Counter Monitor:

http://software.intel.com/en-us/articles/intel-performance-counter-monitor

(3) PCM-power utility:

http://software.intel.com/en-us/articles/intel-performance-counter-monitor

(4) Linux 2.6.38. Cross References:

http://lxr.linux.no/linux+v2.6.38/

(5) Solaris Cross References:

http://src.opensolaris.org/source

(6) Planificación, creación y liberación de procesos:

http://sopa.dis.ulpgc.es/ii-dso/leclinux/procesos

(7) Benchmarks SPEC CPU2006 Documentation:

http://www.spec.org/cpu2006/Docs/

(8) *Access the Linux kernel using the /proc filesystem,* M. Tim Jones, 2006-03-14:

http://www.ibm.com/developerworks/linux/library/l-proc/index.html

(9) *The Linux Kernel Module Programming Guide*, Peter Jay Salzman, Michael Burian, Ori Pomerantz, 2007-05-18 ver 2.6.4.

(10) *Transparencias asignatura AISO*, Manuel Prieto Matías, Universidad Complutense de Madrid.

(11) *Linux Kernel Development*, Robert Love, Addison Wesley, 2010 3rd Edition.

(12) *Professional Linux® Kernel Architecture,* Wolfgang Mauerer, Wiley Publishing, Inc.